BUILT-IN TESTS

FOR A

REAL-TIME EMBEDDED SYSTEM

by

PETER ANDREW OLANDER

VOLUME I

Submitted in partial fulfilment of the
requirements for the degree of
Master of Science,
in the
Department of Computer Science,
University of Natal
1991

Durban
1991

ABSTRACT

Beneath the facade of the applications code of a well-designed real-time embedded system lies intrinsic firmware that facilitates a fast and effective means of detecting and diagnosing inevitable hardware failures. These failures can encumber the availability of a system, and, consequently, an identification of the source of the malfunction is needed. It is shown that the number of possible origins of all manner of failures is immense. As a result, fault models are contrived to encompass prevalent hardware faults. Furthermore, the complexity is reduced by determining syndromes for particular circuitry and applying test vectors at a functional block level.

Testing phases and philosophies together with standardisation policies are defined to ensure the compliance of system designers to the underlying principles of evaluating system integrity. The three testing phases of power-on self tests at system start up, on-line health monitoring and off-line diagnostics are designed to ensure that the inherent test firmware remains inconspicuous during normal applications. The prominence of the code is, however, apparent on the detection or diagnosis of a hardware failure.

The authenticity of the theoretical models, standardisation policies and built-in test philosophies are illustrated by means of their application to an intricate real-time system. The architecture and the software design implementing the idealogies are described extensively. Standardisation policies, enhanced by the proposition of generic tests for common core components, are advocated at all hierarchical levels.

The presentation of the integration of the hardware and software are aimed at portraying the moderately complex nature of the task of generating a set of built-in tests for a real-time embedded system. In spite of generic policies, the intricacies of the architecture are found to have a direct influence on software design decisions. It is thus concluded that the diagnostic objectives of the user requirements specification be lucidly expressed by both operational and maintenance personnel for all testing phases. Disparity may exist between the system designer and the end user in the understanding of the requirements specification defining the objectives of the diagnosis. It is thus essential for complete collaboration between the two parties throughout the development life cycle, but especially during the preliminary design phase. Thereafter, the designer would be able to decide on the sophistication of the system testing capabilities.

PREFACE

The practical work described in this thesis was carried out at UEC Projects (Pty) Ltd., Mt. Edgecombe, Natal from November 1988 to April 1990 under the management of Mr Ian Harris and the supervision of Mr Douglas Law-Brown.

These studies represent original work by the author and have not been submitted in any form to another University. Where use was made of the work of others it has been duly acknowledged in the text.

ACKNOWLEDGEMENTS

The author is indebted to the management of UEC Projects (Pty) Ltd. for providing the opportunity and privilege to submit the work contained herein. Special thanks must be extended to the staff of the Displays and Personnel Departments, but in particular to the following people :

Ian Harris whose support rendered this thesis possible.

Douglas Law-Brown whose expertise as an electronics engineer and competence as a project leader proved extremely valuable and to whom a great deal of credit is due.

Michael Baudin who was responsible for the hardware design of the card cage and spent a significant amount of time and effort in explaining the intricacies of this design.

Michael Stege who laid the foundation within the department with respect to the research into built-in test policies and philosophies.

Alexander Polmans who assisted in the design and development of portions of the code.

Athol M^{c}Clean and his team who were responsible for the hardware design of the console. In particular, thanks must be given to Nitin Mehta whose assistance in understanding the design and whose aid in the development of the applicable software is greatly appreciated.

Grant Robertson and colleagues at the Personnel Department who arranged the financing through the UEC Projects (Pty) Ltd. Study Assistance Scheme.

In addition, thanks must be bestowed upon Professor Alan Sartori-Angus of the Computer Science Department, University of Natal, Durban who supervised the writing of this thesis.

LIST OF CONTENTS

APPENDICES

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

AKBM	Alphanumeric Keyboard Module
ALU	Arithmetic Logic Unit
APCC	Applications Processor Card Cage
API	Applications Processor Interface
BCD	Binary Coded Decimal
BIT	Built-In Tests
BITE	Built-In Test Equipment
BTU	Bus Terminator Unit
CCITT	International Consultative Committee for Telegraphs and Telephones
CPU	Central Processing Unit
CRC	Cyclic Redundancy Code
CS	Code Segment
CSPM	Control and Status Panel Module
DRAM	Dynamic Random Access Memory
EMAC	Environmental Monitoring and Control
EPROM	Electrically Programmable Read Only Memory
FAHD	Fan Housing Drawer
FET	Field Effect Transistor
FTMP	Fault Tolerant Microprocessor
GDU	Graphics Display Unit
GIM	Graphics Interface Module
GKS	Graphics Kernel System
HDLC	High-Level Data Link Control
IP	Instruction Pointer
LED	Light Emitting Diode
MIDS	Manual Input Devices
MPSC	Multi-Protocol Serial Communications Chip
MSCC	Mass Storage Controller Card
MSU	Mass Storage Unit
PIC	Programmable Interrupt Controller
PIT	Programmable Interval Timer
POST	Power-On Self Test
PSU	Power Supply Unit
PVAM	Protected Virtual Address Mode
RAM	Random Access Memory

RBM	Rollerball Module
ROM	Read Only Memory
SCMB	Serial Communications to Multibus
SCSI	Small Computer Systems Interface
SDB	System Data Bus
SIFT	Software-Implemented Fault Tolerance
SKM	Softkey Module
SS	Subsystem
SSR	Solid State Relays

1 INTRODUCTION

1.1 SCOPE

A real-time system usually consists of two distinct parts, namely the controlled system and the controlling system. The controlled system is the hardware which makes up the system and interfaces with the environment. If the controlling system is designed to cause the controlled system to respond to any changes in this environment, then it must be able to do so within a reasonable time. The definition of "a reasonable time" depends entirely on the application of the system. For example, a thermostat controlling the temperature of a room need not react as instantaneously as a missile that is flying towards a mobile target.

In most microprocessor applications, the integrity of the controlling system is extremely important. Real-time systems must ensure a high availability and it is essential that any fault should be reported as soon as possible. Ideally, a real-time system will perform in a fault-free manner throughout its life time. In reality, however, this is rarely the case because hardware systems fail when their components degrade. During execution of the applications task of the real-time system, such failures could cause unexpected situations to arise. It is important, therefore, that any real-time system should possess total control over all possible eventualities and situations that may arise in its environment, whatever the cause. Correctness and completeness are vitally important in the design of any real-time system and suitable decisions should be made by the controller to cater for these unexpected situations. The more catering that is done for such situations, the greater the system availability factor.

The availability requirements can vary considerably from application to application. Continuous operation of computers aboard unmanned missiles or spacecraft is essential to a successful mission, since defective units cannot be repaired. In many space or military applications, the personnel are already burdened with more urgent tasks, and repairing a faulty computer system should be a quick and easy process. Most commercial computers, however, are accessible to maintenance personnel. Although it is inconvenient if a failure occurs, it is not catastrophic, and may be repaired with relatively little cost involved.

Maintenance and availability have become increasingly important in the design of real-time systems, due mainly to labour cost increases and the particular application of the systems designed. More automatic methods have had to be provided to detect and diagnose faults in modules, thus enabling less skilled technicians to repair defective units more quickly. Since hardware failures are a practical reality, methods have been devised to improve system availability. In addition to catering for

unexpected situations during the execution of the applications task, system designers are aiming to include techniques to detect execution errors. These errors may be caused by numerous factors and system designers are faced with the problem of deciding whether or not to include any form of testability in their system, and to what extent and level of sophistication.

In cases where software is an integral part of the real-time system, diagnostic routines have been written that isolate faulty hardware. The problem, however, is that errors could be caused by hardware and/or software and are often not discrete in nature. In addition, increasingly complex integrated circuits impose severe limitations with regards to accessibility, thus making the problem of test generation more and more difficult. A significant aspect of designing for testability, especially in the early planning stages, is the specification of features to be incorporated in the hardware that are designed to aid and enhance the diagnostic capabilities of the system.

The decisions made will depend ultimately on the design under consideration. The requirements specification relating to the real-time system should ideally describe the action that the system is to take for every possible situation that may arise. All requirements of the design should be carefully analyzed and appropriate decisions implemented based on these specified requirements. The generation of such a specification is probably the most difficult part of the design of any system, and generally a well-written requirements specification results in a well-designed system.

The writing of a suite of built-in tests for a computer system requires a real understanding of a large and complex set of interrelated subjects. These include such fields of study as reliability, maintainability, availability and testability. A brief discussion of factors encroaching upon these areas is necessary in order to present the applicable built-in test philosophies and concepts. A detailed discussion of the above engineering fields is, however, beyond the scope of this study, and, where necessary further references are cited.

Test philosophies presented are pertinent to the two phases of system operational testing and system maintenance testing. The tests are aimed at facilitating rapid system repair during a mission or at a repair centre. Although the theory pertinent to the generation of test patterns at the electronic level are not considered in any detail in this text, they do form a foundational basis, particularly in the study of design for testability. Some excellent references [CHANG et al, 1970; BREUER and FRIEDMAN, 1976; KRAFT and TOY, 1981] provide insight into essentially four techniques that have been used

extensively, viz. the path sensitising technique, the so-called D-algorithm, the Boolean difference method and Poage's technique. Some of these classical testing methodologies have been extended to a higher level of integrated circuitry, but the detail and complexity involved in applying the theory renders these approaches impractical, particularly during operational testing.

The theory presented works to pave the way for the implementation of practical techniques. In the writing of the text, the author has attempted to model faults that are likely to occur in a typical real-time embedded system, and to decide on a philosophy to detect such faults.

1.2 PREVIOUS STUDIES IN THE FIELD OF BUILT-IN TESTS

Since about 1965, test generation for logic circuits has been the subject of extensive research. It is one of the oldest areas of the study of fault-tolerant computing, a broad discipline that encompasses all aspects of sophisticated computer design. Diverse areas of fault-tolerant study range from failure mechanisms in integrated circuits to the design of robust software. The main objective of most of these studies has been to develop a systematic approach to the derivation of test procedures that will expose all manner of faults in typical logic circuits. Generally, the philosophy has been to apply a test sequence of input vectors to a particular logic circuit that causes the output(s) to differ from the normal, fault-free condition.

High reliability and availability in computer design was first achieved through so-called fault avoidance techniques, which involved computer design that used high-quality, thoroughly tested components. Sometimes, simple redundancy techniques were used to achieve limited fault tolerance. Automated recovery techniques were seldom used as there was little confidence in the hardware. The drastically increased reliability requirements, together with the increased computer speed quickly made manual recovery obsolete. For example the 1964 *Saturn V* launch computer had a reliability requirement of only 0.99 for 250 hours compared to the late 1970s FTMP (fault tolerant microprocessor) and SIFT (software-implemented fault tolerance) computers that had reliability requirements of 10^{-9} failures per hour over the 10 hour mission time [PRADHAN, 1986, p. xiii; SIEWIOREK, 1986, p.460].

Circuit testability, however, has to continually undergo rapid revamping to keep pace with the nearly revolutionary changes in circuit technology. While the number of components that can be supported on a chip is increasing, the chip itself is becoming susceptible to a more diverse variety of failures, ranging from internal open circuits and shorts to encapsulation and bonding failures. Research into these problems have resulted in a sounder understanding, yielding newer fault models, such as bridging faults, stuck-at faults and crosspoint faults.

Carefully formulated modelling techniques for treating fault diagnosis became firmly established in the mid-1960s [BRULE et al, 1960; JOHNSON, 1960; POAGE, 1963; CHANG, 1965; KAUTZ, 1968; KIME, 1986]. Since hardware was composed of discrete and small-scale integrated circuits, diagnosis of faults to the individual gate or line level was of particular interest. For this reason, many models dealt with stuck-at-faults in combinational networks. As the number of gates per board increased, larger line-replaceable units were created, and diagnosis to board level became an important element in the overall diagnosis philosophy. Efforts were made to design system diagnosis procedures and to analyze test data in a systematic manner [AGNEW et al, 1965; FORBES et al, 1965; HACKL and SHIRK, 1965; CHANG and THOMIS, 1967; KIME, 1986]. Early theoretical work [PREPARATA et al, 1967; RAMAMOORTHY, 1967; KIME, 1970; KIME, 1986;] and moves to employ redundancy at higher levels were motivated by these efforts of systematic approaches. Evidence of this is portrayed in dynamic redundant fault-tolerant systems that have switchable modules at board or component level, rather than at on-chip registers.

In the late 1970s, IBM introduced the so-called level-sensitive scan design technique which permits access to the internal nodes of a circuit without requiring a separate external connection for each node accessed. Since this is made possible at the cost of additional logic circuitry used primarily for testing, designing for testability has evolved to a stage of receiving far more attention as a design criterion than was initially envisaged.

Over a period of ten to fifteen years, various fault diagnosis models were developed, often appearing to be unrelated to each other. It has been shown [KIME, 1979], however, that if the concepts of "fault" and "test" are treated from a hierarchical point of view, a new model can be derived that is broad enough to encompass past diagnostic modelling efforts as well as many contemporary diagnosis practices.

Over the last few decades, several factors have influenced the computer industry and these recent trends have been the major driving forces behind the research of reliability, maintainability, availability and testability. The major factors are:

a) Systems have been exposed to harsher and more polluted environments resulting in greater fluctuations in temperatures, humidity and electromagnetic interference. Consequently, the need for more robust hardware has arisen.

b) Due to the increasing computer market size, the majority of users have become less knowledgeable regarding the actual operation of the system. This results in systems needing to be more user-friendly and tolerant to abuse.

c) Escalations in labour costs and decreases in hardware costs have also had an enormous effect on the industry. The greatest impact here is that it often becomes cheaper to replace faulty hardware, rather than to troubleshoot it.

d) Systems have become larger, resulting in more components and thus the increased overall failure rate. To hold these overall failure rates at an acceptable level, the failure rates of the individual components have had to be decreased and more redundancy has had to be introduced.

e) More competitive computer markets have resulted in manufacturers needing to supply better and more reliable products.

Most of the conducted research has arisen from the practical implementation of existing systems. These systems may be identified in terms of four gradings of availability requirements. Firstly, at the bottom of the scale are the general-purpose commercial systems, where availability is not of paramount importance. This is followed by high availability systems, where the loss of a single user is acceptable, but a system-wide crash is not. In these systems, research has shown [SIEWIOREK, 1986] that limited diagnostic fault coverage should ideally be implemented. The second-highest availability grading is that of long-life systems, such as those of the STAR and Voyager spacecraft systems. In these systems, diagnostic coverage is extensive and redundancy is high.

At the top of the scale is that of real-time control systems, where faults can jeopardise human life or have an immense economic impact. In such cases, correctness and preciseness is extremely important. The development of systems such as SIFT designed by SRI International and FTMP by Draper Labs [SIEWIOREK, 1986], both of which were designed for real-time control of aircraft have led to studies and implementation of software and hardware designs that achieve ultra-high reliability and availability. The approach taken in FTMP was to design special hardware with concurrent error detection capabilities so that incorrect data never leaves a faulty module, whereas SIFT had to mathematically prove the correctness of the system. Besides this, the SIFT project had to solve a number of challenging problems, including distributed clock synchronisation and reaching consensus on system health in the presence of faults [WENSLEY et al, 1978; SIEWIOREK, 1986, p.460-463].

In a somewhat similar manner to SIFT, the space shuttle software [COOPER and CHOW, 1976; SKLAROFF, 1976; AWST, 1981; SIEWIOREK, 1986] consists of five voters, all responsible for guidance, navigation, control, system management, payload management, pre-launch and pre-flight checks. During critical missions, four of the five computers execute redundant tasks, while the fifth performs non-critical tasks and acts as a backup to the primary system. Each computer compares its own output with the other three via software. If a disagreement is detected, the disagreeing output is signalled out, and a vote is cast regarding the disagreement. If the disagreeing unit is voted out, then the redundancy management unit removes its computer from service. A total of three failures can be tolerated. To minimise the probability of a common software bug, two independent companies were responsible for the writing of software packages that met the same requirements specification.

1.3 OBJECTIVES

A theoretical investigation into built-in test policies and philosophies and the illustration of the practical implementation of these techniques provides useful experience for the design of built-in tests for future systems. The objectives in presenting the material contained herein are:

(a) to provide a base for a general understanding of relevant built-in test policies

(b) to investigate the built-in test requirements for a real-time embedded system

(c) to illustrate that features designed to aid and enhance the diagnostic capabilities of a system should be specified during the preliminary design phases of the development cycle

(d) to show that the development of a comprehensive built-in test philosophy enables the coherent development of the built-in test capabilities

(e) to provide an example of the integration and practical implementation of the researched principles during the development of a typical real-time embedded system

(f) to illustrate the complexity involved in the generation of a set of built-in tests for a typical hardware system

(g) to demonstrate the feasibility of generating a practical set of tests that are both comprehensive and unobtrusive during normal operation, whilst simultaneously allowing a high degree of availability to be maintained

(h) to show the attaining of the built-in test requirements by means of fault simulation on the real-time embedded system

The concepts presented in this material may be extended quite readily to the design of various types of small computers that have different architectures, special-purpose digital controllers or large-scale processors designed with high availability features. In particular, the text has been written as a design guideline for the practising electronic engineer or computer scientist developing embedded tests for a small computer system.

2 THEORY

2.1 INTRODUCTION

This chapter outlines some of the theoretical aspects pertaining to hardware failures and the corresponding faults that these failures may cause. Specifically discussed are some of the reasons why hardware failure is a practical reality. The text includes an overview of the basic theoretical issues relating to the modelling of faults together with techniques applicable to software tests on hardware.

2.2 FAILURE MODES

Typically in the development of a large project, the system will experience considerable reliability growth. During early development, the achieved reliability of a new design is much lower than its final predicted reliability because of initial engineering deficiencies as well as manufacturing flaws [ANDERSON et al, 1982, p. 28:12]. The reliability growth of a hardware item is a test-fail-correct process with the aim of designing the hardware to realise its full reliability potential. Engineers will analyze hardware in a manner aimed at identifying basic faults at the part level and determining their effects at higher levels of assembly. This iterative documented process, known as failure mode effects and corrective actions, can be performed with actual failure modes from field data, hypothesised failure modes derived from design analyses, reliability-prediction activities or experience of how parts fail.

In addition to providing insight into failure cause-and-effect relationships, the failure mode and effects analysis provides the disciplined method for proceeding part by part through the system to assess failure consequences. Failure modes are analytically induced into each component and the severity and frequency of occurrence are evaluated and noted. The entire process of introducing failure modes into a system causes a snowball effect, since the resulting failure effect becomes, in essence, the failure mode that affects the next higher level. If the process is iterated for all failure modes, the ultimate effect may be established at system level. In this fault-tree analysis, failure at the root may thus be caused by a failure mode at any one of several leaf nodes.

The use of computational techniques to analyze basic faults, determine failure-mode probabilities and establish criticalities allows for the formulation of corrective suggestions, which, when implemented will eliminate or minimise critical faults. This corrective action is most effective during preliminary system design and after final design before full-scale production is under way. In order to achieve a high system reliability and availability, it is essential that the hardware design engineer, the diagnostic software engineer and the reliability engineer work together to identify and eliminate all potential failure modes.

2.3 PHYSICAL COMPONENT FAILURES

Part failures can be broadly divided into two classes, namely malfunctioning manufactured components and malfunctioning operational components. Although most assembly lines do have quality control checks on delivered items, some malfunctioning parts will inevitably escape initial detection. The classes are therefore not exclusive, since faulty manufactured components may only fail during the mission.

Manufacturers are very concerned, however, about reducing the incidence of fabrication-related failures in order to increase the yield and thus lower the cost of integrated circuits. For example, recently high-density memory chips are being manufactured with redundant elements that can be switched into operation to replace faulty elements by the use of current pulses or a laser beam [ABBOTT, 1981; SUD, 1981; ABRAHAM and AGARWAL, 1986, p. 6].

2.3.1 Malfunctioning manufactured components

Manufacturing defects can be caused by several factors, such as defects in oxide pinholes, photoresist or etching defects, conductive debris, scratches, weak bonds or partially cracked chips or ceramics. Some causes of manufacturing defects are cited below [ABRAHAM and AGARWAL, 1986, p. 6-7]:

a) Defects in the crystalline structure of silicon cause devices in the region of the defect to be faulty.

b) Improper doping profiles can result in devices with unwanted characteristics, which in turn can result in intermittent or permanent faulty behaviour.

c) Poor encapsulation can result in the penetration of moisture into the package, resulting in long-term corrosion-related failures.

d) Impurities in the packaging could result in low-level radiation which can lead to run-time data on the chip being lost.

e) Aluminum metal can be subject to corrosion, resulting in long-term failures.

f) High current densities in thin wires can result in metal migration, eventually resulting in a break in the wire.

g) Formation of spurious "whiskers" can lead to resistive shorts.

h) Migration of alkali ions can cause a shift in the thresholds of transistors, which will manifest themselves as intermittent failures, eventually becoming permanent.

i) High field intensities can impart energy to electrons, causing these "hot" electrons to move into and be trapped in the gate oxide in MOS transistors, also causing shifts in threshold voltages.

j) The defects may also be caused by assembly operations or even an assembly line worker's learning, motivation, fatigue or negligence.

Failures of integrated circuits due to electrostatic discharge during assembly operations have become more significant as technological advances have produced devices with smaller geometries and thinner dielectrics. Studies [ANDERSON et al, 1982] have shown that CMOS and linear integrated circuits are the most sensitive to improper handling which can result in input gate breakdown caused by static electricity.

2.3.2 Malfunctioning operational components

Operational malfunctions occur as a result of system operation, where simply the old age of the part causes the physical or chemical structure of electron devices or components to deteriorate. Systems may also be called upon to operate beyond their design capabilities due to an unusual mission requirement, which could be detrimental to the functioning of the system and cause part failure.

Degradation in inherent reliability can also occur as a result of excessive handling from frequent preventative maintenance or poor corrective maintenance activities. The replacement of analog by digital circuits, aims at reducing the requirement for frequent preventative maintenance, while the substitution of faulty hardware in a real-time system at a board level tends to reduce poor corrective maintenance activities.

2.4 CIRCUIT FAILURES

Circuit faults may be classified as logical or parametric. A logical fault is one that causes the logic function of a circuit element or an input signal to be changed by some other logic function, whereas a parametric fault alters the magnitude of a circuit parameter, causing a change in some factor such as circuit speed, current or voltage [FUJIWARA, 1985, p. 8].

It has been established [ANDERSON et al, 1982] that semiconductor circuit malfunctions generally arise from two sources, namely transient circuit disturbances and component burnout. The transient spike or "hazard" is more common because it occurs at much lower energy levels.

Malfunctions associated with timing are due mainly to circuit delays. Although all inputs to asynchronous circuitry should be constrained to change only when the memory elements are in stable conditions, delay faults could affect the timing operation of the circuit. Since an asynchronous circuit is assumed to have a finite, positive delay, bad circuit design can result in the generation of both transient spikes or race conditions. If either is input to a flip-flop or Schmitt trigger, or interpreted as a control signal, this may result in an incorrect state.

2.5 FAULT MODELLING

Problems arise when attempts are made to formalise and classify all manner of faults that are likely to occur in a set of electronic devices. The discussion above has shown that several types of failures can occur both during the manufacture and operational life-time of a component. The types of failures that are seen also vary with technology. For example, bipolar circuits usually have a higher device power dissipation, are more likely to exhibit hot spots and temperature-related problems, and are also prone to metal migration problems because of high current densities.

As technology changes and existing problems are solved, new problems arise, thus making it impossible to list all possible failures that can be used to derive tests for circuits. Another problem with obtaining information pertaining to long-term failures is the fact that the upgrade rate for integrated circuits is so high that many new devices have not been in use for very long.

It can be seen that a practical approach to testing should avoid working directly with physical failures, due to their large number and complex nature. Generally speaking, people are not interested in the detail to basic device level of the cause of a failure, and some level of abstraction is needed. A physical failure that changes the function of a circuit can be detected by applying an appropriate sequence of tests to the input and monitoring the output for errors. When tested with fault models

of this nature, many different physical failures may cause the same error. If, however, the fault model accurately describes all the physical failures of interest, then it is only necessary to derive tests that detect all the faults of the fault model. In this way, the number of primitive entities to be considered in deriving a test is reduced considerably. Quite often, the details of the physical failures are unknown or are too numerous to consider and a fault model can be hypothesised to cover most of the possible failures. Another advantage of this approach is that the tests may become generic and totally independent of the current technology.

When designing hardware or software with test capabilities in mind, it is necessary to consider what is most likely to fail. To provide an answer to this question is, of course, not trivial. The types of faults that are known to occur from past experiences need to be considered and applied to the various functional blocks. For example, an n-p-n transistor implementing an inverter with an open base or collector would cause the output on the collector to be permanently high or stuck-at-1. Conversely, a short-circuit between the collector and the emitter would cause the output to be permanently low or stuck-at-0.

Short-circuits (known as bridging faults) generally result in stuck-at-faults. These faults are usually the most common, but it has been revealed [FUJIWARA, 1985, p. 8-12] (i) how a CMOS two-input NAND gate with p-MOS and n-MOS FETs can cause combinational circuits to become sequential and (ii) how extra or missing devices in programmable array logic circuits at a particular crosspoint in the array can upset the logical output. These may or may not be able to be modelled by the classical stuck-at-fault.

In determining a set of input vectors for a nonredundant logic circuit with n inputs, it is possible to adopt a brute force philosophy by applying all 2^n possible input combinations to the circuit. As n becomes large, however, this approach becomes extremely inefficient. Examination of the physical behaviour of the circuit reveals a smaller set of tests which will be sufficient to detect all likely occurring faults. As an example, consider a two-input OR gate with output function $F = A + B$. The analysis of typical stuck-at faults for input A is depicted in Table I. As can be seen from this illustration, faults on input A will only be detected when all inputs are zero and when A is high and the other input is low.

Table I : Analysis of the output function of a two-input OR gate with typical stuck-at faults.

Inputs *AB*	Input *A* stuck-at	Expected output	Actual output	Fault detected
00	0	0	0	No
00	1	0	1	Yes
01	0	1	1	No
01	1	1	1	No
10	0	1	0	Yes
10	1	1	1	No
11	0	1	1	No
11	1	1	1	No

Thus, for faults on a two-input OR gate, the input test vectors are:

(i) $A = 0, B = 0$: tests all stuck-at-1 faults

(ii) $A = 0, B = 1$: tests input B for stuck-at-0

(iii) $A = 1, B = 0$: tests input A for stuck-at-0

By induction, for an n input OR gate the input vectors will be (i) all inputs low and (ii) a "walking" one on each respective input line with the rest of the $(n - 1)$ inputs low. Similarly, for an n input AND gate the input vectors will be the complement of the test vectors used in checking the OR gate. The test for a NOT gate is trivial, because this is simply the complement of the input.

2.6 TESTING METHODOLOGIES

There are basically three techniques that can be applied to the generation of a test set for circuitry, namely structural level test generation, functional level test generation and random testing. Firstly, the structural level test generation aims at testing the circuitry mainly during the fabrication process. Each line of a module is tested for classical faults at a electronic level. This method is therefore not applicable to the operational testing of an embedded system and is not considered any further here. Functional level test generation and random testing methodologies are, on the other hand, applicable to the design under consideration and are discussed below.

2.6.1 Functional level testing

Since all logic circuits may be decomposed into NOT, AND or OR functions, it is possible to test for the classical stuck-at-faults. For complex systems where the majority of the circuitry is comprised of integrated circuits, the process of decomposing all circuitry into these basic functions would be an impossible task. In other complex systems where no integrated circuitry exists, the decomposition may still be far too difficult to achieve. A higher level of testing needs to be established and agreed upon. Ideally, the implementation of built-in tests only need to resolve circuit behaviour to functional block level. Although many of the internal gates of functional blocks are not directly influenced by variations in I/O signals, these would be tested indirectly by the varying I/O signals of juxtaposed gates. Thus, if a functional block is tested with sufficient input signals to test *all* possible functional operations of that block and the output corresponds to what is expected for each input, then the functional block is operating in a fault-free manner. This approach greatly reduces the complexity of the task of writing a suite of built-in tests for any system. The objective of testing complete system integrity is still achieved, however, provided that the built-in tests check *every* possible functional operation of each block.

The functional testing approach is often the only method by which some circuits are tested. It is very difficult to grade the quality or fault coverage of these tests, however, and it has been observed [ABRAHAM and AGARWAL, 1986, p. 50] that in many cases these tests do not detect existing failures. This is because the faults could cause the system to apparently perform its function correctly, while simultaneously performing other spurious tasks. As an example, if a multiplexer contains a stuck-at or bridging fault, an incorrect data line (in addition to a correct one) could be selected. The performance of these spurious tasks are undesirable and it is important that the test procedure detect such side effects.

Two rules have been laid down [ABRAHAM and AGARWAL, 1986, p. 50] that should be followed when using a functional block testing approach which will detect the most likely physical failures, viz.

(a) A test must verify that no unintended function was performed together with the intended function.

(b) Information about the structure and the faulty behaviour of the system in addition to the functional information should be available in order to generate tests of a reasonable length for a complex system.

Unfortunately, for a complex system, the number of possible erroneous functions to be checked is extremely large, and the test would be both prohibitive and complicated. If no information is available regarding the structure or faulty behaviour, it becomes imperative to reduce the size and complexity of the test, which places the amount of fault coverage in jeopardy.

For the reasons discussed, the term "functional-level test generation" is defined to mean an approach whereby a higher level fault model is first derived for the system under test to include the most likely circuit failures. Test sequences are then derived to detect faults in this fault model. Structural information about the system is utilized to reduce the size of the test set when this theory is applied to the system.

The functional-level test generation is illustrated in the hierarchy of Figure 1. Tests can be generated in relation to any level of the hierarchy, provided that the test detects a significant portion of the faults potentially present at the lowest level in the hierarchy. Test result details can help to more clearly separate faulty components for a variety of lower-level faults that can occur. This detail can be lost, however, in moving up a level in the hierarchy, so careful thought must be given by the diagnostic designer in deciding on circuit subdivisions.

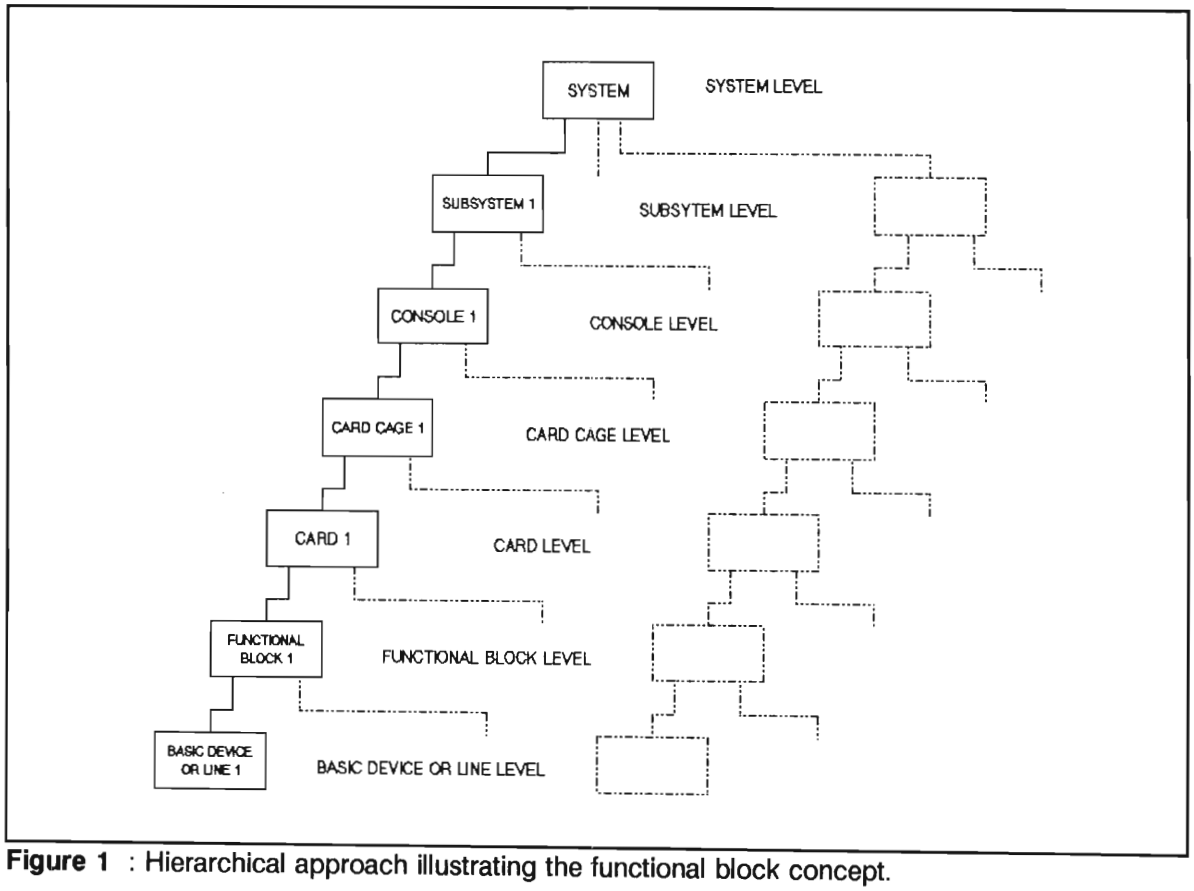

Figure 1 : Hierarchical approach illustrating the functional block concept.

The fault model for the functional block consists of a fault set FS = $\{f_1, f_2, ..., f_m\}$, for *m* possible faults, where f_i is present if the *i*th unit is faulty [KIME, 1986]. To simplify the model, an assumption is made that only one fault is present at any given time, and that this fault is "solid" (i.e. determinate). The fault pattern set is therefore F = $\{F_0, F_1, ..., F_m\}$, where $F_0 = 0$ and $F_1 = \{f_1\}$, $F_2 = \{f_2\}$, etc.

Tests are modelled from the same hierarchical viewpoint as faults. The levels in the test hierarchy, to some degree, coincide with those of the fault hierarchy. A test set for a fault set FS is denoted by TS = $\{t_1, t_2, ..., t_n\}$, where t_i is test *i* for a set of *n* tests. A test pattern TP = $\{T_0, T_1, ..., T_p\}$ (where $p \leq n$) is defined as a subset of TS consisting of the set of all tests that failed on application of test set TS. The set of all test patterns that are expected to occur is T = $\{T_0, T_1, ..., T_n\}$.

A vector notation may be used to describe test patterns. In this notation, $T_v = \{t_1, t_2, ..., t_n\}$, where $t_i = 1$ if the *i*th test fails, and $t_i = 0$ if the *i*th test passes. For example, in the test pattern $T_{eg} = (0,1,0,1)$, test t_1 and test t_3 have passed, while tests t_2 and t_4 have failed.

Figure 2 shows an example of a functional block, consisting of duplicated computational blocks whose outputs (OUT_1 from the first logic block and OUT_2 from the second logic block) are compared by a self-checking equality comparator. Data from an external source is buffered and the control circuitry provides control signals for the computational blocks. The equality comparator, which is used for concurrent fault detection during normal operation, is used here to compare the outputs of the computational blocks under the applied test. The tests to be considered are software tests applied by hardware external to the functional block.

The fault and test patterns are related to each other via a fault-pattern-test-pattern event space. To illustrate the relationship, suppose that in Figure 2 the fault pattern set F is restricted to consist of a good functional block plus functional blocks containing exactly one of the faults of the fault set FS = $\{f_1, f_2, ..., f_6\}$, i.e. F = $\{F_0, F_1, ..., F_6\}$.

The test set is TS = $\{t_1, t_2, ..., t_6\}$, where each t_i corresponds to testing for each f_i. Determining the associated test patterns is achieved by examining the behaviour of each test for each fault pattern. The event space is then formulated as per Table II and Table III. As an example, assume that the control circuitry is faulty. On OUT_1, the results of tests t_1 and t_3 will be indeterminate. Similarly, the results of tests t_2 and t_4 on OUT_2 will also be indeterminate. Test t_5 will register a failure on both outputs, while the comparator test t_6 will not be affected and will consequently pass.

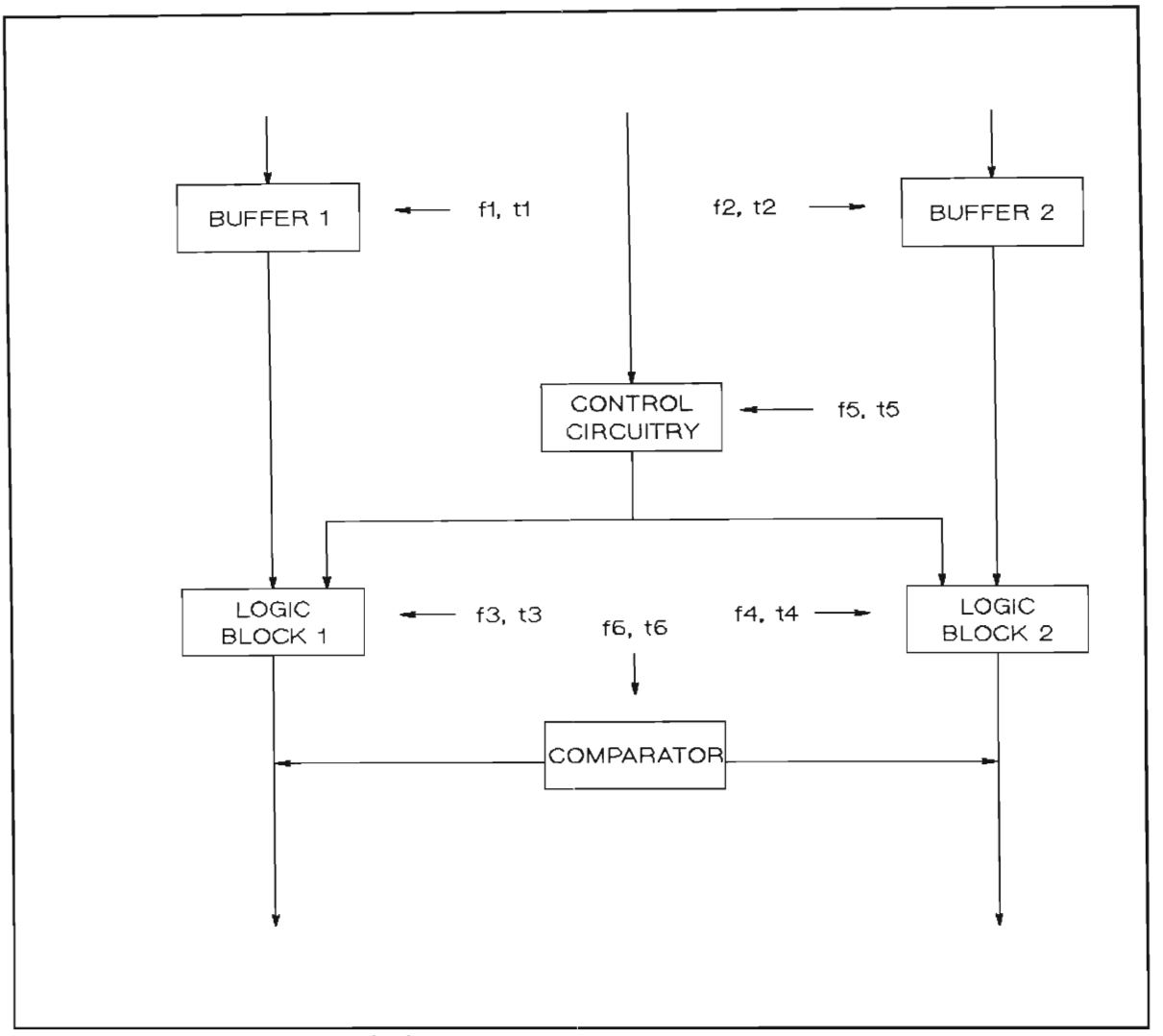

Figure 2 : Typical functional block

Table III may be used to perform diagnosis by assembling the actual test results into a test pattern, known as a syndrome. The syndrome is then compared with each of the tabulated test patterns. If a match occurs, the fault pattern corresponding to the test pattern may be present. For example, if the syndrome is 111110, then the corresponding fault pattern indicates that fault f_5 is present, and the syndrome has enabled the diagnosis of faulty control circuitry. However, if the syndrome is 101010, then it is indeterminate whether fault f_3 (logic block 1 faulty) or fault f_5 (control circuitry faulty) is present.

Table II : Fault table for a typical functional block.

Faulty circuitry	Fault pattern	Test pattern (OUT_1) $T_{v1} = (t_1t_3t_5t_6)$	Test pattern (OUT_2) $T_{v2} = (t_2t_4t_5t_6)$
None	F_0	0000	0000
Buffer 1	F_1	1X00	0000
Buffer 2	F_2	0000	1X00
Logic Block 1	F_3	X1X0	0000
Logic Block 2	F_4	0000	X1X0
Control logic	F_5	XX10	XX10
Comparator	F_6	0001	0001

0 = Test passed ; 1 = Test failed ;
X = "Indeterminate"

Table III : Combined fault table for a typical functional block

Faulty circuitry	Fault pattern	Test pattern (OUT1 and OUT2) $T_v = (t_1t_2t_3t_4t_5t_6)$
None	F_0	000000
Buffer 1	F_1	10X000
Buffer 2	F_2	010X00
Logic Block 1	F_3	X010X0
Logic Block 2	F_4	0X01X0
Control logic	F_5	XXXX10
Comparator	F_6	000001

0 = Test passed ; 1 = Test failed ;
X = "Indeterminate"

The syndrome for the functional block may be written as a Boolean expression from the fault table of Table III. This yields the syndrome :

$$
\begin{aligned}
S = \quad & f_1 (t_1\, t_2'\, t_4'\, t_5'\, t_6') \\
& + f_2 (t_1'\, t_2\, t_3'\, t_5'\, t_6') \\
& + f_3 (t_2'\, t_3\, t_4'\, t_6') \\
& + f_4 (t_1'\, t_3'\, t_4\, t_6') \\
& + f_5 (t_5\, t_6') \\
& + f_6 (t_1'\, t_2'\, t_3'\, t_4'\, t_5'\, t_6)
\end{aligned}
$$

[where t_i' = not (t_i)]

In order to see how this expression may be used in diagnosis, suppose that the test results are $t_1 = 1, t_2 = 1, t_3 = 1, t_4 = 1, t_5 = 1$ and $t_6 = 0$. The syndrome expression yields $S = f_5$, indicating the control circuitry faulty. Alternatively, for test results $t_1 = 1, t_2 = 0, t_3 = 1, t_4 = 0, t_5 = 1$ and $t_6 = 0$, $S = f_2 + f_5$, showing that either of these faults may be present.

The theory presented above can effectively be applied to represent any fault-pattern-test-pattern event space. However, some real diagnosis problems arise in the attempt to formulate this event space, including test completeness, test invalidation, and test result information loss. A "complete" test is one that fails for exactly one fault from the set present, while passing for all absent faults [KIME, 1986, p. 587]. Therefore, a test must detect all underlying faults in the hierarchy that manifest themselves in the fault under consideration. As an example, consider buffer 1 of Figure 2 . If the test t_1 for this buffer was designed to test for all data line stuck-at faults, and instead a bridging fault occurred between two lines, then there is a strong possibility that this fault would go undetected. Thus, if bridging faults are assumed to occur at a lower hierarchical level and the test was not designed to detect them, then the test is incomplete. In general, test completeness, although assumed in the theory presented, is difficult to achieve in practice.

Another problem is that of test invalidation. A complete test is defined to be "valid" if it always fails when one or more of the faults for which the test is complete exists, and always passes when all faults for which the test is complete are absent. What this definition is saying is that, ideally, the result of a test on one or more faults is not influenced by the presence or absence of other faults. Unfortunately due to circuitry interaction, this is usually not the case in reality.

A third diagnostic problem is that of information loss. In performing any form of diagnostics, there is a distinct trade-off between the complexity involved and the amount of information lost. As an example, suppose that two tests t_1 and t_2 have the same results for fault pattern F_1 (00 or 11) and complementary results for fault pattern F_2 (01 or 10). If this was the only method of distinguishing between the two faults and this fact was overlooked by the modeller (i.e. the test results were modelled as "don't cares" for both fault patterns), then important test result information would have been lost in the diagnostic design process.

2.6.2 Random testing

The studies of functional-level test generation have exposed several problems associated with the practical implementation of the theory. Another approach, which can be applied independently or together with the functional approach, is that of random testing. This is probably the simplest approach to the test generation problem. Instead of basing test vectors with specific fault models in mind, the tests are chosen independently according to some fixed probability distribution and applied to the circuit under test. The functional operation of the circuitry is considered, and a test is generated based on this functional operation. The output is then compared with an expected error-free reference vector.

Although simplifying the task of generating a suite of tests, this method immediately poses three queries, namely:

(a) What is the number of test vectors to be randomly applied to a particular circuit ?
(b) What is the size of the fault coverage for a particular circuit ?
(c) What is the level of confidence that can be placed on the test results ?

The determination of the length of the random sequence that is required to obtain a satisfactory fault coverage can entail an analysis which can be more complicated than the deterministic test generation procedure being replaced [SHEDLETSKY, 1977; ABRAHAM and AGARWAL, 1986, p. 72]. Although an analytical method of estimating fault coverage would be preferable, it has been revealed [SHEDLETSKY, 1977] that none of the techniques yet developed are capable of getting accurate fault coverage estimates without a great deal of computation. During system development it may be necessary, therefore to use fault simulation techniques to determine fault coverage.

2.7 SUMMARY

This chapter has provided an introductory discussion of failure modes and the physical failure of components resulting from defects during manufacturing or failure during operation. In addition, a discussion of parametric and logical circuit faults was also provided. The formalisation and classification of all manner of faults for a complex real-time embedded system was shown, however, to be impractical. It was established that a better approach to generating tests is to apply some level of abstraction to the system under test. For this reason, the discussion dealt with a hierarchical approach to fault modelling and test generation. An example was introduced in an attempt to convey the complexities involved in designing a suitable test for typical circuitry. The example illustrated that the number of input test vectors that detect a known classical fault for a particular circuit may be reduced considerably if the physical behaviour of the circuit is examined.

Three testing methodologies were introduced, namely that of structural-level test generation, functional-level test generation and random testing. The electronic level of the structural-level test generation technique rendered this method inapplicable.

A fairly lengthy discussion was presented regarding the functional-level testing methodology. Applying this technique, the designer decomposes the system under test until a functional block remains. All classical faults in this functional block then form a fault set. Under the assumption that only one fault is present at any one particular instant, the test set is generated. Each test is designed to test for a corresponding fault. Consideration is given to the logic output corresponding to the application of each test and a fault-pattern-test-pattern event space is constructed. The syndrome that enables fault diagnosis from the fault and test patterns applied to the functional block may either be determined from the fault-pattern-test-pattern event space or from the resulting Boolean expression.

Random testing, on the other hand, considers the test generation problem from a more statistical viewpoint. Tests are chosen independently according to some fixed probability distribution. A brief discussion of the advantages and disadvantages of this approach was included in the text.

Several problems associated with the testing techniques that face the diagnostic designer were described. These included test completeness, test invalidation and test result information loss. In addition, queries concerning the determination of the number of test vectors, establishing the fault coverage and determination of the level of confidence in the test results were posed.

The testing methodology applied will depend ultimately upon the system under consideration. A logical approach is to combine the theories of functional level test generation and those of random testing in order to devise non-deterministic test vectors for some cases. Although the functional test generation technique should be the main approach, some of the classical faults should also be borne in mind in designing the tests. This would enable a higher level of testing to be applied whilst simultaneously maintaining a reasonable level of confidence in the system integrity.

3 PHILOSOPHIES

3.1 INTRODUCTION

Up to this point, not much has been said concerning the policies and philosophies that should be adhered to throughout the development of a system with testable features. This chapter aims to introduce and describe those idealogies. The built-in test features may be divided into three modes of operation, viz. (i) Power On Self Tests at system start up, (ii) on-line diagnostics and health monitoring and (iii) off-line diagnostics. Philosophies applicable to each mode are discussed in the ensuing sections.

Standardisation policies are discussed at both system and component levels. The component level standardisation discussion presents theoretical issues and generic routines pertinent to the generation of tests for common hardware components.

3.2 BUILT-IN TEST PHILOSOPHY

A real-time system must be able to provide a service that can be closely defined in terms of guaranteed minimum mean time between failures and mean time to repair. If faced with a hardware failure, the system should, if possible, provide a useful degraded service that still enables quick and easy fault diagnosis. Built-in test equipment (BITE) and built-in tests (BIT) are designed in such a manner that systems may test themselves, providing both diagnostic capabilities and a confidence check of the system. BITE is the hardware built into the circuitry that provides fault detection, whereas BIT is the fault detection or diagnostic routine that utilizes BITE and/or test sequences to facilitate the task of locating defective units.

The incorporation of BIT and BITE in system design is essential in detecting and diagnosing all manner of failure modes in both the development and applications of a real-time system. In order to maintain a high system availability during a mission, the trend in system design is to incorporate enough testability that checks the entire system integrity, whilst simultaneously not causing an excess on system overhead.

Both BITE and BIT apply a sequence of input patterns that produce erroneous responses when faults are present and then compare the responses with expected ones. In order to design a complete test sequence for a specific circuit, the test conditions must be oriented toward checking the circuit at the level of the components themselves, rather than at the level of the microinstruction set.

The existence of fault models at the various levels of a hierarchical design is useful for finding the effects of faults and deriving tests in a hierarchical fashion, treating the testing problem in much the same way as the original design problem. Principles normally associated with the design for testability theory are applicable to the test generation process in such a case. Two guidelines should be used throughout the implementation of the BIT for a real-time system. Firstly, it should be attempted to adhere as far as possible to the enhancement of controllability and observability in the division of circuits into subcircuits, since it has been established [FUJIWARA, 1985, p. 145-149] that the testability of a circuit is closely related to these two factors. Secondly, the top-down approach of "divide and conquer" should be incorporated. The system as a whole should be partitioned into smaller and smaller constituents until a functional block remains. It has been shown [FUJIWARA, 1985, p. 145] that the computer run time to generate tests is approximately proportional to the number of gates to the power of three. Thus, dividing a circuit in half, reduces the BIT task to one eighth for each of the two subcircuits.

3.2.1 Power On Self Test (POST)

To ensure confidence that the system will perform reliably before entering a normal operational environment, initial "sanity" tests should be performed by the resident firmware. This start-up testing serves to validate the hardware and provide the operator with advisory notifications of critical hardware failures that would prevent normal system operation. Tests should be run in a sequence that examines as much circuitry as possible as quickly as possible.

At system start up, the embedded firmware should execute diagnostics on the integrity of the real-time system in an "expanding kernel" manner, i.e. basic critical tests should be initially performed, followed by further tests based on the former results. Certain functional units should be defined to be critical, and, in the event of failure, the error should be reported, if possible, and the system halted. If a non-critical unit fails a power-on self test, it should ideally report the failure to the operator, but operation should be able to continue at the discretion of the operator.

Normally in a real-time embedded system, a single board computer is executing as master, and in the "expanding kernel" fashion this card would initially test the processor ROM and RAM followed by other on-board peripheral chips. The basic idea is to start evaluating a small section of the processor card and to expand the diagnostic process to include more and more hardware with each succeeding step until the entire card has been completely tested. If no critical faults have been detected, off-board tests should be performed in a logical order. Ideally, these off-board tests should be to poll the presence status of the various cards in the system, followed by integrity tests based upon the presence results. Intelligent units should simultaneously perform their own POST and report their

results to global or dual ported RAM locations. These locations, together with units equipped with BITE circuitry that can report aspects of their operational status may then be interrogated and their status recorded. Usually some degree of checking can be applied to external devices and interfaces. For example, it may be possible to loop outputs back to inputs, thus testing through the interface packages to the peripheral side.

Ideally, no tests should be performed that rely on the functionality of untested units. The bootstrap approach of testing the integrity of the core of the system first can be based on one of two philosophies. Firstly, the designer may build the core using sufficiently reliable components that the BIT may assume the core to be fault-free. With this approach, the BIT may neglect the testing of the core and immediately proceed to examine the rest of the system. In general, this is an extremely poor assumption, because all electronic components have a finite life-time. The alternative and more realistic philosophy is to specify that the core must first be validated. Ideally, however, the philosophy employed in the implementation of BIT for a system should be a combination of the above two approaches. Therefore, the integrity of the processor, ROM and RAM should all be tested, and the components used in the design of the core should comply with relatively high reliability requirements. In a real-time embedded system it is not possible to test the functionality of the core without relying on the functionality of untested units, and, hence some faith must be placed in the core of the system being operational. Thus, in order for the system to test itself, quite a bit of the hardware must already function correctly. In this sense, the POST does tend to be of rhetorical significance, but, if the system passes all tests, it does provide the operator with confidence that the system is indeed operational. Beyond this, however, it should be realised that system self tests do have severe limitations in some cases.

The POST is required to test as many of the system functions and interfaces as possible, while simultaneously minimising the time from system start-up to full operational status. If the POST is suspicious with regards to the correct operation of certain hardware functions, these should be reported, since this indicates a potential problem. The onus should then be on the operator to perform more extensive off-line diagnostic testing.

3.2.2 On-line diagnostics and health monitoring

Whilst continuing to perform its main applications task, a real-time system should be performing some form of health monitoring, especially in circumstances where availability is of paramount importance. In these environments, for example, railway signalling, telephone switching or process control, space or military applications, triple redundancy schemes have been designed. In such a configuration, three complete systems operate in synchronism and periodically compare results. Democracy rules in such an instance, because any unit which produces a result at variance with the results of the other two is out-voted and ignored. In some sophisticated systems, limited corrective measures are taken by the BIT, pending operator intervention, and the out-voted system either disconnects itself from the network, or is disconnected by one of the other two. In nearly all redundancy configurations, maintenance action can be undertaken even if the system continues operation by means of its back-ups. Standby machines do not contribute any useful work except in emergencies. In order to gain full advantage of the redundancy and to improve availability, any fault should be reported immediately so that repairs can commence as soon as possible.

The disadvantage associated with on-line testing is the overhead involved in interrupting the main applications task of the system. This overhead must be weighed against the possibility of a failure occurring in the interval during which the test sequences are not being run. Such a failure will not be detected by the BIT, and the system may produce erroneous results. To minimize the likelihood of this happening, the frequency of the health monitoring tests may need to be increased. Some systems [KRAFT and TOY, 1981, p. 212-214] avoid the need to increase the frequency of testing by implementing a "duplication and match" philosophy. In such instances, redundant systems act as "shadow processors", matching critical outputs. When these outputs disagree, fault diagnosis routines are called to establish which system is at fault.

In addition to the firmware, the hardware also performs on-line diagnostics by means of integrated circuitry designed for the automated detection and indication of fault conditions (BITE). BITE can be designed to provide immediate notification of hardware failures as they occur. The operator should be aware of the techniques of fault indication employed by the system hardware, as these may be by visual or audio alarms [STEGE, 1988].

It can be seen from the above discussion that two major classes of redundancy exist, namely active redundancy and passive redundancy. The latter requires that external elements detect, make a decision, and switch to another element or path as a *substitute* for a failed element or path, whereas for active redundant circuits this is not the case. The decision to use redundancy must be based on a careful analysis of the trade-offs involved. Usually the introduction of redundancy increases safety

and mission reliability and availability, but reduces the mean time between failures. When methods of part improvement are shown to be more expensive than duplications or when other ways of improving system availability have been exhausted, redundant design techniques may be the only answer. It may be advantageous to use a redundant design when preventative maintenance is desired with no system down-time or when maintenance is impossible (e.g. an unmanned missile). In such cases, the prolonged real-time response caused by the redundancy needs to be carefully considered. Other disadvantages that also need consideration are increases in weight, space, cost, design time and complexity, which, in turn, results in an increase of unscheduled maintenance actions.

3.2.3 Off-line diagnostics

The off-line diagnostics are usually executed via menu-driven dialogue at an operator I/O device. They may be run individually, in sequences, cyclically, or interactively under the control of the operator. The operator may initiate off-line diagnostics at any time to confirm or analyze more closely a suspected failure detected by the POST or on-line health monitor. An advantage that off-line testing has over the background health monitor is the facility of more rigorous and controlled test conditions, because the module under test is isolated from operational mode conditions which may or may not be confusing the health monitor. Another advantage is that the off-line tests may also be used to commission a newly replaced module before it enters operational service.

Once a faulty unit has been positively identified by either the BITE or BIT, no other diagnostic action may be required, since the fault may be corrected by simply replacing the unit. The use of single boards to house the associated electronics of large-scale integrated circuits that implement processors and memories has aided the philosophy of isolating faults to line replaceable units. In manned systems where the skilled maintenance personnel are not readily available, the operator will have the responsibility of repairing a fault which may arise. If the standard repair procedure to be observed by the operator when a fault occurs is the replacement of the circuit board containing the fault, then the both the system maintenance and the fault resolution are considerably simplified. It may still be necessary, however, for the fault resolution realised by the diagnostic routine to proceed one level deeper and to test all functional units on the board. If a fault occurs and is diagnosed to a single board, repair by replacement is a quick, simple and highly cost-effective approach to servicing a fault. A faster repair time for the faulty unit, together with the cost savings of not having to develop additional fault diagnosis hardware and software, justify the cost of replacing an entire circuit board. Alternatively, the faulty circuit board may be handed over to maintenance personnel at a later stage who may then execute further diagnostics to a greater resolution.

This hierarchical view of faults can be very simply illustrated with regards to the goals of a diagnosis. Suppose that the goal of diagnosis is to identify a faulty board in a subsystem. Once the board is removed from the subsystem, the new diagnosis goal is to locate the faulty physical component on the board. When the level of the faulty component is reached, it is unnecessary to perform further diagnosis because no lower level of repair is possible. The lower levels in the hierarchy are still important, however, because the accomplishment of the series of diagnostic goals depends on the *detection* of a fault at the lowest level, such as a stuck-at fault on the output of a gate.

After substituting a new module for a faulty one, the integrity of the replacement unit must also be verified, as there is no guarantee that it is compatible with the system. Ideally, the system would have been shut down during unit changes, implying that on power-up, the POST will once again be performed. However, it may be necessary for the complete system containing the new unit to be exercised exhaustively to ensure that no further faults exist.

In all of the above three BIT and BITE categories, the BITE may operate together with the BIT or independently of it. If operating independently, non-software addressable BITE circuitry within the hardware could simultaneously monitor and give the operator notification of fault conditions. If, however, the BIT and BITE are operating interactively (i.e. the BITE is software addressable), BIT can poll the BITE status or the BITE can be designed to interrupt the system processor when a fault occurs.

3.3 STANDARDISATION PHILOSOPHY

When designing a real-time embedded system that is relatively large and that requires a team effort to achieve objectives, it is advantageous to strive for standardisation amongst common hardware and software modules. If cards are standardised (and hence interchangeable), then more line replaceable units will be available in the event of a card failure. For example, suppose that subsystem *x* is executing a critical task where lives and/or property are at stake and an interchangeable unit fails. If there are no more spare units available and subsystem *y* is idle or executing a non-critical task, then the common hardware may be retracted from subsystem *y* and inserted into subsystem *x*. Standardisation not only enables interchangeability of the common hardware modules, but also simplifies and minimises effort during the development cycle.

It has been mentioned in previous discussion that the top-down modular decomposition design approach should be used to define functional blocks to be tested in order to achieve overall system testability. To a certain extent, this classical approach should also be used in the design of the system software. The problem with top-down decomposition and the ensuing stepwise refinement, however,

is that single-purpose functional primitives usually result, that are generally only applicable to the problem statement of the software requirements specification at hand. Although the design meets the initial specification, it is not centred around any identifiable object, and is inevitably not reusable. It is for this reason that the design of the BIT should be to follow current software trends in adopting an object-oriented design methodology. This methodology imposes constraints such as abstraction, information hiding and the formalised encapsulation encompassing the module concept with distinct interfaces and implementations [BALLINGER and CONRADIE, 1990]. The modules then designed are totally generic and exhibit reusability with a large range of applications. When comparing these generic techniques to other ad hoc methods of software design, a trade-off exists. Ad hoc techniques aim at making a given design more testable by a relatively inexpensive method, whereas the generic approach is much simpler, reusable and easily maintainable [FUJIWARA, 1985, p. 144; ALLWORTH and ZOBEL, 1987, p. 113-115].

The standardisation effort can be broadly viewed from two levels in the aforementioned hierarchy. At the first level, common boards both at system and subsystem level should be identified and, if possible, standardised. At the second level, common components on cards should also be identified and an attempt should be made to write standardised "library" routines supporting these common components.

The standardisation philosophy should be applied at all hierarchical levels during the development phase. Provided that the standardised routines have been fully tested and are free of bugs, the overall design effort is considerably reduced. The drawback, however, is that the need now arises for the designer to write a formal interface requirements specification. Fortunately, this drawback sometimes tends to be beneficial to the designer, since it now becomes essential to study interfaces that were possibly overlooked in the original software design. Furthermore, if the number of these common units is fairly large, the reduction in software design effort more than justifies the additional effort of the writing of the specification.

3.3.1 System level standardisation

The identification of an intersection set of modules that are common in an array of *n* subsystems is referred to as the standard computing segment. In designing a standard computing segment for a real-time system, the designer is faced with various options regarding the unique identification of each subsystem. The advantages and disadvantages of each choice need to be carefully considered and the appropriate decision implemented.

Firstly, the designer may regard the hardware and firmware of the standard computing segment to be totally standardised and the applications code to be loaded into RAM from a mass storage unit. This option is similar to that of the classical "personal computer" where applications files are loaded from hard or floppy disks. The advantage of this approach would be a totally standardised console throughout all *n* subsystems. This would be particularly advantageous if the number of subsystems was large and thus justified by bulk production. The disadvantage, however, is that each subsystem has to cater for the hardware needs of all the other subsystems. If the applications are quite diverse, this could have a substantial economic impact.

The second option in defining a standard computing segment is similar to that of the first option. The designer still views the hardware and firmware of the standard computing segment as completely standardised. However, instead of loading the applications code into RAM from a mass storage device, all the various applications are an integral part of the standardised code and the configuration of the subsystem is dynamically reconfigurable by operator intervention. Ideally, the standardised code should interrogate the system to decide what hardware is available and, based upon this decision, offer the operator the choice of the subsystems available. Once again the advantage of adopting this approach would be a totally standardised console throughout all *n* subsystems. The disadvantage of the first option no longer exists since the system knows what hardware is available at power up. Instead of each subsystem having to cater for the hardware requirements of all the other subsystems, the responsibility is now shifted to the firmware. Thus the drawback is a significant increase in the BIT software complexity as this now has to allow for all the various hardware permutations and eventualities that may occur.

Finally, the BIT firmware may require that one card in the standard computing segment allow for the unique identification of each subsystem. Although this implies that the card itself is not standardised, it has the advantage of being the simplest approach. As an example, suppose that a computing segment consists of a main applications processor card, a memory card and a system data bus controller card connected via a Multibus backplane and that all three cards have their own on-board ROM. Furthermore, suppose that at system level, these cards are common to all *n* subsystems that interface to the system data bus, i.e. they form the standard computing segment. In an effort to achieve standardisation and interchangeability throughout these various subsystems, a generic set of routines that are common to all subsystems (referred to as the standardised code) could be written. The memory card could be identified as being common subsystem hardware containing unique subsystem-specific applications code. The standardised code resident on each main applications processor card of all *n* subsystems could then reserve an area of the ROM on the memory card that provides the standardised code with a description of the system configuration.

3.3.2 Component level standardisation

Several devices are common to most intelligent boards and it is appropriate to discuss the application of test philosophies to these common components. Although these components may vary in technology, their purpose is respectively the same, implying that generic test philosophies may be applied to each type of device. It is advantageous to standardise the application of such test procedures, and, in particular those applicable to the testing of the core system of an intelligent board.

Diagnosis of faults cannot be reliably performed if the core itself is not functioning correctly. Ideally, the core system should be tested by an external source that is known to be fault-free. Such an external source could be human or electronic. If the core system is required to be tested by a manual check, however, this is extremely error-prone and certainly not a mature system design approach. In addition, an embedded system interfaces mostly with users that are usually unskilled in electronics as opposed to trained maintenance personnel. Alternatively, if the external source is electronic, the question then arises as to what will test the external source itself. This dilemma causes the system designer to view the design as an isolated and independent entity that has no external source available to test the core. The BIT must therefore rely heavily on the BITE in order to detect any arising fault in the core system.

Even if the BIT is not able to provide a reasonable diagnosis of a fault occurring in the core system, it still performs an extremely useful function inasmuch as providing a reasonable amount of confidence in the correct operation of the hardware. This far outweighs the disadvantage of extra cost and effort in the writing of the software.

One of the objectives in system design is to minimise the amount of hardware in the core system, but still providing a mechanism to facilitate the testing of it. The majority of the core system will include the ALU, the registers, control circuitry and data transfer paths that perform the opcode fetch-execute sequence, the system clock and a small portion of memory. Ideally, the diagnostics that test the major portions of the overall system will be code resident in ROM, thus keeping the core system to an absolute minimum and eliminating the necessity of bootstrapping the BIT from an I/O device.

In a previously developed system, each microprocessor, on-board ROM and on-board RAM was defined to comprise the core system. The philosophy adopted was for all intelligent units resident in the system to test their own core system concurrently at system power up. The core system thus forms an important portion of the entire BIT effort and the theory applied to the testing of the core system is now introduced. The algorithms used to implement the theory are also presented.

3.3.2.1 Test generation for microprocessors

The microprocessor is an extremely complex device and there is an added difficulty because the description of the fault-free microprocessor is itself not elementary. The various fault models that have been suggested [ROBACH and SAUCIER, 1975; ROBACH and SAUCIER, 1978; THATTE and ABRAHAM, 1980; BRAHME and ABRAHAM, 1984; ABRAHAM and AGARWAL, 1986, p. 59-67] should be compared and the appropriate decision taken based on the operational requirements of the user. One proposal presented is to check each instruction in the instruction set. Another suggestion is simply to run applications programs, and, if these execute successfully, to assume that the microprocessor is operating in a fault-free manner. Although the former proposal would certainly be an exhaustive test, this approach is extremely tedious. Adopting the latter approach would, on the other hand, reduce the confidence of the user.

Techniques have also been presented to derive tests for general microprocessors to detect faults in the fault model [PARTHASARATHY et al, 1982; BRAHME and ABRAHAM, 1984]. The approach adopted in developing a generic routine for the microprocessors of a system could be based on these techniques. The fault model of the microprocessor and its various functions together with a definition of the faulty behaviour for the functions is developed at the instruction and register transfer level. Adopting this approach, however, poses the problem of deriving tests for the faults, since in order to check for an instruction or internal state of the microprocessor, other (possibly faulty) instructions must be executed. The faults that are attempted to be detected may be masked by these faulty instructions.

A microprocessor is modelled as a graph where each node represents a register or set of registers. An edge represents data or information transfer. Instructions are modelled as consisting of sequences of microinstructions, with each microinstruction consisting of a set of micro-orders. This conceptual model also applies to non-programmed microprocessors.

The principles presented in the references cited were applied to the Intel iAPX 8086 microprocessor family to produce the data transfer graph together with the associated sample set of micro-orders given in Figure 3 and Table III, respectively.

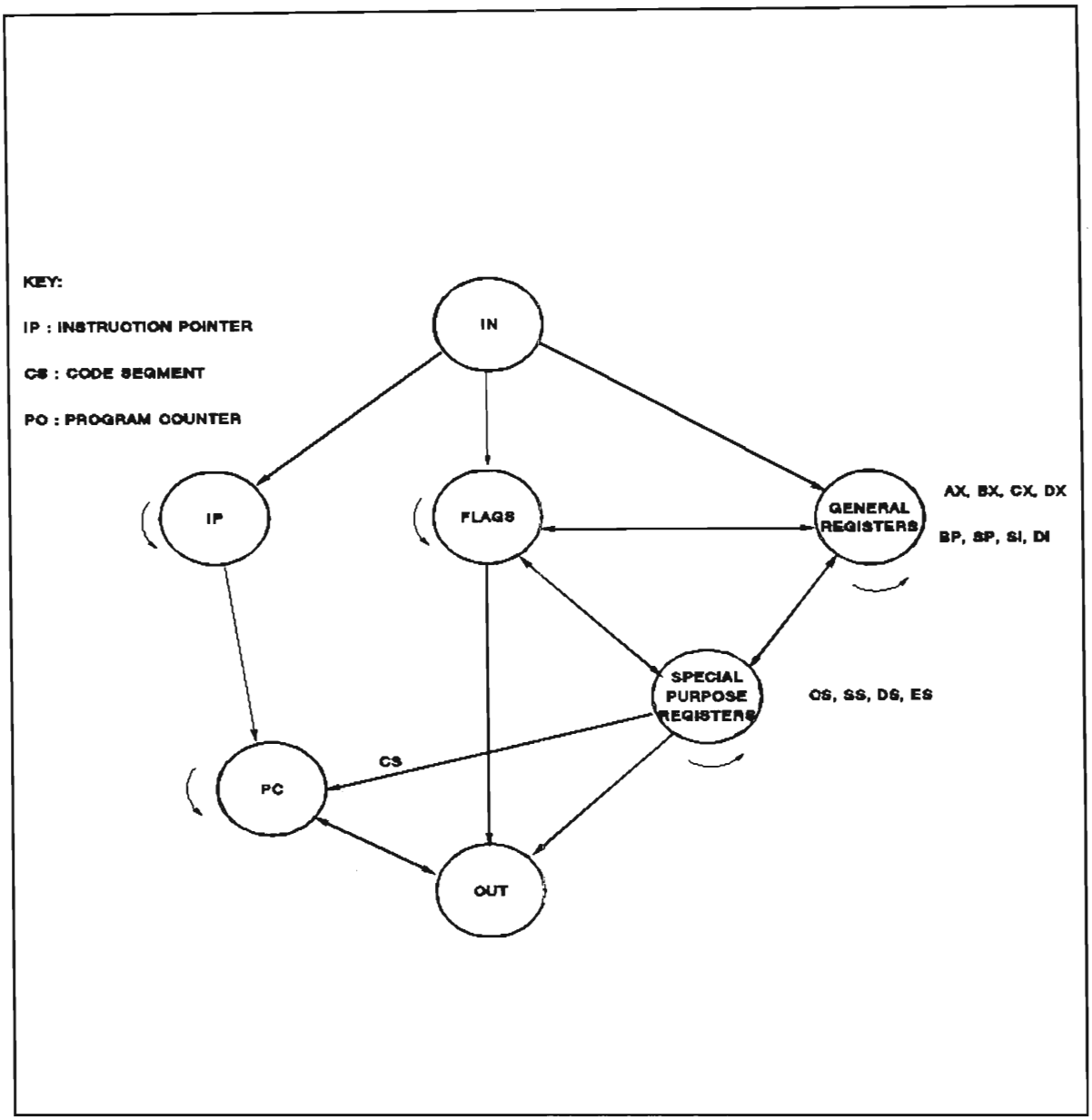

Figure 3 : Data transfer graph representing the Intel 8086 microprocessor family

Table III : Sample set of micro-orders for a typical microprocessor

Type 0 : Clear, Negate, Shift, Rotate
Type 1 : Move, Test, AND, OR, Exclusive OR, Exchange
Type 2 : Add, Subtract, Multiply, Divide

The eight general registers (AX, BX, CX, DX, BP, SP, SI and DI) of the 8086 microprocessor family form a set of equivalent registers, as do the four special purpose registers CS, SS, DS and ES. In addition, the code segment register (CS) is used together with the instruction pointer IP to form the program counter PC. All data I/O is performed through the external interface nodes IN and OUT, respectively. The micro-orders are divided into three types depending on their function. A micro-order is of type 0 if it operates on one register only, type 1 if it involves a transfer of data from one register to another or if it is a logical operation, and of type 2 if it is an arithmetic operation.

The functional blocks for which models are proposed are registers and register decoding, data transfer paths, the arithmetic and logic units and the instruction decoder and control function. For the register decoding and registers, failures may occur in any manner that corresponds to failures in RAM. This fault model is hence considered to be the same as that of the RAM and is discussed later.

Typically a data path consists of the internal data bus, address bus, and input and output busses connected to the ALU. Any number of lines of a data path may be stuck-at-zero or stuck-at-one and any pair of lines could be coupled. Two lines *i* and *j* are coupled if the value of *j* depends on the value of *i*. Furthermore, any number of data paths in a microprocessor may be faulty in any manner described.

The arithmetic and logic unit design varies between microprocessors (even within the Intel 8086 family), depending on the ALU capability as well as the speed of operation. Tests must be derived for a particular design of ALU based on a fault model that is most appropriate for that design. Generally the test consists of instructions to transfer the operands from memory to the source registers, instructions to perform the operation under test, and instructions to read the results from the destination register into memory.

Faults in the instruction decoding and control unit could be generated by one or more of the following events:

(a) The failure in execution of one or more of the micro-orders causes an incorrect or incomplete instruction execution.
(b) A set of incorrect micro-orders that are active in addition to the correct active micro-orders causes spurious execution of microinstructions.

The testing of the instruction decoding and control function is the most difficult part of the test generation procedure for microprocessors. A step-by-step procedure has been outlined [PARTHASARATHY et al, 1982] that can be generically applied if the testing is done from an external

source with the aid of diagnostic tools such as a logic analyzer and an oscilloscope. Due to the "stand-alone" nature of an embedded system, this is not possible, and the test generation algorithm must rely on untested hardware.

Following the principles outlined earlier and other theory that has been set out [PARTHASARATHY et al, 1982], the high-level test generation algorithm for the system-wide generic tests of the Intel 8086 microprocessor family is presented in Table IV. Test generation algorithms for detecting faults in the instruction execution process are based on testing the reading of registers for all registers by executing the reading instructions in a particular order. It should be noted that the read instructions themselves could be faulty.

Table IV : Test algorithm to detect faults in a typical microprocessor

1) Test the core set of instructions, i.e. loading of registers, comparison of registers, absolute jumps and conditional branches.
2) Test the ability to read registers without disturbing other registers.
3) Test the general purpose data movement.
4) Test the status register, i.e. the zero flag, the carry and overflow flags, the parity flag and the sign flag.
5) Test the ability to perform shifts and rotations.
6) Test the ability of the arithmetic and logic unit to perform correctly.
7) To ensure proper procedural transfer of control, test that the stack manipulation is operating in a fault-free manner.

A core set of instructions is first checked, and the remaining instructions are then tested. Table V gives a more detailed presentation of the test for core instructions. Note that in presenting the algorithm, certain assumptions are made, viz.

(a) the number of memory write operations does not increase when a fault exists
(b) when faulty, the maximum number of microinstructions in an instruction does not exceed some finite value M and the maximum number of micro-orders in a microinstruction does not exceed *N*.

In Table V the compare and branch instructions are tested for all conditions. The compare instruction is executed $N * M + 1$ times to ensure that faults will not mask each other. It can be proved [ABRAHAM and AGARWAL, 1986, p.64] that any existing fault will be detected in the worst case after this number of iterations.

Table V : Core instruction set test procedure

```
START:      mov Reg1, #Data1          ; Reg1 = #Data1
            mov Reg2, #Data1          ; Reg2 = #Data1
            cmp Reg1, Reg2
            je  A
            jmp ERROR

A:          ...

            je TEST_LOAD
            jmp ERROR

TEST_LOAD:  mov Reg1, #Data2          ; Data1 <> Data2
            mov Reg2, #Data3          ; Data2 <> Data3
            cmp Reg1, Reg2
            je  ERROR
            mov Reg1, Data1
            mov Reg2, Data1
            je  ERROR                 ; Repeat N * M + 1 times

            ...

            ; The rest of the microprocessor test is executed at this stage.

            ...

            mov AL,1                  ; AL = 1 for a successful test
            jmp FINISH
ERROR:      mov AL, 2                 ; The accumulator is loaded with a two for a fault
FINISH:     ret
```

3.3.2.2 Test generation for ROM

Most ROM tests perform a sum of the bytes stored in the ROM and compare this calculated sum with a stored checksum. This text proposes the implementation of a ROM checksum test of a different nature. The algorithm is based on the polynomial or cyclic redundancy code (CRC) error detecting and correcting theory for data transmission [TANENBAUM, 1981, p. 128-133]. It is extremely important for the ROM to be error-free and this method is proposed due to its relatively high probability of error detection.

Polynomial codes are based upon treating bit strings as representations of polynomials with coefficients of 0 and 1 only. A k-bit binary string is regarded as the coefficient list for a polynomial with k terms, ranging from x^{k-1} to x^0. The CRC-CCITT international standard of $x^{16} + x^{12} + x^5 + 1$ (10001000000100001 binary) is one such polynomial that could be used. Studies have shown

[TANENBAUM, 1981, p. 132] that this polynomial detects all single and double errors, all errors with an odd number of bits, all burst errors of length 16 or less, 99.997% of 17-bit error bursts, and 99,998% of 18-bit and longer bursts.

The ROM bank (with the exception of the checksum itself) should be treated as a long binary number and the CRC-CCITT polynomial then divided into the bank by a process of long division. The remainder becomes the calculated checksum and this is compared to the stored checksum. The application of this theory implies that for all on-board ROM tests, the code will effectively perform a checksum test on itself.

An example of the implementation of the process of long division of the polynomial into the ROM bank is portrayed in the algorithm of Table VI. The bank is divided into its 64 Kbyte constituents and a remainder is calculated firstly on a byte-by-byte basis (procedure CRC_RESULT) and then on a block-by-block basis (procedure TEST_ROM_BLOCK). The result is compared to a stored checksum.

Table VI : Generic ROM checksum algorithm

```
CRC_RESULT

  begin
  Determine the remainder when dividing the incoming byte by the polynomial.
  Return the remainder.
  end

TEST_ROM_BLOCK

  begin
  Iterate the CRC_RESULT calculation for the size of each ROM block.
  Accumulate this remainder for the entire ROM bank.
  Return the remainder.
  end
```

3.3.2.3 Test generation for RAM

The purpose of random access memory is to store a range of bits in each cell. The microprocessor must have the ability to read from or write to each cell without altering the value stored in any other cell. It has been explained [THATTE and ABRAHAM, 1977; NAIR et al, 1978; SUK and REDDY, 1981; ABRAHAM and AGARWAL, 1986, p. 56] that the RAM test needs to be able to test for the faults as set out below:

(a) Test for memory cells that are stuck-at-zero or stuck-at-one.
(b) Test the ability to perform zero-to-one and one-to-zero transitions.
(c) Test for coupled cells.

Several algorithms have been proposed to detect the classical faults which can occur in RAMs [KNAIZUK and HARTMANN, 1977]. These algorithms vary considerably and there is a distinct trade-off between their associated fault coverage and duration of execution. The most exhaustive test is one designed to cause a zero-to-one and a one-to-zero transition for each memory cell with confirmations between transitions that check that the data has been stored correctly. For memory with n cells, this procedure would involve a write to a single cell plus reads from all n cells. The test would have to be repeated n times to check the entire memory range. Empirical evidence has shown [HAYES, 1975; ABRAHAM and AGARWAL, 1986, p. 54] that this exhaustive test is of the order of n^2 in length, which is far too comprehensive to be practically viable, because it would probably exceed the time constraints provided in the requirements specification.

Another approach is to "march" a set of logic highs and lows down the address bus in some sequence and then to repeat this performance with complementary data. Although this test takes a time proportional to n [ABRAHAM and AGARWAL, 1986, p. 54], it does not test the RAM very thoroughly.

The derivation of an algorithm to test a subset of memory cells which provides enough confidence in the integrity of the RAM without weighing too heavily on system overhead is now outlined. The approach generically adopted for a system under test should conform to aforementioned philosophies. Thus the integrity of the data and address busses should be established prior to the testing of the RAM devices. The algorithm is presented in Table VII.

Table VII : Generic system RAM test algorithm

1	Read a predefined location and abort if a time-out occurs.
2	Write data pattern 55 hexadecimal to this location.
3	Read and verify the location.
4	Write data pattern AA hexadecimal to the same location.
5	Read and verify the location.
6	Write data pattern 55 hexadecimal to this location again.
7	Read and verify the location once again.
8	Write to the same location and read and verify the proper execution using hexadecimal data patterns FF, FD, FB, F7, EF, DF, BF and 7F respectively.
9	Write unique patterns to unique addresses.
10	Read and verify the values at the addresses generated in the previous step.
11	Test the RAM itself by writing 55 hexadecimal to each location.
12	Alternately perform a "dummy" write to clear the data lines and read the RAM to verify that all locations are 55 hexadecimal.
13	Write a data pattern of AA hexadecimal to the RAM locations.
14	Alternately perform a "dummy" write to clear the data lines and read the RAM to confirm that each location is equal to AA hexadecimal.

Step (1) of Table VII checks to see whether the RAM that is about to be tested is available in the system. If the RAM is not available (detected, for example, when a time-out occurs) the memory should be marked as absent and the test aborted.

Successful completion of steps (2) to (7) implies that each of the eight data lines have undergone a zero-to-one-to-zero transition, which proves that there are no stuck data lines. Step (8) examines the possibility of shorted data lines by transmitting a "walking zero" down the data bus and, on successful completion, proves that no two data lines are coupled. In the event of failure, the test enables quick and easy identification of shorted lines by means of analyzing the results read.

Steps (9) and (10) validate the address bus by exercising each address line and writing a unique pattern to a unique address. Once the entire address range has been written to, the values are verified by reading the same address range. The generic algorithm is presented in Table VIII and the addresses generated by this algorithm to exercise the first eight address lines are presented in Table IX.

Finally, steps (11) to (14) of Table VII checks the ability of the RAM to store information and tests for possible coupling of adjacent memory cells. It ensures that every bit in the RAM has undergone a zero-to-one-to-zero transition and that the RAM is fully operational.

Table VIII : Address generator algorithm for generic system RAM tests

```
GENERATE_RAM_PATTERN

  begin
  For each 64K RAM block
    begin
    Determine next address
    Increment data counter
    If the action required is to write the data then
      begin
      Write the data
      Check whether a time-out occurred
      end
    else
      begin
      Read the data
      Check whether a time-out occurred
      Verify the data
      end
    end
  end
```

Table IX : Generation of addresses to check the validity of the address lines.

Data (Hex)	Address (Hex)	Comment
00	0000	Initialises address bus
01	0001	Exercises A_0
02	0002	Exercises A_1
03	0004	Exercises A_2
04	0008	Exercises A_3
05	0010	Exercises A_4
06	0020	Exercises A_5
07	0040	Exercises A_6
08	0080	Exercises A_7

3.4 SUMMARY

Philosophies applicable to the generation of a set of tests for a real-time embedded system were discussed in this chapter. In particular, the three modes of power-on, on-line and off-line testing were considered.

Consideration was also given to standardisation policies that should be applied at all hierarchical levels during the system development cycle. The standardisation efforts at system and component levels were especially discussed. Theoretical issues and generic routines relating to the generation of built-in tests for common core hardware components were presented. The generic routines consisted of sample test algorithms for microprocessors and memory banks. An example was given showing the construction of a data transfer graph and its associated micro-orders for a typical microprocessor. All test algorithms were shown to provide enough user confidence in the core system integrity whilst simultaneously not being too exhaustive or weighing too heavily on system overhead.

4 A REAL-TIME EMBEDDED SYSTEM

4.1 INTRODUCTION

It is advantageous to illustrate how the presented theories and philosophies were practically applied to the development of a real-time embedded system. This not only serves to substantiate the idealogies, but also provides a basis for improvement in future designs. This chapter draws on past experiences in the presentation of such practical examples.

It has been noted that in order to generate tests of a practical nature, information about the structure and the faulty behaviour as well as the functional information of a complex system should be available. For this reason, the subsystem was decomposed in a hierarchical fashion to this basic functional level.

The system under study was really one of several subsystems, each designed to perform a dedicated task. The subsystem interfaced to the other subsystems via a system data bus. The overall view is shown in Figure 4.

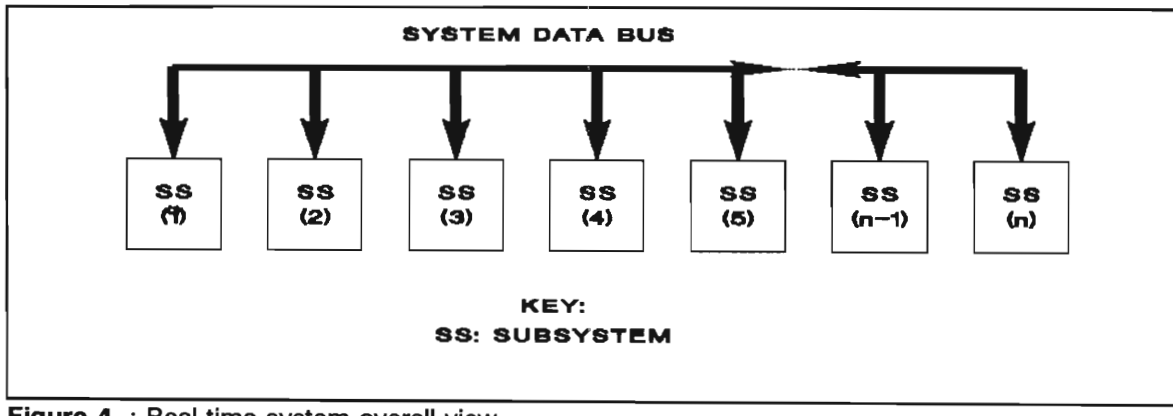

Figure 4 : Real-time system overall view

With reference to the hierarchy described in the previous chapter, the subsystem described (SS(n) of Figure 4 above) was divided further into three consoles as shown in Figure 5. Dual redundancy was implemented at console level, with another console (console 2 of Figure 5) performing an identical function to the system described in this text (console 1 of Figure 5). A third console (console 3 of Figure 5) formed a subset of the console under consideration, and was designed to drive certain peripheral devices, including a printer, plotter, mass storage unit and video cassette recorder. The communication between the three consoles was achieved by means of a local area network.

The discussion to follow focuses on the first console. Firstly, the operator and environmental interfaces are described. This is followed by a look at the console card cages and a presentation of the electronics resident in the main applications processor card cage.

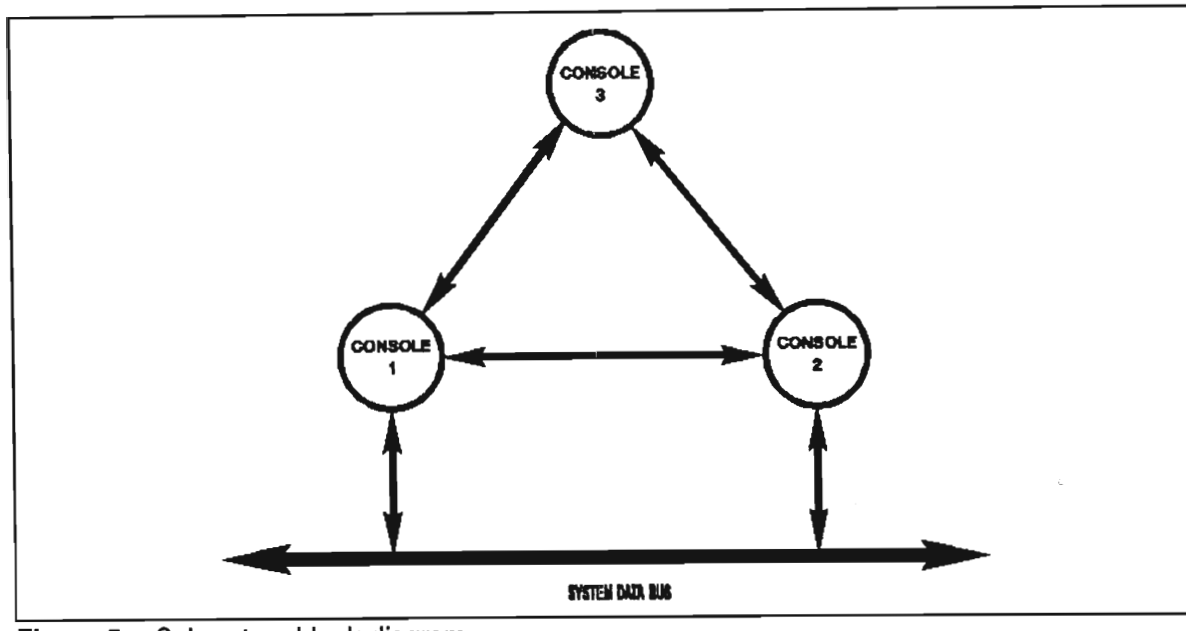

Figure 5 : Subsystem block diagram

4.2 THE CONSOLE

Figure 6 shows a pictorial view of the console that housed the real-time system. The console was designed in a modular fashion, thus enabling damaged parts to be easily replaced. The console consisted of six drawers and was hermetically sealed when all the drawers were fully closed. Three of the drawers each housed a graphics display unit and the other three each housed a card cage [METHA, 1989a; LAW-BROWN, 1990a].

The manual input devices to the console consisted of console control switches, user defined switches, softkey modules, a keyboard and a rollerball. Status and warning indicators, together with three graphics display units catered for the output. The console control and user defined switches were all located on the status and control panel, while each softkey module was positioned on the right hand side of the associated graphics screen. The desk top housed the keyboard and rollerball modules.

The layout of the status and control panel of the console is depicted in Figure 7. A key switch located at the top of the panel required that a key be inserted before the console could be operated. Every other switch had integral backlighting, status and warning indication and could be tested for brightness control via a lamp test switch.

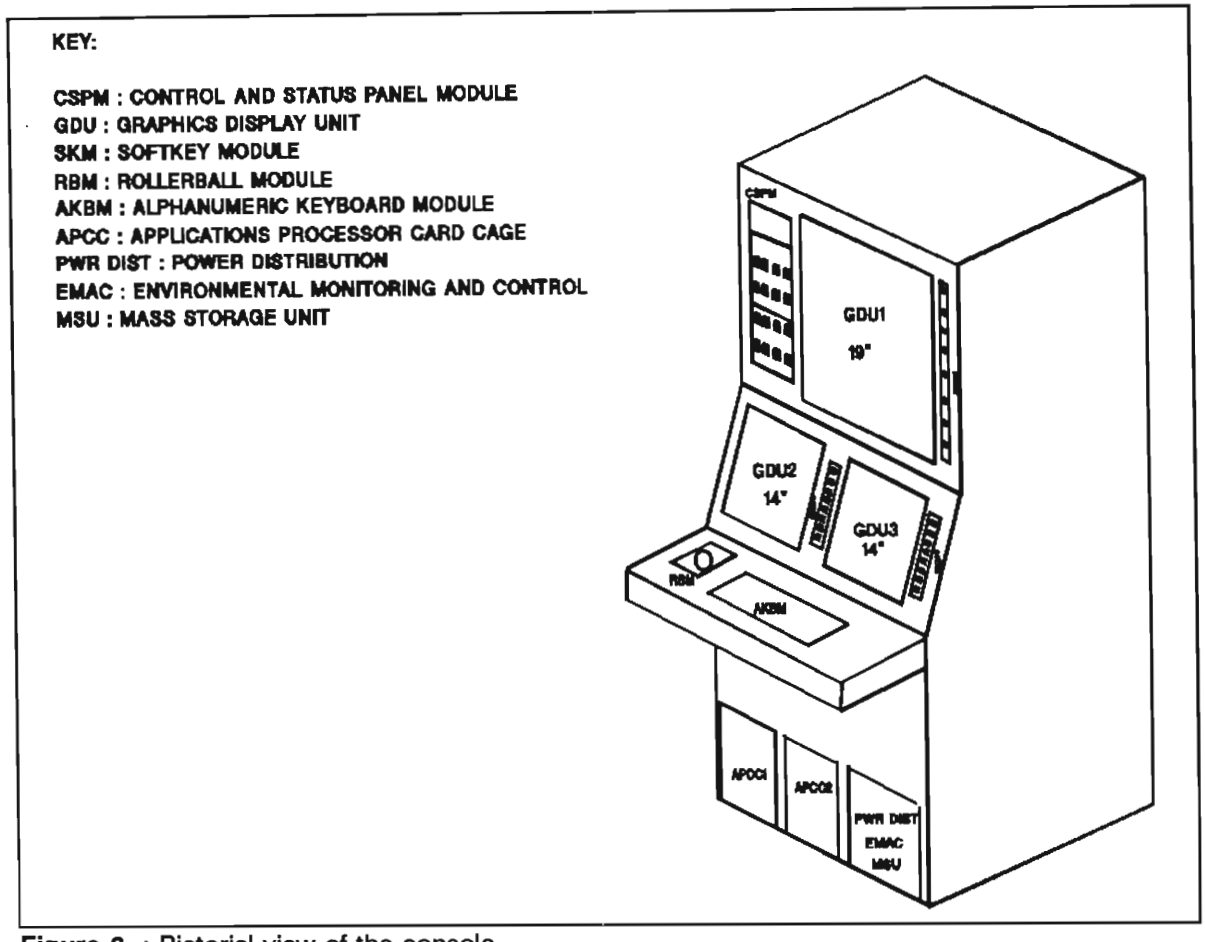

Figure 6 : Pictorial view of the console

The console was powered by a single 220V 50Hz power lead. The presence of power at the input to the console was indicated by a mains neon on the status panel. This power was passed into the console via a contact breaker mounted on the gland and terminal housing unit at the rear of the console. The power was then fed through an array of solid state relays to each respective module in the console. The solid state relays were controlled by environmental monitoring and control circuitry, which applied power to the system only under normal operating conditions. This circuitry was powered up by an auxiliary connected directly to the power supplied by the contact breaker. Figure 8 shows the power routing to the console and an overview of the console elements controlled by the environmental monitoring and control hardware.

Power was applied to the console when the console switch was closed. In order to prevent accidental operation, the switch was encompassed by a protective cover.

The console allowed for the disabling of all the environmental monitoring and control hardware protection mechanisms by means of a system override function. Hence, closure of the system override alternate action switch caused power to be enabled to all elements, irrespective of the monitoring of fault conditions.

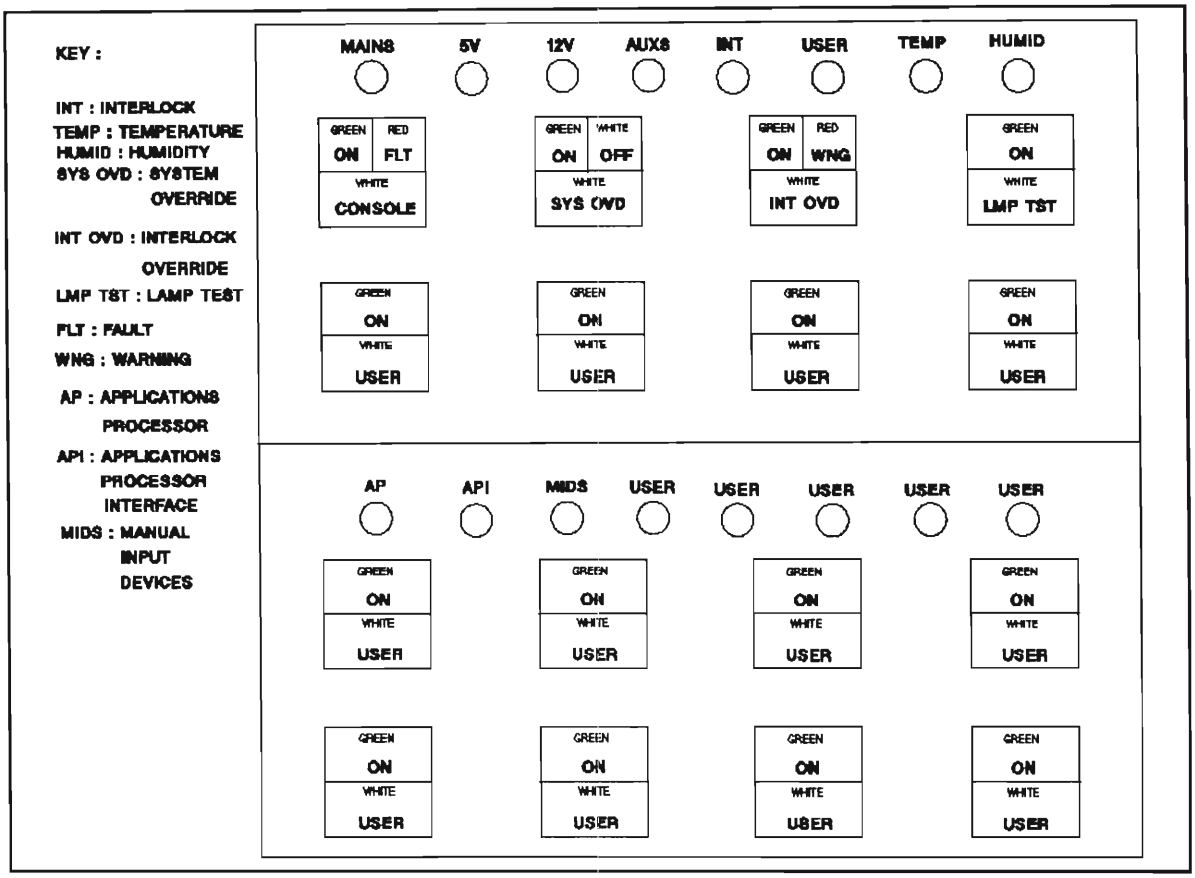

Figure 7 : Status and control panel

All graphics display units and card cages in the console resided in retractable drawers, which, when opened, disabled power to each respective unit. The interlock override function permitted the drawers to be retracted without disabling the power distribution to the appropriate unit, thus catering for maintenance requirements.

The environmental monitoring and control hardware controlled the solid state relays which enabled the power distribution to each respective unit by realising the following Boolean equation:

If A = "Console switch on" and
B = "System override switch on" and
C = "Temperature < 323 degree Kelvin" and
D = "Interlock switch on" and
E = "Interlock switch of respective unit"
then
POWER_TO_UNIT = A and (B or C) and (D or E)

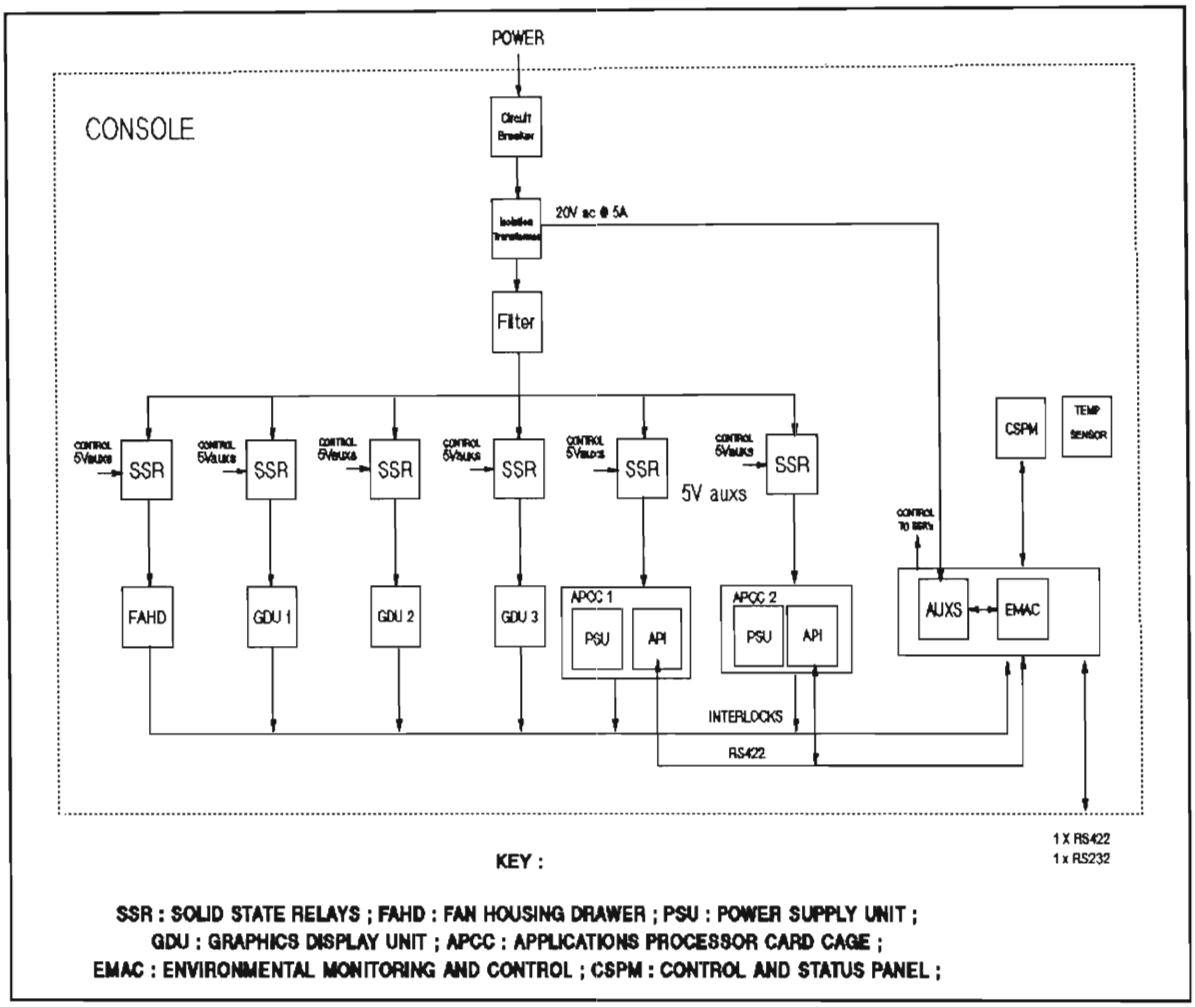

Figure 8 : Power routing and environmental monitoring and control

A softkey module was adjacent to each graphics display unit. Each keypad, together with the keyboard, had associated electronic modules which transmitted the input to an applications processor interface card. This card then catered for the communication of the input to the main applications processor. Each key featured tactile feedback and could be configured to enable automatic stuck key detection or automatic key repetition. Individual keys or complete modules could be masked or unmasked by the applications software. Each module catered for eight user-defined inputs, and, in addition, the alphanumeric keyboard featured sixteen function keys.

The rollerball module communicated with the processor via the serial port on the first graphics interface card. In addition to the rollerball itself, this module contained three user-defined buttons.

Output from the console to the operator consisted of three graphics display units and an array of status and warning indicator lights. Each screen was driven by an associated graphics interface module located in the main applications processor card cage. All of the status and warning lights were controlled by the environmental monitoring and control card, with the exception of the processor fault indicator. This light was driven directly by BITE circuitry resident on the bus terminator unit.

4.3 THE CARD CAGES

All three card cages were mounted on fan housing units. When the fans were operational, the forced air cooling removed heat from the components on the boards and provided for adequate air flow that was uniformly distributed over the surface of each board.

The first card cage housed the heart of the real-time system. It could be withdrawn from the console on slides to allow the contents of the card cage to be viewed. All power and signal connections passed through an input/output connector panel mounted on the rear of the card cage. These could be unplugged, allowing the complete drawer to be removed from the console as a unit. A power supply on this card cage converted the mains to the required values (i.e. +5V, +12V and -12V).

The hardware and software contents that resided in this card cage form the major portion of the discussion to follow. The card cage layout is depicted in Figure 9.

The cards that resided in this card cage were:

a) Two CPU cards
b) Two EPROM/RAM cards
c) A System Data Bus Controller card
d) A Bus Terminator Unit
e) A Dynamic RAM card
f) An Applications Processor Interface card
g) A Serial Communications to Multibus card
h) Three Graphics Interface Modules
i) A Mass Storage Unit card

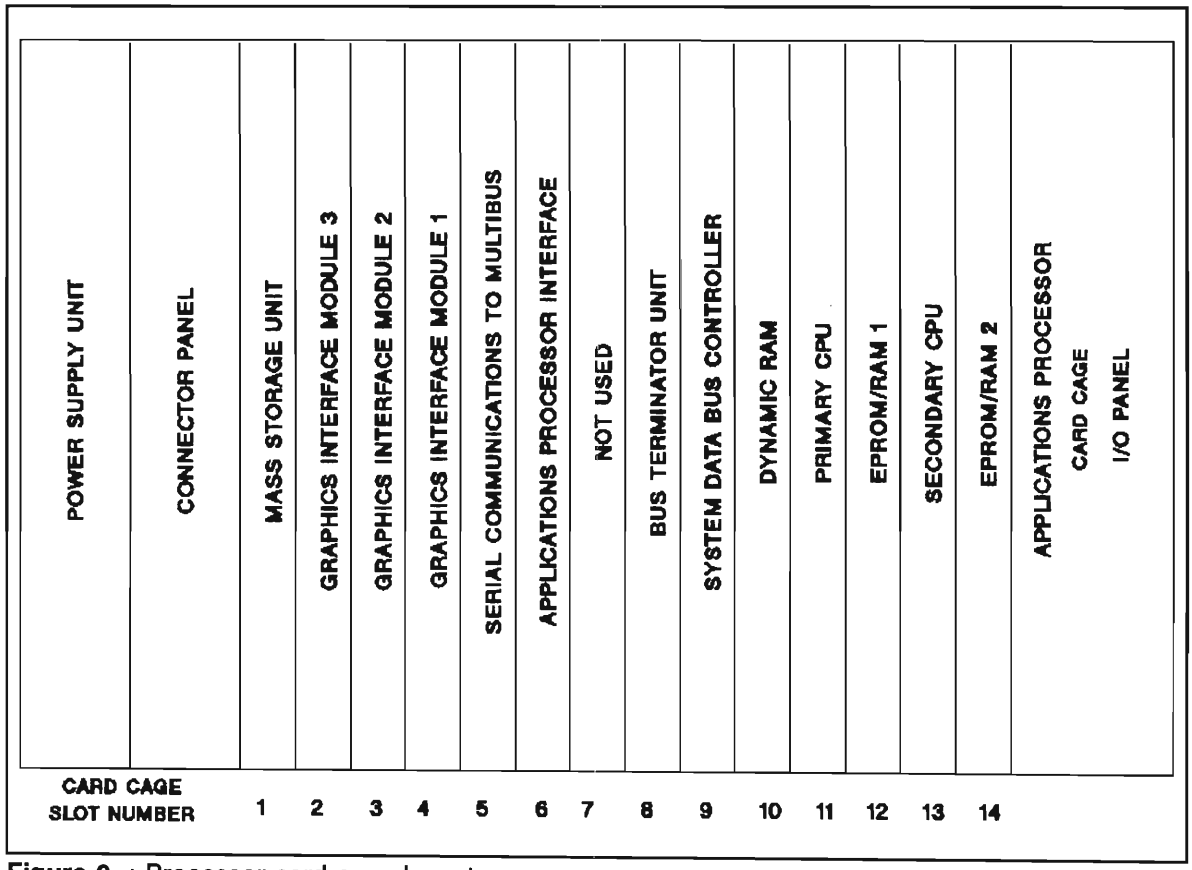

Figure 9 : Processor card cage layout

The environmental monitoring and control circuitry was on a single card located in the second card cage. Besides providing the monitoring and control functions of the internal environment of the console, this card also drove the backlighting, status and warning indication lights on the front panel. The temperature and humidity were monitored and the speed of the cooling fans was modulated accordingly. Limit violations resulted in the generation of warning signals and ultimately the protective shutting down of the console.

With the exception of the rollerball module, the environmental monitoring and control card also implemented the status data generation of all the console manual input devices. Communication from the environmental monitoring and control card to the main applications processor was achieved via a serial link to the applications processor interface card resident in the first card cage.

A 360 Kbyte floppy disk drive was mounted in the third card cage of the console. This allowed for data storage, retrieval and transportability. The disk drive was controlled by the mass storage unit card resident in the first card cage of the console.

4.4 THE MAIN APPLICATIONS PROCESSOR CARD CAGE

The discussion to follow concentrates on the system hardware resident in the processor card cage of the console. The system memory map, the system I/O map and the interrupt structure pertinent to the processor card cage are provided in appendix A.

4.4.1 Main system bus and termination

The interconnection diagram for the processor card cage is presented in Figure 10. The processor card cage incorporated Intel Corporation's Multibus I general purpose system bus structure that enables system cards to interact with each other. This system bus can directly access up to 16 megabytes of memory. It is an asynchronous, multiprocessing system bus designed to perform 8-bit and 16-bit transfers between single board computers, memory and I/O expansion boards. Its interface structure consists of address and inhibit lines, bi-directional data lines, control lines, interrupt lines, and bus exchange lines [INTEL, 1983]. The Bus Terminator Unit terminated all the appropriate system bus lines via pull-up resistors.

The Multibus I device interaction is built upon the master-slave concept and is capable of supporting a multiple master environment via bus exchange logic. Any device which has the ability to control the bus, is a potential bus master. Any module not capable of controlling the bus, but that responds to commands from bus masters are defined to be bus slave devices.

The processor card cage contained intelligent slaves that were potential bus masters. The Multibus structure catered for mutual exclusion by allowing up to eight bus requests to be made simultaneously, but only one bus master to be granted the bus at any one time. Once the bus had been granted to a bus master, it had exclusive control until such time as it released the bus. A slot-priority scheme to resolve bus master contention allowed the system to perform parallel bus arbitration in the form of a priority resolution circuit. This parallel resolution was achieved via an eight to three line priority encoder followed by a three to eight line demultiplexer. All inputs were pulled high via a resistor array. The bus request defaulted to the lowest priority when there were no bus requests [BAUDIN, 1990a].

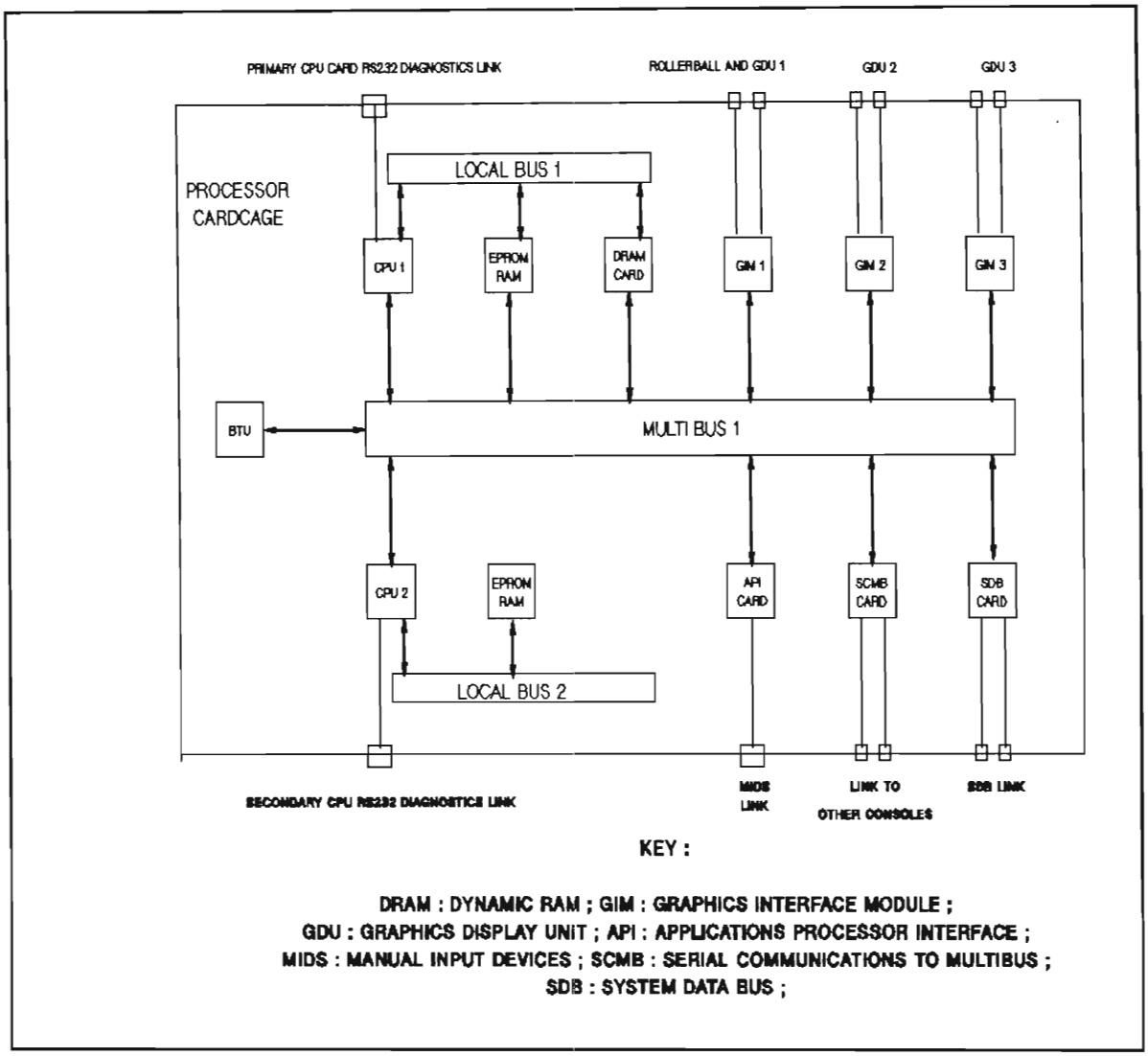

Figure 10 : Processor card cage interconnection diagram

Multibus I data transfers are performed with the usual handshake technique:

(a) the address is placed on the address bus (together with the data in the case of a write)

(b) a read/write command is generated

(c) the slave responds by accepting the data (write) or placing the data on the data lines (read)

(d) the slave generates a transfer acknowledge allowing the completion of the cycle

Interrupts are requested by activating one of eight interrupt request lines. The interrupt acknowledge signal is generated by the bus master when an interrupt request has been received.

4.4.2 Local bus extension

The processor card cage also incorporated the Intel local bus extension interface. This specialised, high-speed interface allows memory expansion using off-board memory that appears to be on-board by linking local bus memory boards directly to the processor board via a local bus cable and connectors. Since the subsystem operated in a dual processor environment (there were two identical CPU cards present in the subsystem), two independent local bus extensions were utilised, one for each CPU card.

The local bus extension interface uses an asynchronous operation protocol for both a read or a write to a memory location or I/O port. Specified signal level interactions must occur during an operation for the operation to proceed.

4.4.3 The CPU card

A functional block diagram of the central processing unit board is provided in Figure 11. The main functional features are described below [GREYLING, 1989a].

The card interfaced to the Multibus and local bus extension via P1 and P2 connectors respectively. The power requirements for the card were catered for via the Multibus interface.

4.4.3.1 The microprocessor

The microprocessor on the CPU card was an Intel iAPX 286 (80286) device [INTEL, 1985a; INTEL, 1987a; INTEL, 1989, p. 3:1-3:55]. This chip may be operated in one of two modes, viz. real address mode (hereafter referred to as real mode) or protected virtual address mode (also referred to as protected mode). In real mode the 80286 can access up to one megabyte of physical memory. In this mode the processor may be regarded as an enhanced 8086 or 80186 microprocessor.

Protected mode allows access of sixteen megabytes of physical memory and one gigabyte of virtual memory per task. This mode also provides a four-level privilege system, memory management and protection features, automatic task switching, and the extended instruction set that controls them.

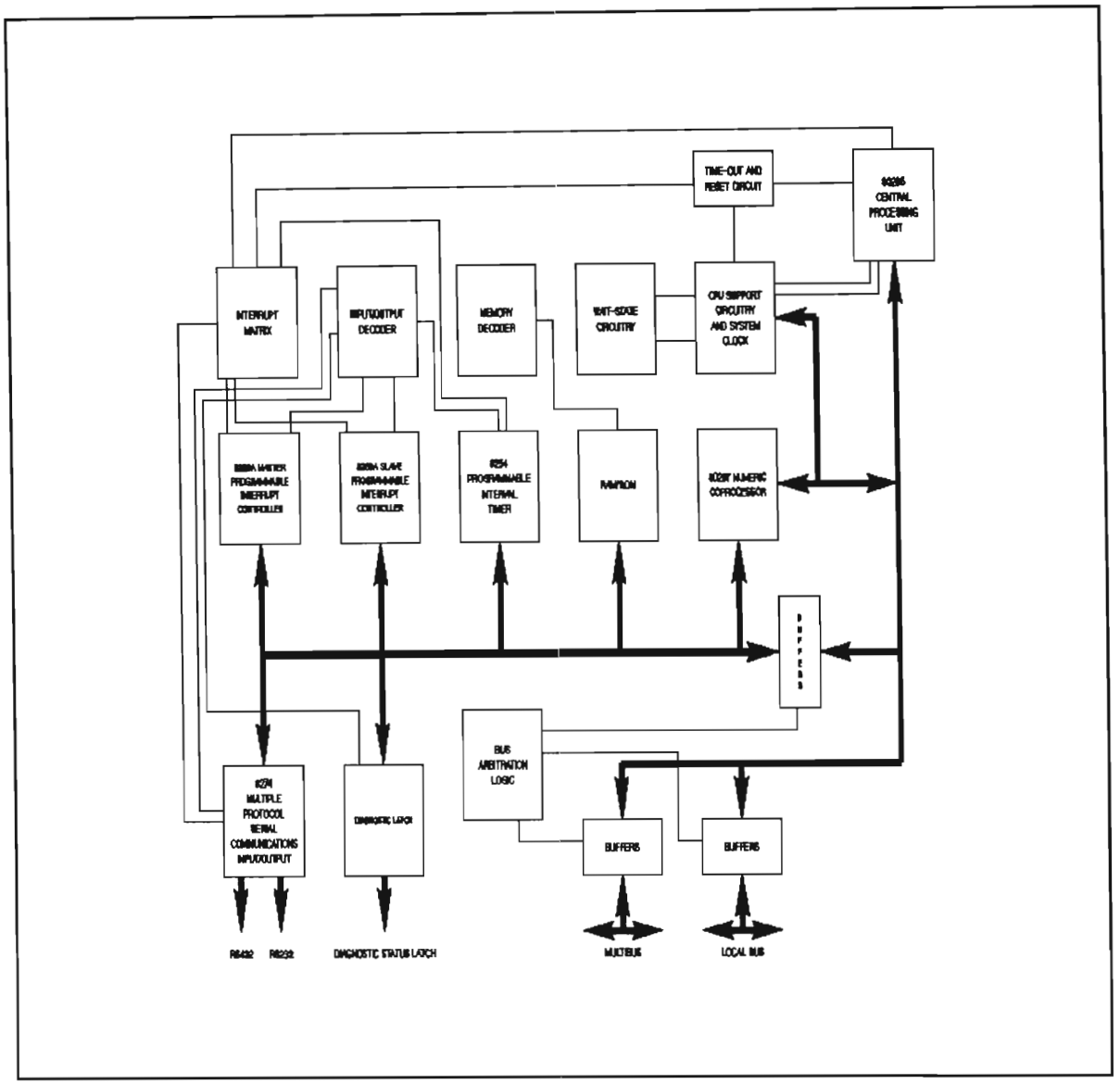

Figure 11 : CPU board block diagram

Memory and I/O is accessible as bytes or words. Words consist of any two consecutive bytes addressed with the least significant byte stored in the lowest address. The I/O address space contains 64K addresses in both modes. Byte wide peripheral devices may be attached to either the upper (accessed with odd I/O addresses) or lower byte (accessed with even I/O addresses) of the data bus.

On power-up or when reset, the 80286 automatically begins operation in real mode. In real mode, the chip select generation programmable array logic also ignored the upper four address lines generated by the 80286. This allowed the on-board EPROM to reside at an address range covering the desired 80286 reset address.

It was possible to switch to protected mode by issuing one software command to change the status of the machine status word, together with a word write to an output port. This word write set the protected mode latch that enabled the memory decoding programmable array logic to take into account the upper four address lines. Full virtual address operation was then allowed, and the on-board EPROMS could consequently be mapped into the upper code segment area of the sixteen megabyte address space.

The 80286 contains a pre-fetch queue of two instructions and six bytes that allows the CPU to "look ahead" at the next instructions. The hardware within the 80286 automatically fills in the queue during the time that the local bus is idle. This factor thus increased the performance of the board.

A pipelined timing technique is used within the 80286 to enhance processing speed by allowing higher local bus bandwidth and thus condensing local bus cycles. Pipelined timing allows a new bus operation to be initiated every two processor cycles, while allowing each individual bus operation to last for three processor cycles. The valid address of the next bus cycle overlaps with the valid data of the current bus cycle by a minimum of one system clock cycle, thereby making use of both the local data bus and the processor address bus during the same processor clock phase.

The base architecture of the 80286 has fifteen registers, grouped into the following four categories:

a) Eight 16-bit general registers are used interchangeably to contain the operands of arithmetic and logical operations. Four of these registers (AX, BX, CX and DX) may be used in their entirety or individually as two 8-bit registers each. The other four registers (BP, SP, SI and DI) are mainly used for specific purposes. BP is used to point to the base of a structure, while SP is the stack pointer. SI and DI are often used as source and destination indices in data transfers.

b) Programs usually consist of different code modules and data segments. However, at any given instant during run-time, only a small subset of these modules and segments are actually in use. The subset normally consists of a single code and data segment and possibly the stack. The 80286 architecture takes advantage of this by providing mechanisms to support direct access to this working subset of the execution environment. Four 16-bit special purpose registers determine the segments that are immediately accessible to an executing 80286 at any given instant. These segment selectors (CS, SS, DS and ES) are used to identify the code segment, the stack segment, the data segment and any extra segment that may be required.

c) Three 16-bit special-purpose status and control registers are used to record and alter certain aspects of the 80286 processor state. The first is the instruction pointer that contains the offset address relative to the current code segment of the next sequential instruction to be executed. Hence, a combination of the code selector and instruction pointer (CS:IP pair) form a 32-bit program counter. The second special-purpose register is the flags register that records specific characteristics of the result of logical and arithmetic instructions and that also controls the operation of the 80286 within a given operating mode. Finally, the machine status word records when a task switch takes place and controls the operating mode of the 80286. The setting of the least significant bit of this register places the 80286 into protected mode. The other three control bits control the processor extension interface.

4.4.3.2 Clock generation and reset circuitry

The 80286 microprocessor was reset and controlled by an 82284 clock generator device. The processor used a double frequency system clock to control bus timing. All signals on the local bus were measured relative to the system clock input. A 12 MHz crystal oscillator module drove the 82284 clock generator device. The processor divided the system clock by two to produce the internal clock speed of 6 MHz [INTEL, 1985a].

The 82284 contained a reset input, which could be generated from an on-board or an off-board source. The on-board reset was achieved by means of an RC circuit feeding a Schmitt NAND gate [GREYLING, 1989a]. This signal enabled a tri-state buffer which pulled the reset line active until the RC circuit had fully charged. The buffer was then disabled.

The external reset was generated by the reset line supplied on the Multibus. This signal line was capable of being driven by any open-collector / tri-state external device that was connected to it.

On receiving a reset signal, the 82284 transmitted the appropriate reset signal to the 80286. It was possible for the reset signal to the microprocessor to be an asynchronous or synchronous ready control.

4.4.3.3 Numeric coprocessor

To facilitate for the processing of high-speed floating point mathematics, an 80287 Numeric Processor Extension chip that ran at a clock speed of 6 MHz resided on the CPU card [INTEL, 1985a; INTEL, 1987a; INTEL, 1989, p. 3:56-3:81, GREYLING, 1989a]. All real mathematics tasks are performed by this coprocessor, which executes its task concurrently to the processing of the 80286. Synchronisation is achieved by means of semaphores, i.e. the 80286 executes a wait instruction to wait for completion of the 80287 operation.

The 80287 instruction set includes a variable length instruction format (including double operand instructions), 8-bit and 16-bit signed and unsigned arithmetic operators for binary, BCD and unpacked ASCII data together with iterative word and byte string manipulation functions.

4.4.3.4 On-board memory

The board had a capacity for 128k of EPROM and 64k of RAM. A registered output programmable array logic chip was used to create the enable signal for all on-board memory. Although the 80286 reset address is FFFFF0 hexadecimal, the programmable array logic device could be configured to ignore the upper four address bits in real mode. This was accomplished by including the output from the protected mode latch into the logic equation of the programmable array logic chip.

4.4.3.5 Serial I/O interface

The main applications processor of the console had a RS232 port connected to the front panel socket mounted on the console [METHA, 1989a; LAW-BROWN, 1990a]. This allowed for diagnostics to be executed without the need to open the card cage drawer housing the processor.

The serial I/O interface on the CPU card was an 8274 Multiple Protocol Serial Controller (MPSC) device [GREYLING, 1989a; INTEL, 1982, p. 9:216-9:250, INTEL, 1985a; INTEL, 1988, p. 2:112-2:149 and p. 2:345-2:382]. Channel A of the 8274 MPSC operated the serial interface for RS422 communications and channel B was used for RS232 communications. The RS232 diagnostic link on the front panel of the console was driven directly by this device. Since the two channels operated independently, simultaneous operation was permissable.

The 8274 MPSC interface to the 80286 microprocessor could basically be configured in two modes, namely polled mode or interrupt driven mode. The polled operation repetitively read the status of the MPSC and based its decision on that status, whereas the interrupt driven operation was accomplished via an external interrupt controller. When the MPSC required service, it sent an interrupt request signal to the microprocessor, which responded with an interrupt acknowledge signal. The MPSC then placed the interrupt vector onto the local data bus.

4.4.3.6 Timers

The CPU card contained an 8254 Programmable Interval Timer (PIT) [GREYLING, 1989a; INTEL, 1982, p. 9:318-9:332], containing three independent 16-bit programmable counters (counter 0, counter 1 and counter 2). The flexibility of the 8254 PIT allows the counters to operate in any one of five modes, namely:

(a) Mode 1: Hardware retriggerable one-shot
(b) Mode 2: Rate generator
(c) Mode 3: Square wave mode
(d) Mode 4: Software triggered strobe
(e) Mode 5: Hardware triggered strobe

The CPU card offered strapping options in the configuration of the counters, and the standardisation of the card required that these options, together with the initialisation of the counters be defined. For this reason, counters 0 and 1 were defined to act as rate generators to be strapped to the interrupt controllers, while counter 3 was configured in square wave mode and linked to the 8274 MPSC device.

In the rate generator mode (counter 0 and counter 1), the counters act like divide-by-N counters, typically used to generate real-time clock interrupts. The output is initially high and when the initial count is decremented to 1, the output goes low for one clock pulse. The output then goes high again and the counter reloads the initial count and the process is repeated.

The square wave mode (counter 3) is typically used for baud rate generation and behaves similarly to the rate generator mode. The output is initially high and when half the initial count has expired, the output goes low for the rest of the count. The process is periodic and repeated indefinitely.

4.4.3.7 Interrupt controllers

The Programmable Interrupt Controller (PIC) functions as an overall manager in an interrupt-driven system environment. It accepts requests from the peripheral equipment, determines which of the incoming requests is of the highest priority, ascertains whether the incoming request has a higher priority value than the level currently being serviced, and issues an interrupt to the CPU based on this determination.

The 8259A Programmable Interrupt Controller is one such device specifically designed for use in real-time, interrupt-driven microcomputer systems [INTEL, 1982, p. 7:120-7:137; INTEL, 1989, p. 2:259-2:282]. A selection of priority modes is available to the programmer so that the manner in which requests are processed by the 8259A can be configured to match the system requirements. The priority modes can be changed or reconfigured dynamically at any time during the applications program. This means that the complete interrupt structure can be defined as required, based on the total system environment.

The interrupt sequencing occurs as follows for the 8259A:

(a) An interrupt request is raised and passed to the 8259A.
(b) If appropriate, the 8259A sends an interrupt request to the microprocessor.
(c) The microprocessor acknowledges the interrupt.
(d) The 8259A sets the in-service register of the appropriate interrupt.
(e) The microprocessor sends a second interrupt acknowledge, and the 8259A releases the interrupt vector onto the data bus.
(f) The in-service register is either reset for an 8259A configured as "automatic end-of-interrupt", or else the 8259A waits for an end-of-interrupt command to be issued.

The 8259A PIC manages eight requests and has bit-slice logic built-in features to enable the cascading of other 8259A's. The CPU card had two 8259A's wired in a master/slave configuration. The master controlled the slave through the three line cascade bus. The cascade bus acted like a chip select signal to the slave during the interrupt acknowledge sequence. In this configuration, the slave interrupt output was connected to the lowest priority master interrupt request input. When the slave request line was activated and later acknowledged, the master would enable the slave to release the interrupt vector during the interrupt acknowledge sequence.

4.4.3.8 Diagnostic status latch

The CPU card contained an 8-bit latch (type SN74373) that could be used as both a general purpose output as well as a diagnostic status output latch [GREYLING, 1989a]. The latch was wired to the local bus P2 connector, and this, in turn, was used to drive two seven segment hexadecimal displays on the bus terminator unit card. The input lines on the bus terminator unit were pulled high by a resistor pack, implying that (a) it was possible to enable the required display via the card cage backplane and (b) an enabled display with floating input defaulted to FF hexadecimal [BAUDIN, 1990a].

The status latch was used extensively during the execution of built-in tests. Provided that the circuitry used to drive the latch was operational, the operator could view the test currently executing. If a critical test failed and the MPSC test had also failed (implying that diagnostic reporting to the RS232 was not possible), then the operator could view the code on the seven segment displays in order to determine what had caused the failure.

A normal operational state without any test failures was indicated by 00 hexadecimal on the diagnostic status latch. In addition to being wired to the displays, the 8-bit output from the latch was also logically ORed and routed to a fault indicator LED on the status and control panel of the console. Thus a non-zero value on the displays caused the fault indicator to flash.

4.4.3.9 Time-out circuitry

The Multibus I transfer acknowledge signal terminates bus operations on the Multibus by driving the 82288 bus controller. However, should an attempt be made to access a non-existing device or memory on the Multibus, the transfer acknowledge may never be generated. This would cause the processor to "hang up", tie up the Multibus and prevent its use by other potential bus masters. The CPU card overcame this problem by implementing a bus grant time-out circuit, using one-shot monostables [GREYLING, 1989a]. If the transfer acknowledge was not activated within a finite period, a time-out occurred, generating a transfer acknowledge and thus releasing the bus. The time-out signal was simultaneously used to generate an interrupt informing the microprocessor that an invalid operation had occurred.

4.4.4 The EPROM/RAM card

The EPROM/RAM card was an extension of memory local to each CPU card. It operated inside the standard double Eurocard chassis, interfacing both to the local bus and to the Multibus [GREYLING, 1989b]. The memory was dual ported and a specific location on the local bus corresponded to the same location on the Multibus. This sharing of resources implies that when the CPU card tried to access memory resident on the EPROM/RAM card, it would initially be attempted via the local bus, and, if a time-out occurred, another memory access would be attempted via the Multibus.

Both the local bus and Multibus interface were capable of decoding one Megabyte of memory within a sixteen Megabyte range. A 2K word or 4K byte boundary decoding was featured. Each had a separate base address decoding programmable array logic chip to accommodate a wide variety of applications. The Multibus and local bus support both 8-bit and 16-bit data transfers. In the case of 8-bit transfers on Multibus, the lower eight data bits were used exclusively and the top eight data bits were ignored. The local bus, however, transferred the low byte on the lower eight data bits and the high byte on the upper eight data bits for byte transfers.

The bus operations were independent of each other, allowing memory cycles from the busses to overlap. Each bus had the capability for exclusive utilisation of the memory. This was particularly useful for a program which required that a certain block of shared memory should remain unmodified during the time that it was being accessed. When interfacing to the 80286 CPU card, bus access time-outs occurred after ten milliseconds. If a bus was attempting to access memory with the other bus asserting the bus lock, the resulting time-out could have introduced errors at the system level. A feature of the EPROM/RAM card was a dead-lock protection which prevented both the local bus and the Multibus from locking the memory simultaneously. If this was attempted, then the local bus got priority and the Multibus had to wait. Although this prevented a deadlock from occurring, the disadvantage was that the Multibus could not be assured a bus lock. This disadvantage was, however, outweighed by the increase in system reliability.

Memory selection was accomplished by the memory bank select programmable array logic device, which was factory programmed during assembly for a specific addressing requirement and type of memory device selected. The capacity of the module depended on the size of the memory used, and varied from 32 Kbytes to 1 Mbyte, using 2K X 8 or 64K X 8 devices, respectively. The standardised

code offered jumper options for subsystem users to configure the card in one of three different ways. For the applications under consideration, the addressing was 384 Kbytes of RAM and 256 Kbytes of ROM (addresses 40000h to 9FFFFh and A0000h to DFFFFh, respectively).

4.4.5 The System Data Bus Controller card

The system data bus controller card catered for communication between the subsystem and the system data bus. It was driven by an 80186 microprocessor and contained 16 Kbytes of on-board ROM and 64 Kbytes of on-board RAM [GREYLING, 1989d]. Of this on-board RAM, 62 Kbytes were accessible to both the controller and the host CPU card and were used to contain the data structures that were shared between the controller and the host. Only word transfers to or from the controller were allowed. The board was based at address 20000h (i.e. 20800h to 2FFFFh was visible to the Multibus).

Control commands from the host CPU card to the controller card were sent via an I/O port. A write to this location caused an interrupt on-board the controller card. Status reports were sent back to the host via locations on the dual ported RAM. The controller card also interrupted the host under certain conditions via one of the eight Multibus interrupt lines.

Approximately 8 Kbytes of the 64 Kbytes of the dual port RAM contained message buffers, self-test results, the system configuration pointer, the system control block, the subaddress control blocks and the monitor control block. The rest of the memory was used by the host. With the exception of the self-test results, all of these data structures deal with the interface to the system data bus. They are thus beyond the scope of this study and shall not be considered any further here. The self-test results are, however, of particular significance, and are described below.

When the system data bus controller card was powered up, or the system was reset, the processor on the controller card executed a self-test procedure which tested :

(a) the 80186 CPU
(b) the on-board EPROM
(c) the reserved 2 Kbytes of on-board RAM
(d) the three integral timers on the card
(e) the integral interrupt controller in the CPU

The results of these self tests were reported via dual port RAM. If the CPU, EPROM or stack test failed, the board halted. The self tests took approximately fifty milliseconds to execute. It was possible to execute several self tests on request by the host at any time after the board had completed its POST, including :

(a) an EPROM checksum test
(b) the reserved RAM test
(c) the interrupt controller test
(d) an I/O recognition test
(e) a shared RAM test

During the execution of these tests, the controller kept track of the addresses of tests (a) and (b). Test (a) was executed 256 times to cover the entire EPROM and test (b) was executed 64 times to cover the reserved RAM area. When test (e) was executed, the host supplied the address range and data in order to avoid possible data corruption.

4.4.6 Bus terminator unit

The bus terminator unit was designed to be located near the centre of a Multibus I card cage. In addition to providing the correct termination for the Multibus, resolving bus priority, and displaying the diagnostic status latches of the two CPU cards in hexadecimal format, the module also:

(a) allowed for both a manual and a software system reset
(b) provided BITE in the form of the Multibus clock failure detection circuitry and a monitoring of overall system fault conditions

It was also possible to issue a software reset to the system from the host CPU card at any time by means of a pre-defined reset code written to the status latch. Each of the status busses on the bus terminator unit were monitored for this reset code by an 8-bit magnitude comparator [BAUDIN, 1990a]. The three possible reset signals (i.e. the manual reset or one of the software resets from either CPU card) were logically ORed together and used as the input to the reset monostable. The monostable was used to prevent the reset code from forcing the reset line permanently low.

On resetting the system, a pulse stretch monostable was activated in order to enable the visibility of the reset LED. The Multibus reset line was also monitored in order to detect a system fault on this line. The reset detection signal was used as an input to the fault detection circuitry.

The Multibus I architecture requires two clocks to be present for successful operation. The BITE on-board the bus terminator unit detected the absence of any of these clocks by buffering the incoming signal and using this buffered clock signal as the input for a retriggerable monostable. The period of the monostable was longer than that of the clock signal being monitored and, therefore, under normal conditions, the monostable remained permanently triggered. The absence of the clock, however, caused the monostable to time-out. The active low output (which then went high) was used as input to the fault detection circuitry and to drive an open collector inverter. This high input to the inverter was used to sink the current from the "clock failure" LED on the bus terminator card.

The fault detection circuitry monitored the system clocks, the reset detection and the status latches of the two CPU cards. As soon as a fault condition was detected, the fault indicator LED on the bus terminator unit and the fault indicator on the status and control panel module started flashing.

4.4.7 Dynamic RAM card

The dynamic random access memory (DRAM) module interfaced to the system via both the local and Multibus connectors, providing 1 Megabyte of dual ported memory [GREYLING, 1989c]. Control on-board the dynamic RAM card was achieved by means of an 8207 dynamic RAM controller chip. The dual port interface allowed the two busses to independently access the memory. Each bus had a separate base address decoding programmable array logic. The base address of the local bus was at 100000h, while the equivalent Multibus location resided at 300000h. The lock function provided each port with the ability to obtain uninterrupted access to a critical region of memory and thereby guaranteed that the other port could not access the same memory prematurely.

The memory module also incorporated error checking and correcting circuitry on-board. The intelligence controlling these fault-tolerant features was by virtue of an 8206 error detection and correction unit. The dynamic RAM could either operate in "correcting" or "uncorrecting" mode. When optimised for quick data access, the 8206 unit was configured in the "uncorrecting" mode, where the delay associated with the error correction circuitry was transparent and a transfer acknowledge was issued as soon as valid data was known to exist. If the error flag was activated, however, then the transfer acknowledge was delayed until after the 8207 dynamic RAM controller chip had instructed the 8206 to correct the data and the corrected data became available on the data bus.

"Uncorrectable" memory errors were indicated by the error correction circuitry to the dynamic RAM controller via an error flag and were latched by an error strobe signal from the dynamic RAM controller to one of the eight Multibus interrupt lines. The error flag could only be reset by the host CPU card.

Error correction or "scrubbing" was performed during the dynamic RAM refresh cycles. Since the 8207 had to refresh the RAM, performing error scrubbing during the refresh allowed it to be accomplished without performance penalties. Upon detection of a correctable error during refresh, the RAM refresh cycle was lengthened slightly to permit the 8206 to correct the error and for the corrected word to be written into memory. Uncorrectable errors detected during scrubbing were ignored. The refresh was not affected by the memory lock function and could hence occur during a locked memory cycle.

The basic function of the on-board error correction circuitry was to detect any memory errors that occurred during read operations. In the "correcting" mode, the board would then correct any single-bit errors. In the "uncorrecting" mode, errors were still detected, but no action was taken to correct any bits.

The 8206 unit used a Hamming single-error correcting code to generate check bits for each bit word associated with it. It was possible to correct all single errors if, and only if, each single error pattern had a different syndrome [BOSE and METZNER, 1986]. The single-error correction could readily be accomplished by a combinational circuit having the syndrome as input and n outputs, such that each of the n different single-error syndromes produced a one at just the output position that corresponded to the single error position.

The modified Hamming code had the advantage that it required the fewest possible check digits for its code lengths. However, the disadvantage was that no room was left for detection of any events of greater than one error in a block. Any such event was interpreted incorrectly as a single-error because each non-zero syndrome was matched with one of the single error events. The addition of a single additional parity digit which was the modula 2 sum of all the other digits improved this situation somewhat by detecting all double error patterns in addition to correcting all single-error patterns.

When a read operation was active, the error correction circuitry generated a new group of check bits already stored in the memory of the error correction circuit. If the two check bits compared, no error had occurred. However, if the board was in the "correcting" mode and a single-bit error occurred, then the error correction circuitry would correct the data word to its original value. The error flag and correctable error lines would be asserted to signal the dynamic RAM controller that a correctable error occurred. Thus, the current read cycle would be extended, and a read-modify-write cycle would be performed with the new corrected data written into memory.

4.4.8 Applications processor interface card

The applications processor interface card allowed data to be passed between the manual input devices and the host CPU and between the environmental monitoring and control card and the host CPU [METHA, 1989b]. The data was buffered by 2 Kbytes of dual ported RAM residing on the applications processor interface card. The location of the dual port RAM was seen by the applications processor interface card to be based at 1000h to 17FFh and from the host CPU as residing at 11000h to 117FFh. The dual port RAM featured two separate I/O ports which allowed independent byte or word accesses for reads or writes to any locations in the dual port RAM.

The dual port RAM was functionally divided into two sections of memory. The host CPU was allowed to read or write to one section, while the applications processor interface card was allowed a read operation only. Alternatively, the applications processor interface card was allowed to read or write to the other section, while the host was only allowed a read.

The applications processor interface was presented with serial data from the manual input device modules and the environmental monitoring and control card. This data was routed to specific locations in the dual port RAM dependent on the source of the data. Control to avoid overwriting valid data before the receiving device had read the data, was included in the protocol.

The interface allowed for control information and test requests to be passed from the host CPU to the manual input device modules or the environmental monitoring and control card via other locations in the dual port RAM. Status information and test results could also be passed by the applications processor interface to the host processor.

Two locations on the dual port RAM of the applications processor interface card were used as general purpose locations. A write to one of these locations caused the host to interrupt the applications processor interface card and, similarly, a write to the other location caused the applications processor interface card to interrupt the host CPU card.

Data written from the host to the applications processor interface card included such commands as reset codes, self test requests, alarm enabling, masking or unmasking of manual input devices, enabling or disabling of stuck key detection facilities and environmental status requests. On the other hand, a transfer initiated by the applications processor interface card included reset acknowledges, key presses and statuses and abnormal operating environmental conditions.

The applications processor interface card POST wrote a specific bit pattern to all the locations in the dual port RAM. The pattern was zero for the base address and incremented by one for each location thereafter. It was written in byte format, and thus a zero followed a value of FF hexadecimal. The values were then read, and a checksum was calculated based on these values. This checksum was then written to the pre-defined general purpose location, causing an interrupt to the host via one of the Multibus interrupt lines. The applications processor interface card then waited for an acknowledgement from the host. If no acknowledgement was received within a certain time, the board timed-out, otherwise the acknowledgement was accepted and the handshaking procedure completed.

Following a successful power up, the applications processor interface card was ready to perform its applications task. The communication protocol was as follows:

(i) For a host-initiated operation:

(a) The host wrote to specific locations in the dual port RAM relating to the function required by the manual input devices or environmental monitoring and control.
(b) The host wrote to the general purpose location on the applications processor interface card, which resulted in an interrupt being generated to the card.
(c) The applications processor interface card read the location and performed the required function based on this command.
(d) On completion of the read operation, an applications processor interface read acknowledge was written to the other general purpose location, causing an interrupt to the host.

(ii) For an applications processor interface card initiated operation:

(a) The applications processor interface wrote to specific locations in the dual port RAM relating to the function performed by the manual input devices or status information pertaining to the environmental monitoring and control.
(b) The applications processor interface wrote to the second general purpose register in the dual port RAM, causing an interrupt to be generated to the host.
(c) The host serviced the interrupt and, if appropriate, sent an additional acknowledgement to the applications processor interface card, depending on the type of operation that was performed.

4.4.9 Serial communications to Multibus card

The serial communications to Multibus board was an intelligent communications controller designed for autonomous execution of complex communication protocols [SIEMENS, 1988, p. 165-180]. The board controlled the communications between the three consoles of the subsystem via the local area network by using the high-level data link control (HDLC) standard protocol [TANENBAUM, 1981, p. 167-172].

Intelligence on-board the serial communications to Multibus board included an 80186 microprocessor, 128 Kbytes of EPROM, 128 Kbytes of dynamic RAM, an 8259A programmable interrupt controller and an 82258 DMA controller. Four DMA channels were responsible for the serial interfaces to the other two consoles.

Four I/O ports on the card controlled the communications from the host CPU to the serial communications to Multibus card. Three of these I/O ports were defined as "write only", i.e. the general purpose register, the reset register and the interrupt reset register. The fourth port was the status register, which was "read only".

Besides having the ability to communicate with the other two consoles, the firmware on-board the serial communications to Multibus controller card had certain testing capabilities [BAUDIN, 1990b]. At power-up, self tests were performed and the board detected whether the host CPU had failed to communicate over the Multibus.

All communications between the host CPU and the serial communications to Multibus card were initiated by the host. The host wrote a command into the general purpose register which caused an interrupt to be generated on-board the card. The card serviced this interrupt by reading the command and then performed the required task. Self tests could also be requested and reported at any time during normal applications.

On completion of a requested task, the data was written to global RAM and, if applicable, an interrupt was generated to the host CPU, depending on the initial command received from the host. If the primary CPU card was executing as the main applications processor, then the task results were reported to global RAM locations on the EPROM/RAM card. However, if the secondary CPU card had been switched in and had taken over as the main applications processor, then the test results were reported to global RAM locations on the dynamic RAM card.

4.4.10 Graphics interface modules

The graphics interface modules were intelligent slave boards used to control the graphics display units by graphics kernel system (GKS) firmware. The intelligence on-board the graphics interface modules consisted of an 8088 microprocessor, an 8087 numeric coprocessor, 128 Kbytes of EPROM, 64 Kbytes of RAM, an 8259A Programmable Interrupt Controller and an 82720 graphics controller chip [SIEMENS, 1988, p. 263-274].

The I/O interface was the same as that of the serial communications to Multibus card, i.e. four I/O ports on the graphics interface modules controlled the communications from the host CPU, viz. the general purpose register, the reset register, the interrupt reset register and the status register. One of the Multibus interrupt lines was dedicated to handle interrupts from the graphics interface modules to the host.

The communications protocol was also the same as the serial communications to Multibus card, i.e. the host CPU wrote a command into the general purpose register which caused an interrupt to be generated on-board the graphics interface module. The graphics interface module serviced this interrupt by reading the command and then performed the required task. On completion of the requested task, an interrupt to the host was generated, if appropriate, depending on the nature of the request made by the host [BAUDIN, 1990c].

The firmware on-board the graphics interface modules was designed to perform on-board POST or off-line diagnostics, accept rollerball positional information fed directly into the RS232 channel of one of the modules, as well as to drive the graphics displays via calls to the graphics kernel system routines. Driving the graphics displays included tasks such as error and system status reporting, generating test patterns and various other graphics applications displays. All the data structures necessary for proper communication to the graphics interface modules were passed through global RAM either on the EPROM/RAM card or on the dynamic RAM card, depending on which CPU card was executing as the main applications processor.

In order to enhance the fault detection capabilities of the system, the intelligent graphics interface modules also had the ability to detect either a Multibus failure or the failure of the main applications processor to communicate over the Multibus and to consequently report these findings to the respective display. In addition, each graphics interface module was configured dynamically at run-time for its specific application by the host firmware. Once the graphics interface module had completed its on-board POST, it expected a command from the host that informed it which module it was to become. Triple redundancy was thus introduced at this level and the system only considered a failure of all three graphics interface modules to be critical.

4.4.11 Mass storage controller card

The mass storage controller card provided the interface and software required to support the control of mass storage devices. The main functional features consisted of an 80186 microprocessor, 128 Kbytes of EPROM, 1 Mbyte of on-board RAM, a multi-protocol controller chip for controlling two independent serial I/O channels, a Centronics interface and a NCR5380 small computer systems interface (SCSI) bus device [SIEMENS, 1988 p. 211-228]. The small computer systems bus interface is a parallel, multimaster I/O bus that provides a standard interface between computers and peripheral devices [ANSI, 1986; GLASS, 1990a; GLASS, 1990b]. Three programmable interval timers and a programmable interrupt controller are built into the 80186 microprocessor.

Commands were issued from the host CPU to the mass storage controller card via I/O ports similarly to the method used for both the serial communications to Multibus card and graphics interface modules. The intelligent slave occupied four addresses in the 64 Kbyte Multibus I/O address range allowing for a general purpose register, a reset register, an interrupt reset register and a status register. The mass storage controller card intelligent slave was unable to initiate operations on other boards in the system.

Three 64 Kbytes of RAM allowed for the transferring of data and commands between the mass storage controller card and the rest of the system. Two of these 64 Kbyte areas were provided by the software on the card as dynamically relocatable windows to view the global RAM on the Multibus. The other area was dual ported and thus occupied 64 Kbytes of the Multibus address space.

In addition to providing the interface to the usual disk operating system command types, drive and disk configurations, status parameters and peripheral control parameters, the mass storage controller card also had the ability to perform various on board tests and to pass the results to the Multibus interface [POLMANS, 1990a].

The method of passing data and parameters was to set up the parameters of a command in a parameter block in memory that was accessible to the mass storage controller card. This memory area could either have been in the dual port RAM area of the card or on another card that the mass storage controller card could access from the Multibus. The address of the parameter block was then written to the parameter block address register (i.e. a pre-defined location) and the command to be executed was written to the general purpose register. The mass storage controller card received the command (which caused an interrupt), retrieved the pointer to the parameter block and executed the command.

On completion of the task, the mass storage controller card set the address of the parameter block in the response block address register (another pre-defined location), set the status byte in the command block to "ready", and interrupted the host. In the event of an error, the status/error number was set to indicate the error number and a pointer pointed to the appropriate error message. Several mass storage tasks could execute concurrently, and, therefore, on receiving the interrupt, the host read the response block address register to identify the source of the interrupt (the response block address register contained the contents of the parameter block address register that initiated the operation). The host then reset the slave interrupt and processed the contents of the parameter block [POLMANS, 1990b].

4.5 SUMMARY

The system architecture of a real-time embedded system was presented in this chapter. A hierarchical approach was taken to achieve this, by providing a general overview of the division of the system into subsystems and the separation of one of the subsystems into three independent consoles. The interface between the subsystem and the system data bus and the interaction of the consoles with each other was briefly discussed.

The man-machine interface input/output, power routing and environmental monitoring and control of one of the consoles was then described. This was followed by an identification of the card cage that contained the majority of the processing power driving the console. A detailed decomposition of the functional blocks resident on the two CPU cards was given, together with a brief description of the rest of boards resident in the card cage. This included the testing capabilities of the various intelligent slaves and their respective interface protocols.

5 SOFTWARE DEVELOPMENT

5.1 INTRODUCTION

The introduction to the system architecture has been provided in order to present the process of software and hardware integration. In the implementation of the software, an attempt was made to adhere to the theories and philosophies presented earlier. All software source files used to implement the built-in tests were written using the PL/M-86, PL/M-286, ASM-86 or ASM-286 traditional programming languages [INTEL, 1985b; INTEL, 1985c; INTEL, 1986b; INTEL, 1987a]. The approach adopted in the development of the software was to understand and apply the object-oriented design techniques to these traditional programming languages.

This chapter relates to the designing of the standardised code at system level and the development of subsystem-specific test software at subsystem level. The overall software flowchart is presented in Figure 12. The standardisation efforts and a more detailed description of this flowchart are presented in the ensuing sections.

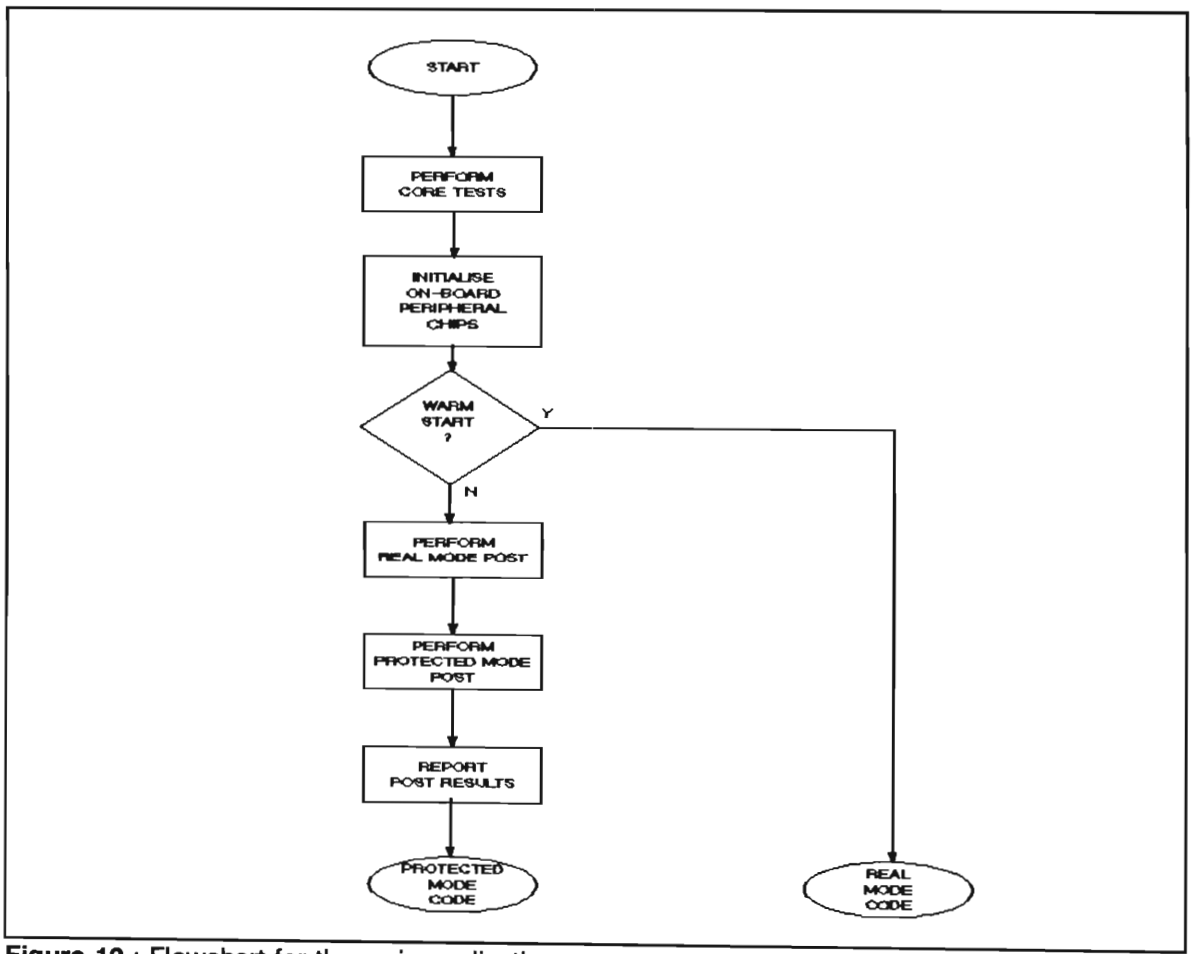

Figure 12 : Flowchart for the main applications process

5.2 SYSTEM LEVEL STANDARDISATION

As was alluded to earlier, a standard computing segment consisting of the CPU card, EPROM/RAM card and system data bus controller card was defined within the system. The standardised code was written and the strapping fields on all three boards was also standardised. The EPROM/RAM card was identified as being common subsystem hardware containing unique subsystem-specific applications code. The standardised code resident on each CPU card of all *n* subsystems placed a restriction on an address range at the top of the EPROM/RAM card that was defined to be the system description. This system description is provided in Table X and an interpretation of each value by the standardised code is presented in Table XI.

Table X : Firmware resident system description

Address	Value	Description
DFF00h	5Ah	EPROM identification
DFF01h	Subsystem dependent	Subsystem Number
DFF02h	Subsystem dependent	Subsystem Version
DFF03h	Subsystem dependent	EPROM/RAM option
DFF05h	Subsystem dependent	Number of processors
DFF06h	Subsystem dependent	Operating mode
DFF07h	Subsystem dependent	Copy descriptor tables
DFF08/9h	Subsystem dependent	IP address of real mode code
DFF0A/Bh	Subsystem dependent	CS address of real mode code
DFFFAh	53h	ASCII for "S"
DFFFBh	55h	ASCII for "U"
DFFFCh	4Dh	ASCII for "M"
DFFFDh	Subsystem dependent	Low checksum byte
DFFFEh	Subsystem dependent	High checksum byte
DFFFFh	Subsystem dependent	Checksums present

Table XI : Standardised code interpretation of the system description

Address	Description	Indication to the standardised code
DFF00h	EPROM identification	The presence of the card in the system.
DFF01h	Subsystem Number	Which applications code was resident on the card.
DFF02h	Subsystem Version	Which version of software was resident on the card.
DFF03h	EPROM/RAM option	The size of the RAM/ROM ratio.
DFF05h	Number of processors	The number of CPU cards present in the subsystem.
DFF06h	Operating mode	Whether the system applications was to execute in real mode or protected mode.
DFF07h	Copy descriptors	Whether the protected mode descriptor tables were to be copied from ROM to RAM.
DFF08h	Real mode address	This double word contained the address of the 80286 real mode system applications code starting address (CS:IP pair).

The standardised code was defined to consist of the following:

(a) Standardised initialisation routines for the standard computing segment.

(b) Standardised POST for the standard computing segment.

(c) Standardised interrupt handlers.

(d) Standardised programmable interrupt controller routines.

(e) Standardised programmable interval timer routines.

(f) Standardised diagnostic status latch routines.

(g) Standardised off-line BIT for the standard computing segment.

(h) Standardised RS232 serial I/O routines.

(i) Standardised ASCII conversion routines.

5.3 SUBSYSTEM LEVEL STANDARDISATION

The two CPU cards, the two EPROM/RAM cards and three graphics interface modules were respectively identified as common hardware within the subsystem and it was attempted to standardise these modules as far as possible. The standardisation effort of each of these modules shall now be discussed.

5.3.1 Standardising the CPU cards

The standardisation of the two CPU cards posed a challenging problem, namely that the standardised code had to satisfy standardisation requirements at both system and subsystem level. This meant that in addition to providing the aforementioned standardised routines, the standardised code had to cater for a dual CPU environment at card cage level[1]. At power up, both CPU cards would be executing identical boot code. In order to avoid conflict during off-board tests, the secondary processor needed to be delayed to allow the primary processor to be the first processor that performed any off-board writes. Implementing a software solution alone implied that the two CPU cards would be non-standard, and, therefore a combination of hardware and software was introduced to solve the problem. Since the secondary processor was slot-dependent, it was possible to provide a backplane solution in conjunction with the firmware. In this manner, any CPU card inserted into the appropriate slot could identify itself as a secondary processor.

The solution implemented was that one of the Multibus interrupt lines was allocated as the definition of a secondary processor. This interrupt line on the respective slot of the card cage was strapped high to provide a permanent interrupt. The standardised code executed a procedure (DELAY_SLAVE_PROCS) that set the 8259A programmable interrupt controller chip to allow level triggered interrupts and unmasked the appropriate interrupt request. If an interrupt was generated on this request line (i.e. the CPU was in a secondary slot), the interrupt service routine set a flag that identified the processor as a secondary processor and caused the processor to delay sufficiently. This ensured that, under normal circumstances, the primary processor would perform the remaining power-on tasks before the secondary processor without the latter causing any interference or data corruption.

---

[1] Each subsystem was limited to a maximum of two CPU cards per card cage.

5.3.2 Standardising the EPROM/RAM cards

Striving for standardisation of the EPROM/RAM cards at subsystem level implied that it was essential that both cards contain identical code, and, therefore were able to execute code to drive both the primary and secondary processors.

In addition to the advantage of only needing a single standardised subsystem EPROM/RAM replacement module, the advantage of standardising the cards was the ability for the secondary processor to execute its relevant tasks without needing the associated local bus EPROM/RAM card. When the EPROM/RAM card on the local bus of the secondary processor was absent, however, the run-time execution of the secondary processor was slower due to the extended length of the opcode fetch cycle.

The drawback to standardising the EPROM/RAM cards was, of course, the extra effort involved in providing redundant applications code on each board. At system level, the incorporation of inter-subsystem redundant code was too complex to coordinate due to the vastly differing applications of the various subsystems. For this reason, the cards were only standardised at subsystem level and not at system level.

5.3.3 Standardising the graphics interface modules

Each graphics interface module was defined to report to a separate area of global RAM. Although the port addresses were defined by link options on the graphics interface modules, the firmware on each graphics card was standardised. Each graphics interface module was thus designed to be dynamically reconfigurable in the sense that a command from the host CPU informed each graphics interface module regarding its respective task. This task was determined at run-time by the host CPU based upon the status of the system.

The firmware on-board the graphics card was so designed that if any command was issued by the host CPU to perform a task that required prior knowledge regarding its run-time configuration, then the graphics card informed the host that it was unable to perform the requested task and ignored the command. Thus a prerequisite was placed upon the host to initially check that the graphics card was present, and, if so, to delegate responsibilities to each graphics interface module. Self test results were then inspected and the system was configured appropriately. The algorithm is shown in Table XII.

Table XII : Dynamic allocation of graphics interface modules

```
TEST_SLAVE

  begin
  Check for presence of slave
  If the card is present then
    begin
    If the card is a graphics card then
      configure the card
    Issue a command to report POST results to global RAM
    Allow enough time for results to be reported
    Copy results to on-board RAM
    end
  end
```

The system had to also consider the possibility of the secondary processor performing the tasks of the primary processor. Due to the architecture of the system, this event required that the global RAM data structures passed between the host CPU card (i.e. the secondary processor in this case) and the graphics interface modules reside on the dynamic RAM card. The EPROM/RAM card on the Multibus could no longer be used since the secondary processor contained its own local EPROM/RAM card located at the same base address. Thus various options arose depending on the run-time status of the system, and, in this fashion, the code was designed to be fault-tolerant at power up.

5.4 DESIGNING THE STANDARDISED CODE

The overall software flowchart was shown in Figure 12 (p. 71). It is appropriate to expand upon this flowchart in order to illustrate the distinction between the standardised code and the subsystem-specific code. The flowchart for the standardised code is presented in Figure 13 and the algorithm for the main initialisation routine of the standardised code is presented in Table XIII.

Table XIII : Main initialisation routine of the standardised code

```
INITIALISE_ALL

  begin
  Test the core system
  Initialise the on-board peripheral chips
  If a "cold start" is detected then
    begin
    Perform the standard computing segment POST
    Analyze the results
    If possible, output the results to the visual display unit
    end
  else
    Perform the real mode subsystem-specific demonstration code
  end
```

Firstly, core system tests of microprocessor, on-board ROM and on-board RAM were executed. If any of these failed, the program immediately halted, otherwise it continued to:

a) initialise the programmable interval timer

b) initialise the programmable interrupt controller

c) initialise the multi protocol serial communications chip

d) initialise the numeric coprocessor

e) check for a "warm start", and, if this was detected, control was transferred to the real mode demonstration code.

f) perform the POST

g) record the results of the POST

h) output the POST results via the RS232 link, if possible

Throughout the duration of the POST, any test defined to be critical caused the processor to halt, and a pre-defined error code was sent to the diagnostic latch. If possible, an error message was also sent to the RS232 link. The main initialisation routine is now described in more detail.

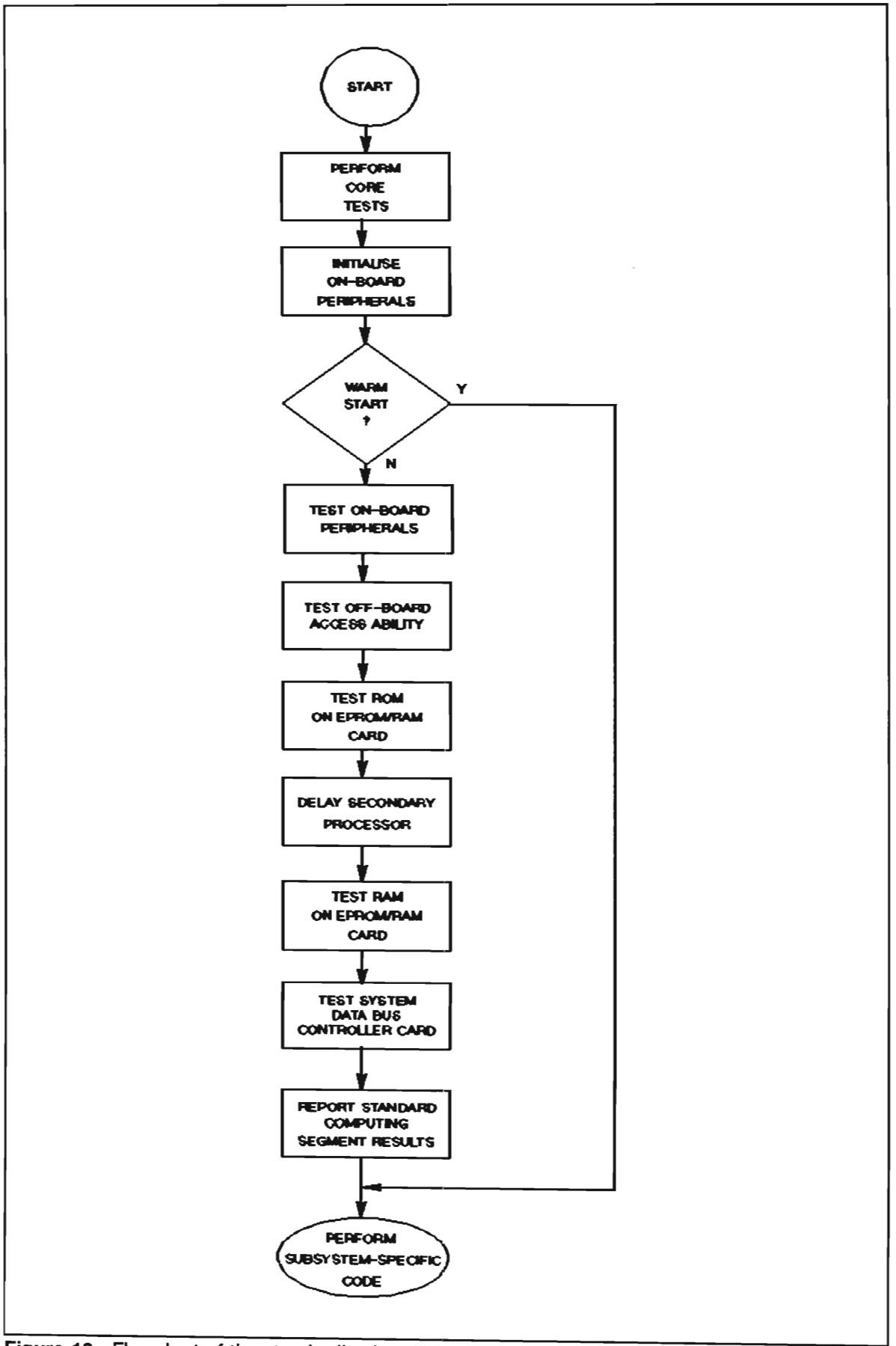

Figure 13 : Flowchart of the standardised code

5.4.1 Detection of a "warm start"

Former discussion of the 80286 microprocessor indicated that the transition from real to protected mode is achieved by setting the protected mode bit in the machine status word. A hardware reset is the only method of changing the operating mode back to real mode.

Due to the fact that the dynamic RAM card was mapped above the first Megabyte boundary, it was essential that the system POST was able to address this range. This was only possible by means of protected mode addressing and thus it was necessary for the POST to enter protected mode. However, software constraints of the subsystem-specific applications code of the subsystem described required that the entry to the subsystem-specific applications code be performed in real mode.

The problem was solved in the following manner. Once the primary processor had performed the core system tests of microprocessor, ROM and RAM and initialised the on-board peripheral chips, it interrogated an on-board variable to determine whether the system was powered up from "cold" (i.e. the variable was uninitialised) or whether a "warm" start had occurred (the variable was initialised).[2] If a "cold" start was detected, the system performed both the real mode and protected mode POST. On successful completion of the POST (i.e. no critical failures occurred), the code determined from the system description whether the applications code was to run in real or protected mode. If the system description indicated the latter, then the protected mode applications code was performed immediately. However, if the applications code was to be entered from real mode, the boot code initialised the "warm" start variable and then issued a system reset by means of a port write to the diagnostic status latch. After resetting and performing the core system tests and initialisation a second time, the processor detected a "warm" start and began the execution of the applications code in real mode.

[2] The on-board RAM test was designed in such a manner that this variable remained unchanged.

Although the standardised code was executed entirely in real mode, the overall standardisation effort catered for similar constraints regarding the entry mode for all *n* subsystems. By specifying that the system description indicate the applications mode of operation, it was possible to standardise portions of the protected mode POST. By adopting the object-oriented design approach, generic modules relating to the testing of common hardware in protected mode became packages and were applied, if needed, to each subsystem. The hardware and software system reset solution described above was one such package that could be incorporated into subsystem-specific POST in order to return to real mode.

5.4.2 Testing of on-board peripheral chips

Once the core system had been tested, the on-board peripheral chips initialised and the run-time evaluation had decided that a "cold start" had occurred, the system began to test the functionality of the on-board peripheral chips. Due to the interaction of some of the on-board peripheral chips, the fault modelling theory presented in chapter two was applied at this level. The appropriate interaction diagram is presented in Figure 14.

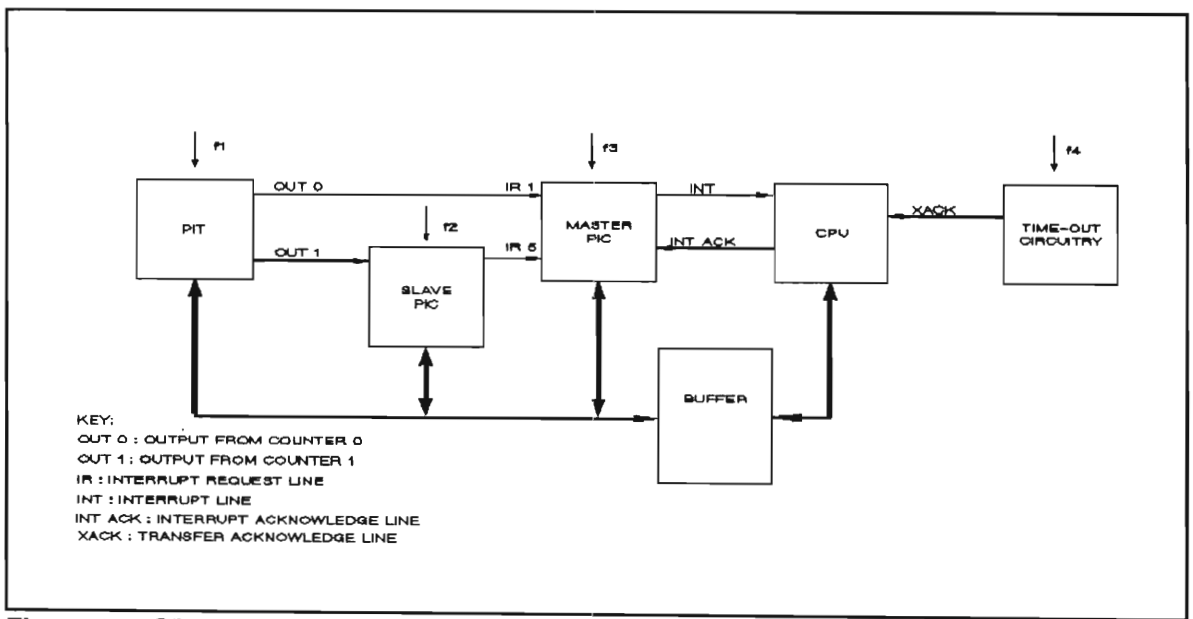

Figure 14 : CPU on-board peripheral chips interaction diagram

Following the theory presented in chapter two, the assumption was made that fault pattern F was restricted to consist of a good functional block plus functional blocks containing exactly one of the faults of the fault set FS = $\{f_1, f_2, f_3, f_4\}$, i.e. F = $\{F_0, F_1, F_2, F_3, F_4\}$. The test set was TS = $\{t_1, t_2, t_3\}$. Test t_1 represented a combination of testing the master programmable interrupt controller and time-out circuitry by performing a write to an invalid port. A time-out interrupt was expected to be generated from this action. Test t_2 tested the master programmable interrupt controller and counter 0

of the programmable interval timer by unmasking the interrupt from counter 0 and delaying long enough to allow the programmable interval timer to generate an interrupt. Test t_3 was similar to test t_2, with the exception that counter 1 was now expected to interrupt. In addition to testing counter 1 of the programmable interval timer and testing the master programmable interrupt controller again, the testing loop now included the slave programmable interrupt controller since counter 1 was routed through this device.

The fault-pattern-test-pattern event space is presented in Table XIV. Inspection of Table XIV yields the syndrome

$$\begin{aligned} S = \quad & f_1(t_1't_2t_3') \\ & + f_1(t_1't_2't_3) \\ & + f_2(t_1't_2't_3) \\ & + f_3(t_1t_2t_3) \\ & + f_4(t_1t_2't_3'). \end{aligned}$$

Therefore, if the test results were $t_1 = 1$, $t_2 = 1$ and $t_3 = 1$, then $S = f_3$, implying that the master programmable interrupt controller was faulty. However, if $t_1 = 0$, $t_2 = 0$ and $t_3 = 1$, then $S = f_1 + f_2$ and it was indeterminate whether (i) counter 1 of the programmable interval timer failed or (ii) the slave programmable interrupt controller failed.

Table XIV : Fault-pattern-test-pattern event space for the CPU card interactive peripheral chips

Faulty circuitry	Fault pattern	Test pattern ($T_v = \{t_1t_2t_3\}$)
None	F_0	000
Programmable interval counter 0	F_1	010
Programmable interval counter 1	F_1	001
Slave programmable interrupt controller	F_2	001
Master programmable interrupt controller	F_3	111
Time-out circuitry	F_4	100

The other software testable functional blocks on-board the CPU card were defined to be non-critical and were tested as independent units. The testing of the multiple protocol serial controller serial I/O interface was tested by means of the transmission of a sequence of instructions to clear the visual display unit screen connected to the RS232 link. Each time data was transmitted, the multiple protocol serial communications chip was expected to generate an interrupt. The success or failure of the generation of such interrupts was recorded by the testing routine.

The 80287 numeric coprocessor was tested by performing a relatively complex real number calculation and checking whether the actual run-time result was within an acceptable range when compared to the expected result.

5.4.3 Testing the remaining standard computing segment

Once the standardised code had established the integrity of the CPU card it began testing the rest of the standard computing segment. The presence of the EPROM/RAM card was defined to be critical so the software had to first test its ability to perform off-board accesses. To achieve this, the standard computing segment was considered at board level in order to apply the theory presented in chapter two. The functional block for this segment is presented in Figure 15. Table XV shows the fault-pattern-test-pattern event space at this level, yielding

$$
\begin{aligned}
S = \quad & f_1(t_1 t_2') \\
& + f_1 f_2(t_1 t_2) \\
& + f_3 f_4(t_1 t_2) \\
& + f_5(t_1 t_2),
\end{aligned}
$$

where test t_1 performed a read of the identification byte in the system description of the EPROM/RAM card and test t_2 read the test status of the system data bus controller card.

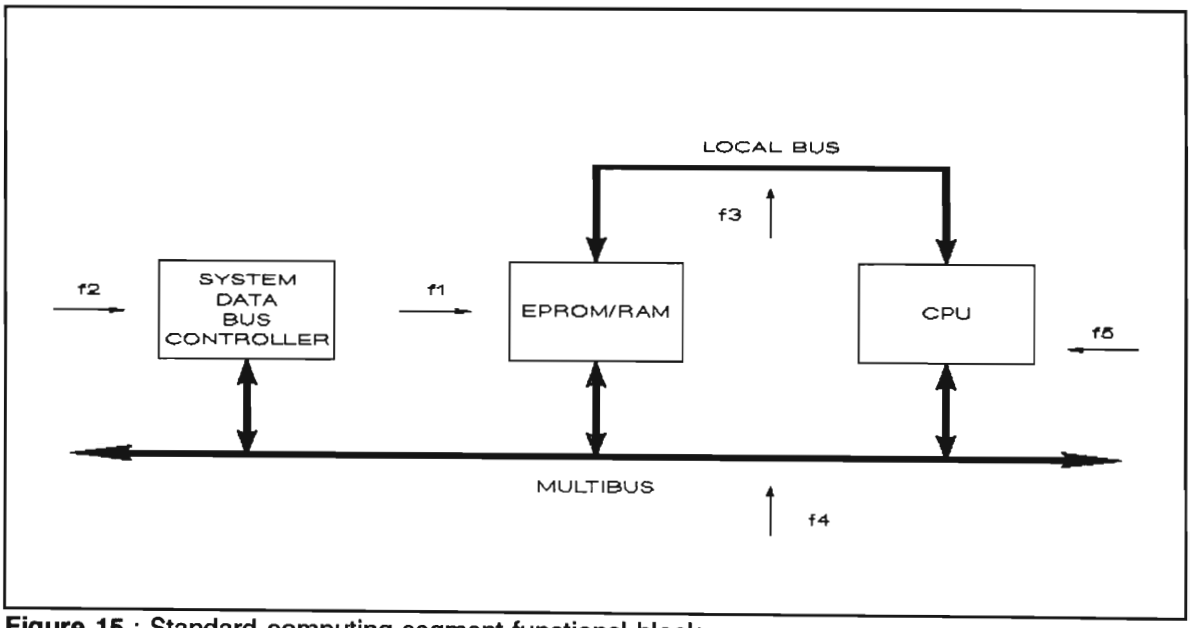

Figure 15 : Standard computing segment functional block

The procedure applying the theory considered the cases of (i) either card absent or faulty, (ii) either bus faulty, (iii) both cards absent or faulty or (iv) both busses faulty. It first read the EPROM/RAM card and, if a time-out occurred, this implied that either (i) the card was absent or (ii) both busses were faulty or (iii) some off-board access mechanism on the CPU card was faulty. Thus to determine

Table XV : Off-board access fault-pattern-test-pattern event space

Faulty circuitry	Fault pattern	Test pattern ($T_v = \{t_1t_2\}$)
None	F_0	0X
EPROM/RAM card	F_1	10
System data bus controller card	F_2	0X
Local bus	F_3	0X
Multibus	F_4	0X
Both cards	$F_1 + F_2$	11
Both busses	$F_3 + F_4$	11
Off-board access ability	F_5	11

the integrity of the Multibus under these circumstances, the code attempted to read the system data bus controller card. If a time-out occurred once again, then $t_1 = 1$ and $t_2 = 1$ and therefore $S = f_1f_2 + f_3f_4 + f_5$, showing that either (i) both cards were absent/faulty or (ii) both busses were faulty or else that (iii) the CPU accessing ability was faulty. In any event, the fault was considered to be critical and thus an appropriate error message was sent to the diagnostic terminal and further execution was suspended.

If fault pattern F_1 was present (i.e. the EPROM/RAM card was absent/faulty), a time-out was generated when test t_1 was applied and the test failed. However test t_2 passed so that $t_1 = 1$ and $t_2 = 0$. In such an instance the standardised code immediately sent an error code to the hexadecimal displays, transmitted an error message down the RS232 link and halted. If, however, no time-out occurred when t_1 was applied, but, instead, the value of the EPROM/RAM identification byte was incorrect, then the system transmitted a "system description help screen" to the visual display unit. This catered for the development environment where the subsystem designer had omitted or erroneously entered the system description.

Once the standardised code had established its ability to perform off-board accesses, the EPROM/RAM card was tested. The generic ROM and RAM tests were applied to the card at the appropriate addresses and a failure of either was deemed critical. The code dynamically assessed the RAM/ROM boundary by consideration of the RAM/ROM option byte in the system description. If the strapping of this boundary did not correspond to the option provided in the firmware, this implied one of two cases, viz. either (i) all of the RAM or a portion thereof would be treated as ROM or else (ii) all of the ROM or a portion thereof would be treated as RAM. In case (i), the long division

would now have included extra memory blocks, but the probability of the actual checksum corresponding to the expected checksum was extremely small, and, consequently the ROM test would have most likely failed. Given that all else was operating correctly, case (ii), on the other hand, would have definitely failed, because any attempts to write to the ROM would have resulted in the generation of a time-out.

Once again the standardised code catered for the development environment by determining whether the checksums were present on the EPROM/RAM card prior to testing the ROM. If no checksums were available, the test was still performed but the failure was ignored. This enabled the standardised code to calculate and inform the designer regarding the expected checksum that needed to be inserted into the system description.

Up to this point all CPU cards in all subsystems would have been concurrently executing standardised code. However, because the code had now moved to testing modules at subsystem level and off-board RAM was about to be tested, it was appropriate to delay the secondary processors. The standardised code interrogated the system description to determine the number of processors, and, on discovering a dual processor environment, delayed the secondary processor in the manner described earlier. The RAM on the EPROM/RAM card was then tested using the generic method previously discussed.

Following successful completion of the testing of the EPROM/RAM card, the standardised code retrieved the self-test results from the system data bus controller card and performed a test on the Multibus port of the dual ported RAM. An absent or faulty system data bus controller card was considered as non-critical because it was still possible to operate the system without inter-subsystem communication.

The standardised code concluded the power on procedure by analyzing the test results, and, if possible, reporting these to the RS232 link. The system number and version of the subsystem-specific code together with the calculated checksums of both the CPU and EPROM/RAM cards was also transmitted. Control was then passed to the firmware resident on the EPROM/RAM card by means of an absolute jump to the real mode address specified in the system description. This subsystem-specific code then began execution of POST routines pertinent to the particular hardware of the relevant subsystem.

5.5 DESIGNING THE SUBSYSTEM-SPECIFIC CODE

Complying to standardisation requirements at system level imposed restrictions on the subsystem-specific code. Reducing these imposed restrictions increased the likelihood of system level standardisation acceptance. The constraints inflicted by the standardised code were for the subsystems to comply with the standardised-to-subsystem-specific code and data interfaces. The code interface consisted of:

(a) reserving an area on the ROM of the EPROM/RAM card for the system description and ensuring the correctness of such a description
(b) considering the state of the CPU on-board peripheral chips at the time of the transfer of control and changing this state if necessary by the means of standardised library hardware driver routines
(c) Linking to the standardised code in order to perform off-line diagnostics on the standard computing segment.

The data interface, on the other hand, needed to reserve certain areas of RAM to enable the recording of such items as test results, the detection of a "warm" start and static and dynamic processor identification variables.

With the exception of these interface requirements, however, subsystem-specific code needed to be restrained no further by system level standardisation. The code had to still consider the subsystem and component levels of standardisation, though. The flowchart for the subsystem-specific code is illustrated in Figure 16 and described below.

The code resident on the EPROM/RAM card first determined whether a "cold" or "warm" start occurred. If a "warm" start was detected, the real mode demonstration was performed immediately. However, if the system detected a "cold" start then the power-on procedure was continued based on the subsystem hardware.

Since both the primary and secondary CPU cards executed standardised code, they both tested the standard computing segment. However, once control was passed to the subsystem-specific applications code (i.e. the executing code was resident on the EPROM/RAM card), the two processors rendezvoused as depicted in the flow diagram of Figure 17.

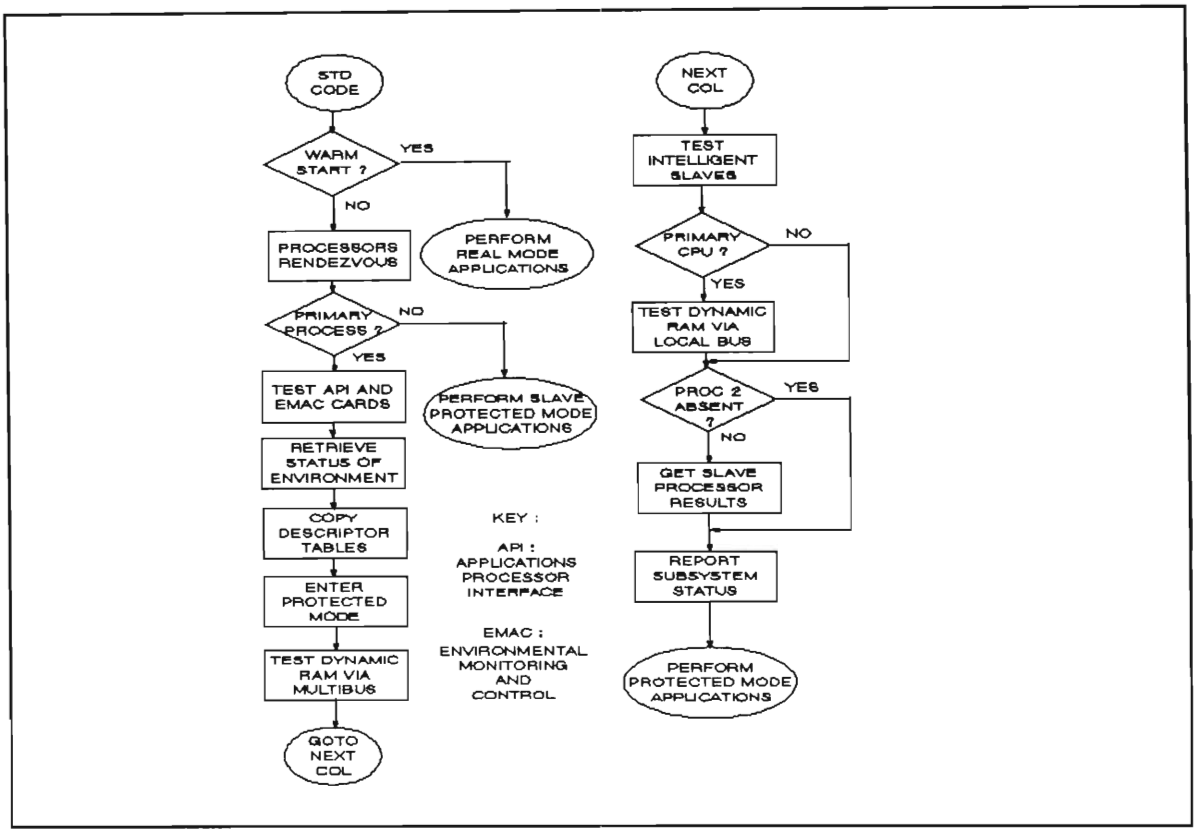

Figure 16 : Flowchart of the subsystem-specific code

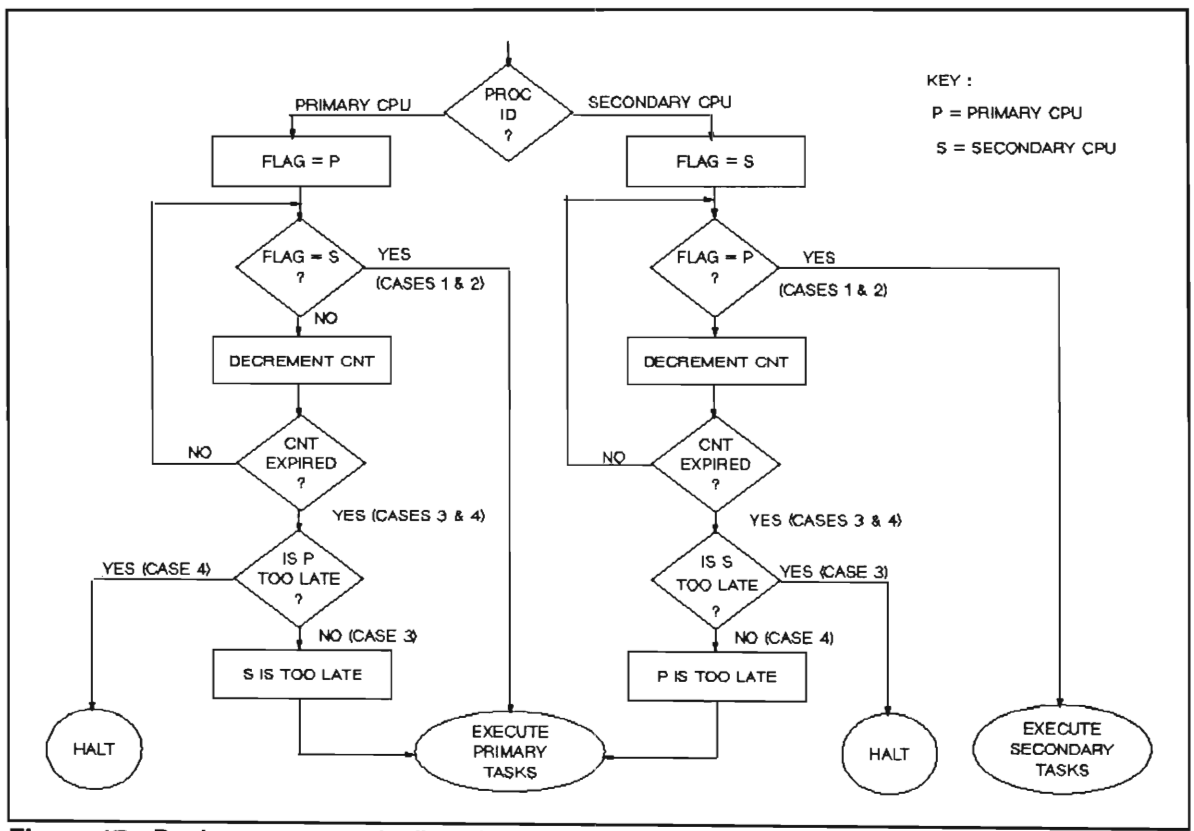

Figure 17 : Dual processor task allocation flowchart

Although the processors knew their respective identifications at this point, the code catered for the run-time cases cited below:

Case 1 : The primary processor reaching the rendezvous point just before the secondary processor.

Under normal operating conditions, the primary processor reached the rendezvous point just before the secondary processor. When this was the case, the primary processor initialised the synchronisation flag and waited for the secondary processor. The secondary processor reset the synchronisation flag and then waited for the handshake to be completed by the primary processor. Following this, each processor performed their normal respective tasks.

Case 2 : The secondary processor reaching the rendezvous point just before the primary processor.

In spite of the delay that the secondary processor had already executed during the standardised code, the secondary processor could still have reach the rendezvous point before the primary processor due to the differing physical clock speeds on-board each CPU card. This was catered for in a similar manner to the normal mode of operation, except that the secondary processor was the first processor to initialise the synchronisation flag.

Case 3 : Either (i) the primary processor reaching the rendezvous point *long* before the secondary processor or (ii) the primary processor reaching the rendezvous point and the secondary processor *never* reaching the rendezvous point.

When this condition arose, the pre-defined count of the primary processor expired before the synchronisation flag was reset. The primary processor then assumed that the secondary processor was faulty and operation continued in a degraded fashion. If the secondary processor eventually arrived at the rendezvous point, it reported an error and halted.

Case 4 : Either (i) the secondary processor reaching the rendezvous point *long* before the primary processor or (ii) the secondary processor reaching the rendezvous point and primary processor *never* reaching the rendezvous point.

> In this case the secondary processor initialised the synchronisation flag and waited for a pre-determined period for the arrival of the primary processor. Eventually, the secondary processor realised that the primary processor was going to miss the rendezvous. The duties of the primary processor were therefore shifted to the secondary processor. If the primary processor did eventually arrive, it would see that its functions had already been performed by the secondary processor, report an error and halt.

Ideally, the first operation that should have been performed after determining a "cold" start was that of the dual processor rendezvous. Since the EPROM/RAM cards were standardised at subsystem level, both CPU cards would still have been executing identical code at this stage. Identification had to therefore be performed dynamically in accordance with the flowchart of Figure 17, thus requiring RAM accessible to both CPU cards. Up to this point, the only tested global RAM was that resident on the system data bus controller card. However, this area could not be used for the handshaking because the system data bus controller card was defined to be non-critical, and, thus the RAM might not have been available at power-on. Instead, RAM that was both critical and accessible to the two CPU cards was identified on the applications processor interface card. This card was justified as critical because of its interface to the environmental monitoring and control circuitry and to the operator. For this reason, both CPU cards performed tests on the applicable handshaking locations on the applications processor interface card prior to performing their own task identification.

When the processors rendezvoused successfully, the secondary processor performed no further activities other than to enter protected mode and become a system slave, waiting for a command from the host. After the rendezvous, the primary processor initiated all further subsystem activities and in this way the entire configuration changed from a "dual-master" to a "single-master" environment. The procedure that followed was the same irrespective of which CPU card was executing as the primary processor - the only difference being that those data structures passed via global subsystem RAM to the secondary CPU card were on the dynamic RAM card as opposed to the primary EPROM/RAM card.

After checking the self-test results of the applications processor interface and environmental monitoring and control cards and testing the dual port RAM of the former, the primary processor retrieved all status and environmental information (e.g. monitoring of the console for stuck-key detection, temperature and humidity conditions) reported by the environmental monitoring and control card. Critical conditions resulted in the system halting in the usual manner. Some conditions were defined to be "abnormal", but not critical. In such cases the operator was notified, but the system software did not suspend execution.

The next item to be tested was the dynamic RAM card. In order to access the dynamic RAM card, the code had to enter protected mode. The method used to enter protected mode was similar for both processors. Protected mode requires descriptor tables in order to perform the correct addressing [GLASS, 1989; INTEL, 1987a; INTEL, 1987c]. Each time a segment is accessed, an "access" bit is set in the associated descriptor. This necessitated that the descriptor tables be RAM based. It was therefore essential for the ROM based code to copy the descriptor tables to RAM, prior to entering protected mode. The protected mode code was built in such a manner that it expected the descriptors to be at the reserved RAM addresses. The source address of the descriptor table to be copied depended on the run-time identification of each processor. This ensured that, in the dual processor environment, each processor received the correct descriptor table copied from ROM on the EPROM/RAM card to the respective on-board RAM locations. The software run-time decisions that were taken at this stage are indicated in the flowchart of Figure 18.

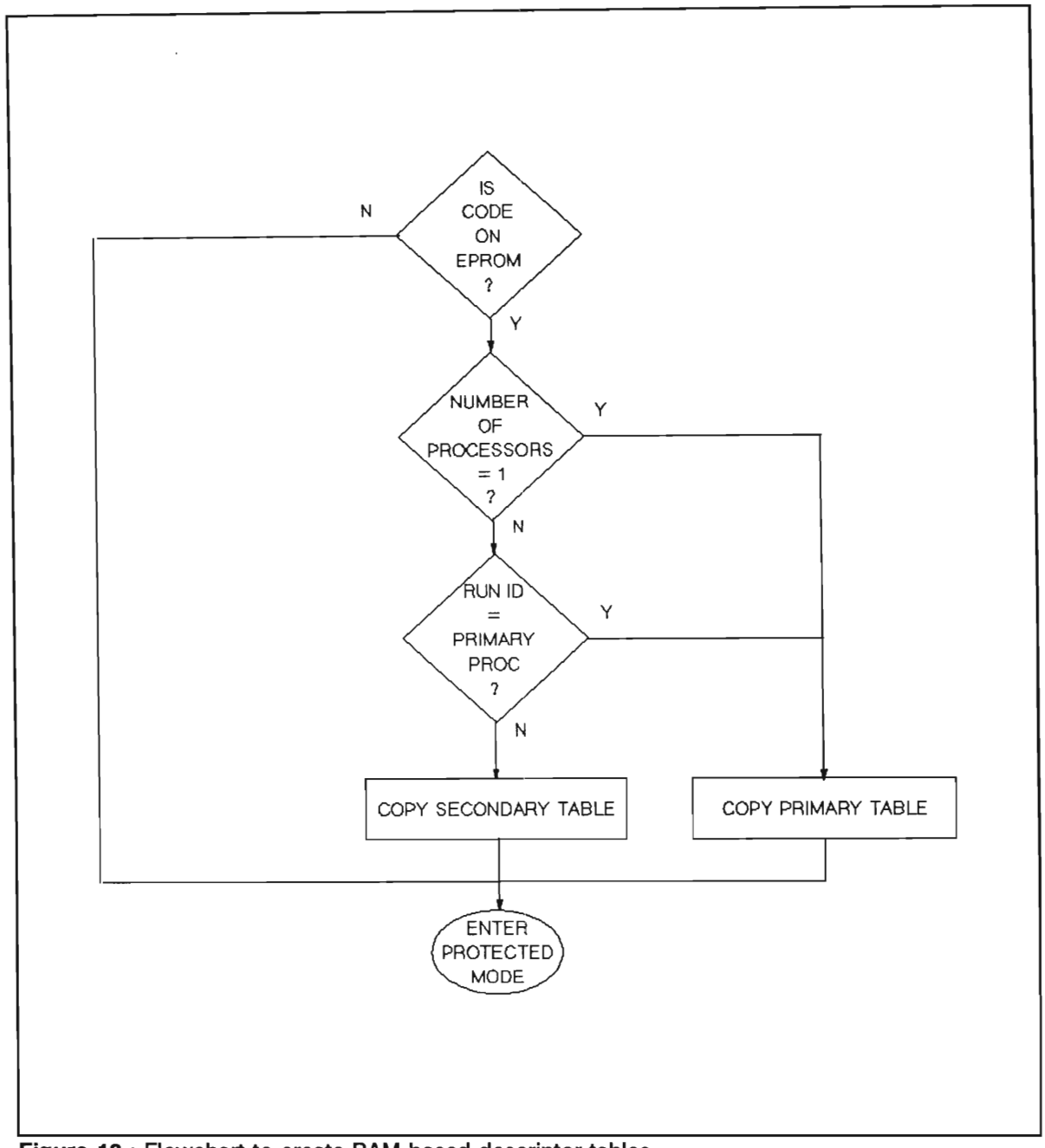

Figure 18 : Flowchart to create RAM based descriptor tables

During development, the descriptor tables were loaded into RAM by means of the in-circuit emulator used for the code debugging [INTEL, 1984 p. 5:49-5:65; INTEL, 1987b]. In this case, it was not necessary to copy the tables. The code catered for this by first establishing whether it was executing from EPROM (i.e. a pre-defined address had been programmed with a pre-determined value). If not, the copy procedure was not executed.

The module to enter protected mode set the protected mode bit in the machine status word and enabled the protected mode latch on the CPU card. The global descriptor table, interrupt descriptor table, local descriptor table and task registers were then initialised [INTEL, 1987c] and a task switch was performed to the protected mode POST.

The dynamic RAM card was then tested via the Multibus using the generic system RAM test algorithm. An absent or failed dynamic RAM card on the Multibus was defined to be critical for two reasons, namely:

(a) the dynamic RAM was used as global memory for the transfer of test results from the secondary processor to the primary processor
(b) when the secondary processor was executing as the primary processor all the intelligent slaves of the subsystem requiring buffer transfers to and from the host CPU used this area as global RAM

Following the successful completion of the dynamic RAM test via the Multibus, the three graphics, the serial communications to Multibus and the mass storage controller cards were tested as depicted in the algorithm of Table XII (p. 76). The first correctly functioning graphics interface module detected was designated the responsibility of reporting the system status. If all three graphics cards were faulty then an error message was transmitted via the RS232 link and the system halted.

If execution continued from this point, all status reporting to the appropriate graphics interface module was performed via the normal operational manner of buffer transfer as described in the interface description of the previous chapter. This follows the "expanding kernel" principle outlined earlier of only using hardware that has been previously tested. At this stage, the POST had proved that :

(a) at least one processor card had passed all on-board critical tests
(b) the EPROM/RAM card was fully operational
(c) the dynamic RAM card was fully operational over the Multibus
(d) there was at least one graphics interface module present in the system

When the primary CPU was executing the primary process it performed a dynamic RAM test over the local bus. Due to the architecture of the system, however, this test and any applications code references to dynamic RAM local bus addresses were invalid when the secondary processor was executing the primary process. Under these circumstances, this fact was flagged and the operator

was informed regarding the degraded operation. The interface requirements specified the necessity of providing fault tolerant applications code due to the possibility of such a degradation in operation [OLANDER and BAUDIN, 1990].

Provided that the processors had successfully completed their rendezvous, the primary processor interrogated the secondary processor at this stage. This interrogation resulted in the on-board test results of the secondary processor being reported to global RAM. The secondary processor was then allocated a graphics interface module for its particular applications output. It was possible to reallocate graphics modules at any stage during the applications.

The system then reported any unreported errors or warnings detected at power up. The operator was provided with the option of conducting more extensive off-line diagnostics or to ignore the errors/warnings and to proceed to the applications. The demonstration of the ability of the real-time embedded system to detect any fault or abnormal status of the system is highlighted in the operational demonstration procedure of appendix E.

5.6 APPLICATIONS CONSOLE FUNCTIONAL DEMONSTRATIONS

In normal applications, the fault-free system would have been reset and a "warm" start initiated. The core tests would then have been performed a second time and control transferred to the real mode applications code. However, to show the functionality of the console, applications demonstration code was written. The assumption made in writing of this code was that a normal fault-free power up had occurred. This, of course, excluded the demonstration of the ability of the off-line diagnostics to diagnose an error.

On successful completion of the POST, the system displayed the main test menu on the first graphics display unit and on the RS232 terminal connected to the diagnostic port of the console status panel module. The menu consisted of :

(a) a demonstration of the rollerball ability
(b) graphics display test patterns
(c) a demonstration of multi-master bus contention capabilities
(d) a demonstration of off-line diagnostic capabilities
(e) a demonstration of the serial communications to the Multibus
(f) a demonstration of the software initiated processor reset which resulted in the return to real mode and subsequent execution of environmental monitoring and control and mass storage controller card demonstrations

The operational demonstration procedure given in appendix E shows the implementation of these console functional demonstrations together with a simulation of faults that illustrate the ability to detect system failures.

5.7 SUMMARY

It was appropriate to show how the introduced concepts were applied to a typical real-time embedded system. This chapter showed how the software of the standard computing segment was standardised thus enabling the CPU cards to be consistent through all *n* subsystems. At subsystem level, the local memory cards and three graphics interface modules were also standardised. The power-on self tests of both the standardised code and the subsystem-specific code was then described in detail.

Two examples were presented at two different hierarchical levels illustrating the practical implementation of the theory. The first example illustrated the derivation of a Boolean expression for on-board peripheral circuitry enabling quick and easy diagnosis to functional block level. In a similar manner, the second example revealed an expression enabling the diagnosis resolution to be realised at board level.

The description of the program flow for the power on routines illustrated how the implemented code substantiated the presented philosophies. Particularly, this chapter has provided an example of the integration and practical implementation of the researched principles during the software development phase of a typical real-time embedded system.

6 CONCLUSION

The text has provided an overview of hardware failures and some resulting classical faults. Typical circuitry was modelled in a hierarchical fashion and ideas were presented in an effort to convey certain philosophies to prospective system designers.

Some important issues were raised, one being the understanding of the requirements specification that defines the goals of the diagnosis. The problem definition envisaged by both the operational and maintenance personnel may be quite diverse. It is not only essential to analyze these differences in user requirements, but also to retrieve valuable user experience on existing systems in order to provide design improvements in future projects.

The diagnostic goals need to be clearly and unambiguously stated by both class of users. The system time constraints and system overhead should be specified, highlighting diagnostic time limitations and the amount of system availability required during testing. Included in this specification should be a mention of the frequency of preventative maintenance activities, since this also affects the frequency of the on-line health monitor testing routine (a balance must be maintained to ensure a relatively high system availability). The system tactical requirements should also be specified. These requirements will include such aspects as the possibility for "battle short" conditions at power-on that allows for a POST bypass and features such as diagnostic override facilities during on-line or off-line diagnostics. Finally, existing standing orders and standardisation policies should be highlighted to enable compliance with or the possible upgrading of such philosophies. The standing orders would include the user procedures followed during power-on and off-line diagnostics, particularly in the event of hardware failure.

In order to construct a meaningful requirements specification, the requirements of both the operational and maintenance personnel should be assessed for all three testing phases. Typical questions would be :

1) What is the hierarchical depth of testing to be realised by the diagnostic routine ?
2) How many faults should the diagnostic routine assume present at any one particular instant ?
3) What is the format of the system status display ?
4) How much information is needed by the user when the system status is displayed ?
5) Is there a necessity to provide diagnostic override facilities during testing ?
6) At what level of user authority should these override facilities be aimed ?
7) What is the typical maximum time allowed for the diagnostics to execute ?

8) How much of the system should be available for the execution of the main applications task during testing ?

9) What is the frequency of preventative maintenance activities ?

10) What is the normal power-on procedure ?

11) What is the procedure followed when a hardware failure is detected at power-on ?

12) Are there any standing orders preventing the operator from bypassing the POST when this is not necessary ?

13) What is the procedure followed when an error/warning is issued by the on-line health monitor ?

14) Under what conditions are the off-line diagnostic routines invoked ?

(a) Are the off-line diagnostic routines only invoked when the on-line health monitor detects a potential problem ?

(b) Are the off-line diagnostic routines invoked at regular intervals, typically during preventative maintenance activities ?

(c) Are the off-line diagnostic routines invoked more frequently than the execution of preventative maintenance ?

15) What is the procedure followed when a hardware unit is diagnosed as faulty ?

16) What is the procedure followed when a faulty hardware unit is a standardised module and no spare module is available ?

Once these requirements have been noted, the designer must then decide on the sophistication of the testing capabilities. A typical real-time embedded system would be hierarchically decomposed into its constituent parts and the theory and policies presented applied to the system. The designer must then evaluate whether the module under consideration requires BIT and/or BITE capabilities, given externally imposed reliability, availability, maintainability and cost constraints. For example, if the testability is required to assume a passive rather than an active role, then BIT is required rather than BITE.

After deciding whether to implement BIT or BITE, the designer must then strive to provide qualitative and quantitative answers to questions relating to the nature of the capability. For BITE, this would include deciding on the location and sophistication of both the fault detection circuitry and the fault

indicators. Either of this hardware could be integrated into the module or remotely sited. Similarly, for BIT, the designer must implement the appropriate solution based on certain criteria, including :

(a) the sophistication of control over module inputs
(b) the type of access to module inputs or outputs
(c) the sophistication of fault analysis and indication

In addition to these design criteria, other factors that would influence the implemented choice would include an evaluation of the impact of the testable features on operational performance and the cost of actually implementing the option. An assessment of these costs would include a trade-off analysis between the cost of the BIT/BITE design and development together with the cost of user training.

Having appraised the test capabilities of their respective subsystems, system designers should also strive for standardisation in test philosophies. This standardisation would include :

(a) system-wide agreements on testing methodologies applied to certain hardware functions
(b) fault detection mechanisms for common modules
(c) decisions to detect or diagnose certain classical faults
(d) depth of testing requirements
(e) status and warning display requirements
(f) tactical requirements (eg. override facilities)
(g) testing time constraints
(h) testing interference with system performance

It can be seen that the generation of built-in test features for a complex real-time embedded system is a non-trivial task that requires cooperation between both system designers and system users. In order to develop worthwhile test features and to ensure efficient system development, built-in test design policies need to be agreed upon during the preliminary design phase. This text has aimed at providing a base for a general understanding of these relevant policies and to show that implementing the policies eases the task of generating a set of built-in tests for a real-time embedded system.

7 REFERENCES

[ABBOTT, 1981]
ABBOTT, R., *Equipping a Line of Memories with Spare Cells*
(Electronics, p. 127-130, 28 July 1981).

[ABRAHAM and AGARWAL, 1986]
ABRAHAM, J.A. and AGARWAL, V.K., *Test Generation for Digital Systems*
(Fault-Tolerant Computing Theory and Techniques, Prentice-Hall, Englewood Cliffs, New Jersey, vol. I, p. 1-97, 1986).

[AGNEW et al, 1965]
AGNEW, P., RUTHERFORD, D., SUHOCKI, R., YEN, C. and MULLER, D.,
An architectural Study for a Self-Repairing Computer
(USAF Space Systems Div., Final Tech. Rep. SSD-TR-64-159, Nov. 1965).

[ALLWORTH and ZOBEL, 1987]
ALLWORTH, S.T. and ZOBEL, R.N., *Introduction to Real-Time Software Design*
(Macmillan Education, London, p. 1-13 & 103-144, 1987).

[ANDERSON et al, 1982]
ANDERSON, R.T., KUS, S., RICKERS, H.C. and WILBUR, J.W., *Reliability Design and Engineering*
(Electronic Engineers' Handbook, McGraw-Hill, U.S.A., Second Edition, p. 28:1-28:58, 1982).

[ANSI, 1986]
American National Standard for Information Systems, *Small Computer Systems Interface*
(American National Standards Institute, Broadway, Ney York, x3.131-1986, 1986).

[AWST, 1981]
Aviation Week & Space Technology, *Velocity, Altitude Regimes to Push Computer Limits*
(McGraw-Hill, U.S.A., p. 49-51, 6 April 1981).

[BALLINGER and CONRADIE, 1990]
BALLINGER, J. and CONRADIE, C.,
Fundamental Differences Between the Traditional and Object-Oriented Approach to Software Development
(Proc. Symp. Object-Oriented Software Design, Wits. Univ., p. 2:1-2:7, 30-31 January 1990).

[BAUDIN, 1990a]
BAUDIN, M.S.E., *Technical Manual for the BTU card*
(UEC Projects, Mt. Edgecombe, Natal, 1990).

[BAUDIN, 1990b]
BAUDIN, M.S.E., *Software Product Specification for the Serial Communications to Multibus Card*
(UEC Projects, Mt. Edgecombe, Natal, 1990).

[BAUDIN, 1990c]
BAUDIN, M.S.E., *Software Product Specification for the Graphics Interface Modules*
(UEC Projects, Mt. Edgecombe, Natal, 1990).

[BOSE and METZNER, 1986]
BOSE, B. and METZNER, J., *Coding Theory for Fault-Tolerant Systems*
(Fault-Tolerant Computing Theory and Techniques, Prentice-Hall, Englewood Cliffs, New Jersey, vol I, p. 280-281, 1986).

[BRAHME and ABRAHAM, 1984]
BRAHME, D. and ABRAHAM, J.A., *Functional Testing of Microprocessors*
(IEEE Trans. Comput., vol. C-33, July 1984).

[BREUER and FRIEDMAN, 1976]
BREUER, M.A. and FRIEDMAN, A.D., *Diagnosis and Reliable Design of Digital Systems*
(Woodland Hills, Calif.: Computer Science Press, 1976).

[BRULE et al, 1960]
BRULE, J., JOHNSON, R. and KLETSKY, E., *Diagnosis of Equipment Failures*
(IRE Trans. Reliab. Quality Control, vol. RQC-9, no.4, p. 23-24, April 1960).

[CHANG, 1965]
CHANG, H., *An Algorithm for Selecting an Optimum Set of Diagnostic Tests*
(IEEE Trans. Electron. Comput., vol. EC-14, no. 10, p. 706-711, Oct. 1965).

[CHANG and THOMIS, 1967]
CHANG, H. and THOMIS, W.,
Methods of Interpreting Diagnostic Data for Locating Faults in Digital Machines
(Bell Syst. Tech. Journ., vol. 53, no. 8, p. 1505-1534, Oct. 1974).

[CHANG et al, 1970]
CHANG, H.Y., MANNING, E.G. and METZE, G., *Fault Diagnosis of Digital Systems*
(New York: Wiley-Interscience, 1970).

[COOPER and CHOW, 1976]
COOPER, A.E. and CHOW, W.T., *Development of on-board Space Computer Systems*
(IBM Journ. of Research and Development, vol. 20, no. 1, p. 5-19, Jan. 1976).

[FORBES et al, 1965]
FORBES, R., RUTHERFORD, D., STIEGLITZ, C. and TUNG, L., *A Self-Diagnosable Computer*
(1965 Fall Joint Comput. Conf., AFIPS Proc., vol. 27, p. 1073-1086, 1965).

[FUJIWARA, 1985]
FUJIWARA, H., *Logic Testing and Design for Testability*
(MIT Press Series in Computer Systems, Cambridge, Massachusetts, p. 1-23 & 144-149, 1985).

[GLASS, 1989]
GLASS, L.B., *Protected Mode*
(BYTE, Peterborough, NH, p. 377-384, December 1989).

[GLASS, 1990a]
GLASS, L.B., *The SCSI Bus (Part 1)*
(BYTE, Peterborough, NH, p. 267-274, February 1990).

[GLASS, 1990b]
GLASS, L.B., *The SCSI Bus (Part 2)*
(BYTE, Peterborough, NH, p. 291-298, March 1990).

[GREYLING, 1989a]
GREYLING, P.M., *Processor Module Technical Manual*
(Andromeda Electron. Syst., Jhb., 1989).

[GREYLING, 1989b]
GREYLING, P.M., *EPROM/RAM Card Technical Manual*
(Andromeda Electron. Syst., Jhb., 1989).

[GREYLING, 1989c]
GREYLING, P.M., *DRAM Card Technical Manual*
(Andromeda Electron. Syst., Jhb., 1989).

[GREYLING, 1989d]
GREYLING, P.M., *User Manual for the System Data Bus Controller Card*
(Andromeda Electron. Syst., Jhb., 1989).

[HACKL and SHIRK, 1965]
HACKL, F. and SHIRK, R., *An Integrated Approach to Automated Computer Maintenance*
(1965 IEEE Conf. Record Switching Theory Logical Design, p. 289-302, 1965).

[HAYES, 1975]
HAYES, J.P., *Detection of Pattern-Sensitive Faults in Random Access Memories*
(IEEE Trans. Comput., vol. C-24, p. 150-157, Feb. 1975).

[INTEL, 1981]
Literature Dept., Intel Corp., *An Introduction to ASM86*
(Intel Corp., Calif., 1981).

[INTEL, 1982]
Literature Dept., Intel Corp., *Component Data Catalog*
(Intel Corp., Calif., 1982).

[INTEL, 1983]
Literature Dept., Intel Corp., *MULTIBUS System Bus*
(OEM Systems Handbook, Intel Corp., Calif., p. 3:1-3:10, 1983).

[INTEL, 1984]
Literature Dept., Intel Corp., *Development Systems Handbook*
(Intel Corp., Calif., 1984).

[INTEL, 1985a]
Literature Dept., Intel Corp., *iSBC 286/12 Single Board Computer Hardware Reference Manual*
(Intel Corp., Calif., 1985).

[INTEL, 1985b]
Literature Dept., Intel Corp., *ASM86 Assembly Language Reference Manual*
(Intel Corp., Calif., 1985).

[INTEL, 1985c]
Literature Dept., Intel Corp., *PLM86 User's Guide*
(Intel Corp., Calif., 1985).

[INTEL, 1985d]
Literature Dept., Intel Corp., *iAPX 286 System Builder User's Guide for DOS Systems*
(Intel Corp., Calif., 1985).

[INTEL, 1986a]
Literature Dept., Intel Corp., *iAPX 86/88, 186/188 User's Manual Programmer's Reference*
(Intel Corp., Calif., 1986).

[INTEL, 1986b]
Literature Dept., Intel Corp., *PLM-286 User's Guide*
(Intel Corp., Calif., 1986).

[INTEL, 1987a]
Literature Dept., Intel Corp., *80286 and 80287 Programmer's Reference Manual*
(Intel Corp., Calif., 1987).

[INTEL, 1987b]
Literature Dept., Intel Corp.,
Integrated Instrumentation and In-Circuit Emulation System Reference Manual
(Intel Corp., Calif., 1987).

[INTEL, 1987c]
Literature Dept., Intel Corp., *Operating System Writer's Guide*
(Intel Corp., Calif., 1987).

[INTEL, 1988]
Literature Dept., Intel Corp., *Microcommunications Handbook*
(Intel Corp., Calif., 1988).

[INTEL, 1989]
Literature Dept., Intel Corp., *Microprocessor and Peripheral Handbook*
(Intel Corp., Calif., vol. I, 1989).

[JOHNSON, 1960]
JOHNSON, R., *An Information Theory Approach to Diagnosis*
(Proc. 6th National Symp. Reliab. Quality Control, p. 102-107, 1960).

[KAUTZ, 1968]
KAUTZ, W., *Fault Testing and Diagnosis in Combinational Digital Circuits*
(IEEE Trans. Comput., vol. C-17, no. 4, p. 352-366, April 1968).

[KIME, 1970]
KIME, C., *An Analysis Model for Digital System Diagnosis*
(IEEE Trans. Comput., vol. C-19, no. 11, p. 1063-1073, Nov. 1970).

[KIME, 1979]
KIME, C., *An Abstract Model for Digital System Fault Diagnosis*
(IEEE Trans. Comput., vol. C-28, no. 10, p. 754-766, Oct. 1979).

[KIME, 1986]
KIME, C.R., *System Diagnosis*
(Fault-Tolerant Computing Theory and Techniques, Prentice-Hall, Englewood Cliffs, New Jersey, vol. II, p. 577-632, 1986).

[KNAIZUK and HARTMANN, 1977]
KNAIZUK, J. and HARTMANN, C.R.P.,
An Optimal Algorithm for Testing Stuck-At faults in Random Access Memories
(IEEE Trans. Comput., vol. C-25, p. 1141-1144, Nov. 1977).

[KRAFT and TOY, 1981]
KRAFT, G.D. and TOY, W.N., *Microprogrammed Control and Reliable Design of Small Computers*
(Prentice-Hall, Englewood Cliffs, New Jersey, p. 163-367, 1981).

[LAW-BROWN, 1990a]
LAW-BROWN, D.C., *Console Technical Description*
(UEC Projects, Mt. Edgecombe, Natal, 16 February 1990).

[LAW-BROWN, 1990b]
LAW-BROWN, D.C., *Console Demonstration Test Procedure*
(UEC Projects, Mt. Edgecombe, Natal, 18 May 1990).

[METHA, 1989a]
METHA, N., *Development Specification for the Subsystem Console Assembly*
(UEC Projects, Mt. Edgecombe, Natal, 2 March 1989).

[METHA, 1989b]
METHA, N., *Interface Control Document of the API PCB*
(UEC Projects, Mt. Edgecombe, Natal, 31 October 1989).

[NAIR et al, 1978]
NAIR, R., THATTE, S.M. and ABRAHAM, J.A.,
Efficient Algorithms for Testing Semiconductor Random-Access Memories
(IEEE Trans. Comput., vol. C-27, p. 572-576, June 1978).

[OLANDER and BAUDIN, 1990]
OLANDER, P.A. and BAUDIN M.S.E.,
Interface Control Document of the Hardware Development with the Main Applications Computer Program Configuration Item
(UEC Projects, Mt. Edgecombe, Natal, 17 August 1990).

[PARTHASARATHY et al, 1982]
PARTHASARATHY, R., REDDY, S.M. and KUHL, J.G.,
A Testable Design for General Purpose Microprocessors
(FTCS 12th Annu. Int. Symp. Fault-Tolerant Comput., p. 117-124, 22-24 June, 1982).

[POAGE, 1963]
POAGE, J., *Derivation of Optimal Tests to Detect Faults in Combinational Circuits*
(Proc. Symp. Math. Theory Automata, p.483-528, 1963).

[POLMANS, 1990a]
POLMANS, A.J.P., *Mass Storage Card Interface Requirements Specification for Multibus Access*
(UEC Projects, Mt. Edgecombe, Natal, 1990).

[POLMANS, 1990b]
POLMANS, A.J.P., *Interface Design Document for the Mass Storage Controller Card*
(UEC Projects, Mt. Edgecombe, Natal, 1990).

[PRADHAN, 1986]
PRADHAN, D.K., *Fault-Tolerant Computing Theory and Techniques*
(Englewood Cliffs, New Jersey: Prentice-Hall, p. xiii-xvi, 1986).

[PREPARATA et al, 1967]
PREPARATA, F., METZE, G. and CHIEN, R.,
On the Connection Assignment Problem of Diagnosable Systems
(IEEE Trans. Comput., vol. EC-16, no. 6, p. 848-854, Dec. 1976).

[RAMAMOORTHY, 1967]
RAMAMOORTHY, C., *A Structural Theory of Machine Diagnosis*
(1967 Spring Joint Comput. Conf., AFIPS Proc., vol. 30, p. 743-756, 1967).

[ROBACH and SAUCIER, 1978]
ROBACH, C. and SAUCIER, G. *Dynamic Testing of Control Units*
(IEEE Trans. Comput., vol. C-27, p. 617-623, July 1978).

[ROBACH and SAUCIER, 1975]
ROBACH, C. and SAUCIER, G., *Diversified Test Methods for Local Control Units*
(IEEE Trans. Comput., vol. C-24, p. 562-567, May 1975).

[SHEDLETSKY, 1977]
SHEDLETSKY, J.J., *Random Testing : Practicality vs. Verified Effectiveness*
(Dig., 7th Annu. Int. Symp. Fault-Tolerant Comput., Los Angeles, p. 175-179, 28-30 June, 1977).

[SIEMENS, 1988]
Siemens AG, *AMS Microcomputer Board System Product Descriptions*
(Siemens AG, Bereich Halbleiter, Microcomputer Systems, Ottobrunn, 1988).

[SIEWIOREK, 1986]
SIEWOREK, D., *Architecture of Fault-Tolerant Computers*
(Fault-Tolerant Computing Theory and Techniques, Prentice-Hall, Englewood Cliffs, New Jersey, vol. II, p. 417-463, 1986).

[SKLAROFF, 1976]
SKLAROFF, J.R., *Redundancy Management Technique for Space Shuttle Computers*
(IBM Journ. of Research and Development, vol. 20, no. 1, p. 20-28, Jan. 1976).

[STEGE, 1988]
STEGE, M.J.C., *BIT/BITE Design Guidelines*
(UEC Projects, Mt. Edgecombe, Natal, 27 January 1988).

[SUD, 1981]
SUD, R. and HARDEE, K.C., *Designing Static RAMS for Yield as well as Speed*
(Electronics, p. 121-126, 28 July 1981).

[SUK and REDDY, 1981]
SUK, D.S. and REDDY, S.M.,
A March Test for Functional Faults in Semiconductor Random Access Memories
(IEEE Trans. Comput., vol. C-30, p. 982-984, Dec. 1981).

[TANENBAUM, 1981]
TANENBAUM, A.S., *Computer Networks*
(Prentice-Hall, Englewood Cliffs, New Jersey, 1981).

[THATTE and ABRAHAM, 1977]
THATTE, S.M. and ABRAHAM, J.A., *Testing of Semiconductor Random Access Memory*
(Dig., 7th Annu. Int. Symp. Fault-Tolerant Comput., Los Angeles, p. 81-87, 28-30 June 1977).

[THATTE and ABRAHAM, 1980]
THATTE, S.M. and ABRAHAM, J.A., *Test Generation for Microprocessors*
(IEEE Trans. Comput., vol. C-29, p. 429-441, June 1980).

[WENSLEY et al, 1978]
WENSLEY, J.H., LAMPORT, L., GOLDBERG, J., GREEN, M.W., LEVITT, K.N.,
MELLIAR-SMITH, P.M., SHOSTAK, R.E. and WEINSTOCK, C.B.,
SIFT: Design and Analysis of a Fault-Tolerant Computer for Aircraft Control
(Proc. IEEE, vol. 66, no. 10, p. 1240-1255, Oct. 1978).

APPENDIX A : SUBSYSTEM MAPS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS

API	: Applications processor interface
DRAM	: Dynamic RAM
EMAC	: Environmental monitoring and control
GIM	: Graphics interface modules
INT#	: Multibus interrupt number
INTA	: Interrupt acknowledge
IR	: Interrupt request
MPSC	: Multi protocol serial communications chip
MSC	: Mass storage controller
NMI	: Non maskable interrupt
NU	: Not used
OUT0	: Counter zero output
OUT1	: Counter one output
PIC	: Programmable Interrupt Controller
PIT	: Programmable interval timer
SCMB	: Serial communications to Multibus
SDBC	: System data bus controller
SER INT	: Serial interrupt

1 SUBSYSTEM MEMORY MAPS

Figure 1 shows the memory map for the full sixteen megabyte address space of the subsystem. The lowest page (i.e. page 0) is presented in Figure 2.

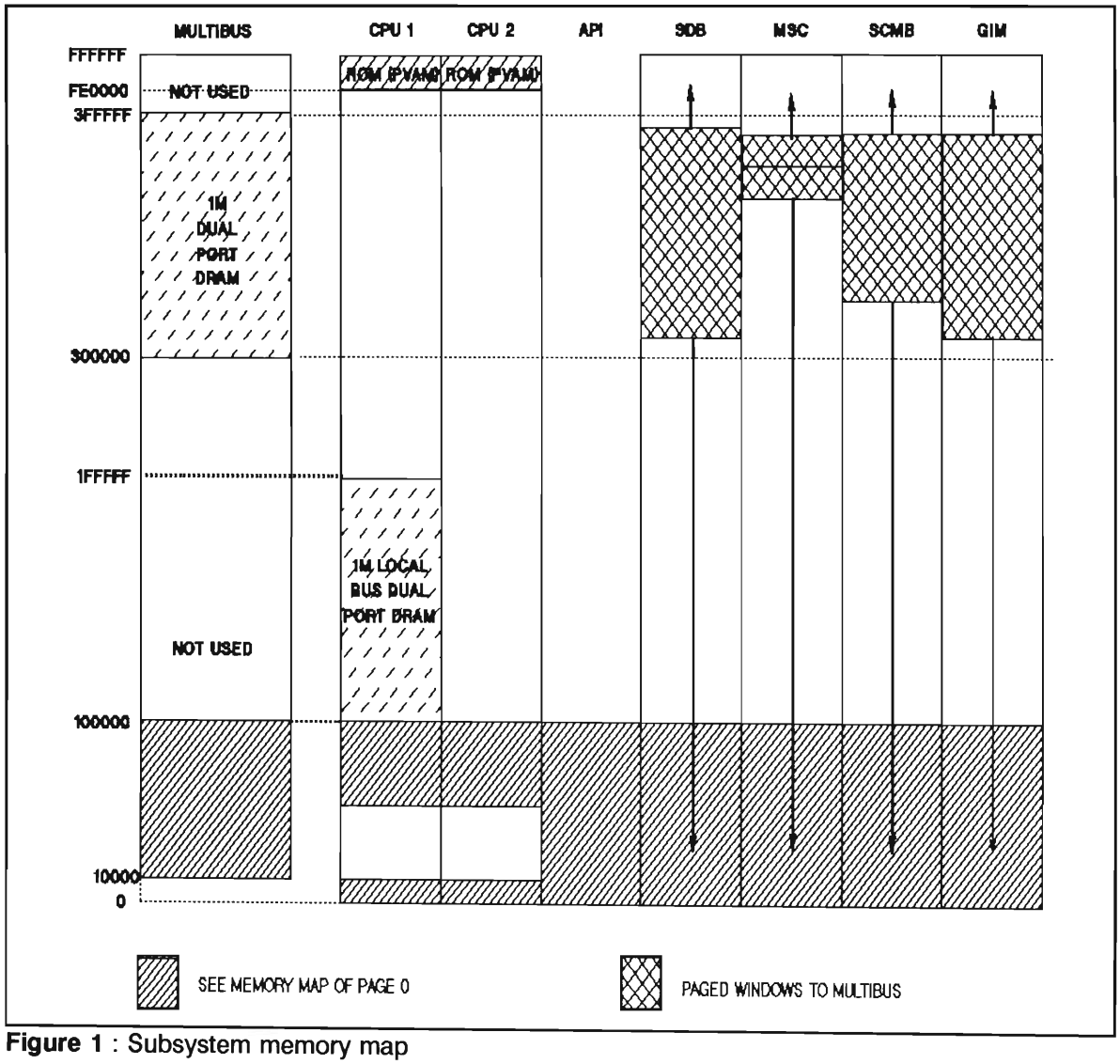

Figure 1 : Subsystem memory map

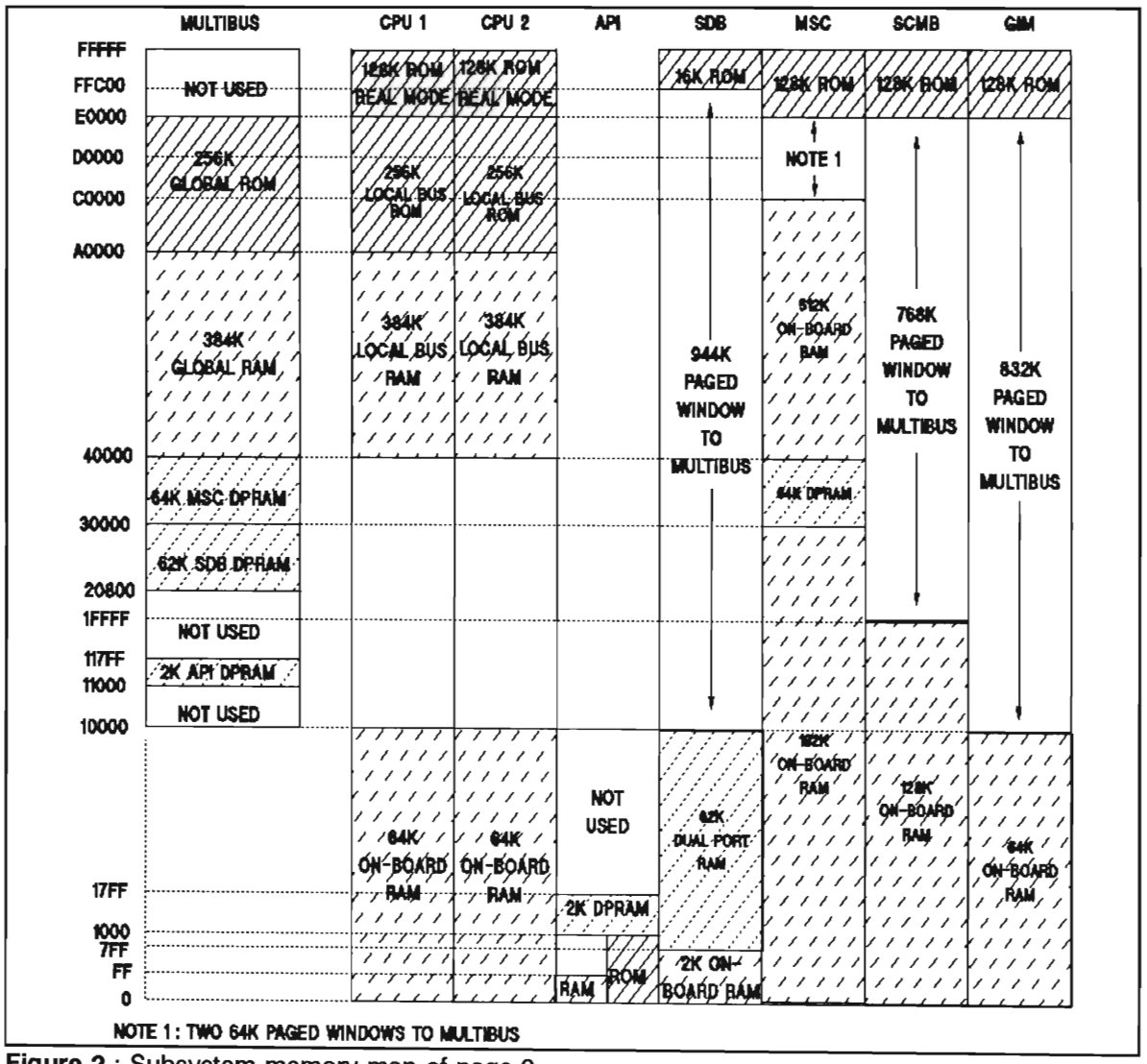

Figure 2 : Subsystem memory map of page 0

2 SUBSYSTEM I/O MAP

Figure 3 displays the I/O map as seen from the primary CPU card of the subsystem.

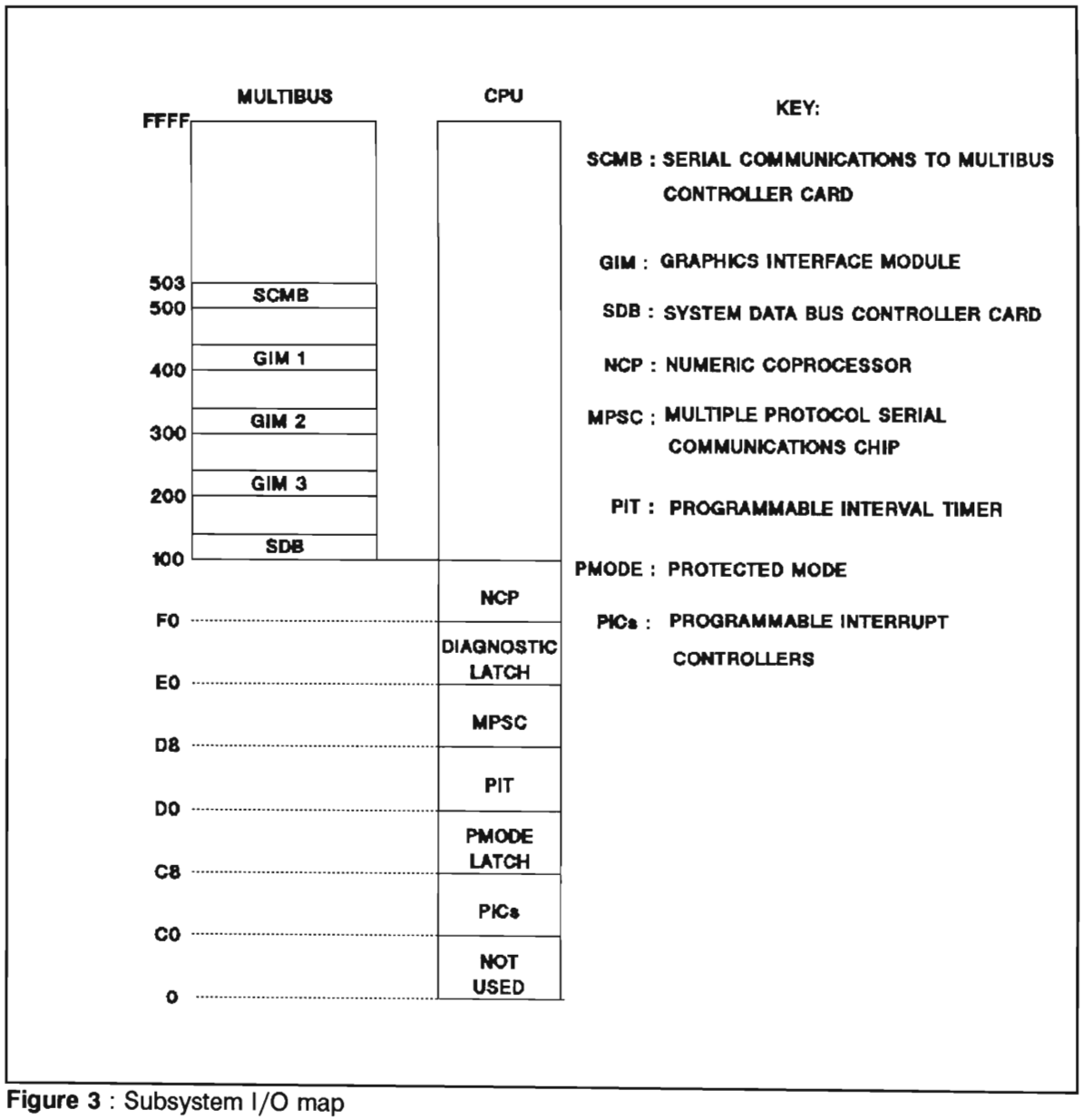

Figure 3 : Subsystem I/O map

3 SUBSYSTEM INTERRUPT STRUCTURE

The source and destination of the interrupt structure for the CPU card is shown in Figure 4 and summarised in Table I.

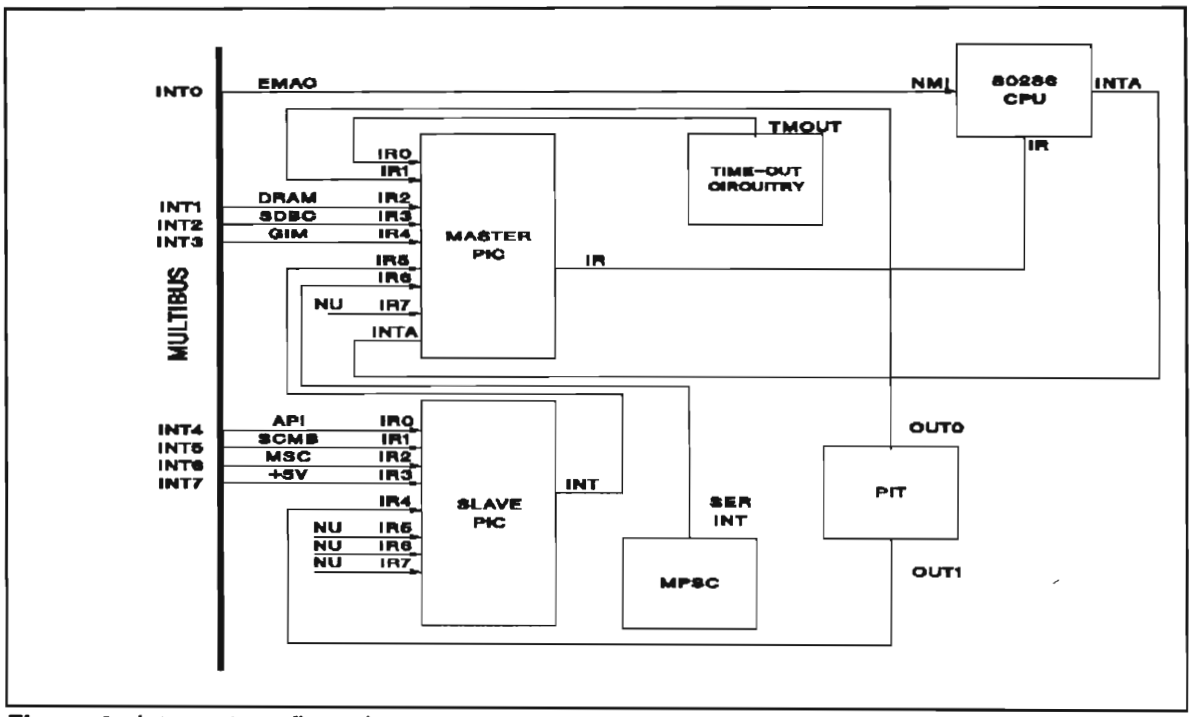

Figure 4 : Interrupt configuration

Table I : Summary of CPU card interrupt structure

Interrupt source	Route of interrupt request
EMAC via Multibus INT0	NMI 80286
Time-out	Master PIC IR0
PIT counter zero	Master PIC IR1
DRAM via Multibus INT1	Master PIC IR2
SDBC via Multibus INT2	Master PIC IR3
GIMs via Multibus INT3	Master PIC IR4
Cascaded slave PIC	Master PIC IR5
MPSC serial interface	Master PIC IR6
API via Multibus INT4	Slave PIC IR0
SCMB via Multibus INT5	Slave PIC IR1
MSC via Multibus INT6	Slave PIC IR2
Secondary processor delay	Slave PIC IR3
PIT counter one	Slave PIC IR4

Note : IR7 is intentionally not used, since this is a unique trap for interrupt requests that are removed prior to servicing.

APPENDIX B : STANDARDISED CODE LISTINGS

LIST OF TABLES

1 INTRODUCTION

The real-time embedded system software was divided into seven components, viz. :

(a) the standardised code of the standard computing segment, resident on the CPU cards
(b) the subsystem-specific applications code, resident on the system EPROM/RAM cards
(c) the system data bus controller card firmware responsible for regulating communications between the Multibus and the system data bus
(d) the applications processor interface firmware, handling most system I/O
(e) the serial communications to Multibus card, catering for the local area network communications
(f) the graphics interface modules, driving the three graphics display units
(g) the mass storage controller card providing external data storage and retrieval

This appendix applies to the software component identified in (a) above, and presents the following relevant information :

(a) the measured utilisation of RAM, EPROM and expected maximum stack requirements
(b) the measured expected minimum duration of execution
(c) a brief explanation of the source code files, together with the full source code listings.

The listings for software component (b) are presented in the subsequent appendix. Only the interfaces to software components (c) to (g) were described in the main portion of the text. With the exception of diagnostic firmware for some common on-board devices, these software components were not the work of the author, and, as a result, the source code listings have been intentionally omitted from the text.

2 SOURCE CODE

2.1 COMPILER/ASSEMBLER

The PL/M software source files were compiled using the Intel PL/M-86 Compiler Version 3.1. All source files were compiled with the following compiler controls in effect:

debug
large
rom

The compiler was invoked by the following command:

```
PLM86 filename.ext
```

All Assembler source files were assembled using the Intel 8086/87/88/186 Macro Assembler Version 2.1. The assembler was invoked by the following command:

```
ASM86 filename.ext
```

2.2 MEASURED RESOURCE UTILISATION

Table I shows the measured memory utilisation and the maximum stack allocation during the execution of the standardised code.

Table I : Standardised code resource utilisation

Resource	Utilisation/allocation
RAM	3 kB
EPROM	17 kB
Stack	16 kB

2.3 DURATION OF EXECUTION

The duration of execution of each test during POST was measured by means of a logic analyzer monitoring the diagnostic status latch on the CPU card. Each time that a new value was output to the status latch, this indicated the beginning of a new test. The test was conducted for the three various RAM/ROM configurations available by definition in the system description. Table II indicates the values recorded for the various standardised code power-on self tests.

Table II : Duration of POST execution for the standardised code

Latch value	Test	Duration (seconds)
1	CPU test	0,000 099
2	On-board ROM test	20,591 394
3	On-board RAM test	1,016 278
4	Initialisation routines	0,056 327
8	Master PIC and time-out test	0,064 641
9	PIT test	0,076 279
A	MPSC test	0,000 795
6	SPIC test	0,010 540
B	Numeric coprocessor test	0,000 568
C	Off-board access test	0,000 094
11	ROM test (option 1)	82,964 637
11	ROM test (option 2)	62,148 522
11	ROM test (option 3)	41,407 962
12	Primary processor delay	0,000 088
12	Secondary processor delay	0,010 628
13	RAM test (option 1)	1,812 597
13	RAM test (option 2)	3,634 442
13	RAM test (option 3)	5,448 683
20	SDB controller self-test	0,000 110
21	SDB RAM test	3,732 663
30	Local bus test	0,028 897

The total POST execution time for each card was determined from Table II as shown below :

Total duration of CPU card tests	= 21,817 015 s
Total duration of EPROM/RAM card tests :	
- Primary processor (option 1)	= 84,777 322 s
- Primary processor (option 2)	= 65,783 052 s
- Primary processor (option 3)	= 46,856 733 s
- Secondary processor (option 1)	= 84,787 862 s
- Secondary processor (option 2)	= 65,793 592 s
- Secondary processor (option 3)	= 46,867 273 s
Total duration of SDB controller tests	= 3,732 773 s
Total duration of bus tests	= 0,028 897 s

The total execution time for the standardised POST is depicted in Table III.

Table III : Total standardised POST execution time

Processor card	EPROM/RAM option	Approximate duration (seconds)
Primary processor	1	110,356
Primary processor	2	91,362
Primary processor	3	72,435
Secondary processor	1	110,367
Secondary processor	2	91,377
Secondary processor	3	72,446

2.4 EXPLANATION OF SOURCE CODE

A brief explanation of the function of each software module pertinent to the standardised code is provided below. The modules are subdivided into the following classes :

(a) PL/M 86 files (<filename>.PLM)

These are the high-level language source files.

(b) ASM 86 files (<filename>.ASM)

These are the assembler code source files.

(c) Entities publicly declared files (<filename>.EPD)

Any entity declared publicly in a source file is declared externally in the respective EPD file.

(d) Included files (<filename>.INC)

The included files contain general compiler commands and constants declarations.

(e) Library files (<filename>.LIB)

The library files consist of four object module files designed to drive the 80287 numeric coprocessor. These executable files were bought directly from the supplier, and, thus the source code is not listed in this appendix.

(a) PLM86 files

1)	STDEXEC.PLM	Standardised code driving routine
2)	STDINIT.PLM	CPU card standard initialisation
3)	STDBIT.PLM	Standard computing segment BIT routines
4)	STDINTS.PLM	Standard computing segment interrupt handlers
5)	STDPIC.PLM	Standardised code for the 8259A PIC
6)	STDPIT.PLM	Standardised code for the 8254 PIT
7)	STDSTAT.PLM	Diagnostic status latch standardised code
8)	STDDIAG.PLM	Standard computing segment off-line BIT
9)	STDIO.PLM	8274 MPSC chip routines
10)	STDCONVT.PLM	Conversion routines

(b) ASM86 files

1) STDCPU.ASM	CPU card microprocessor test
2) STDRAM.ASM	CPU card RAM test
3) STDERAM.ASM	EPROM/RAM card RAM test
4) STDSDB.ASM	Interface to SDB card

(d) Entities publicly declared files

1) STDINIT.EPD	Initialisation EPD file
2) STDBIT.EPD	BIT routines EPD file
3) STDINTS.EPD	Interrupt handlers EPD file
4) STDPIC.EPD	8259A PIC EPD file
5) STDPIT.EPD	8254 PIT EPD file
6) STDSTAT.EPD	Status latch EPD file
7) STDDIAG.EPD	Off-line BIT EPD file
8) STDIO.EPD	8274 MPSC EPD file
9) STDCONVT.EPD	Conversion routines EPD file
10) STDCPU.EPD	CPU card microprocessor test EPD file
11) STDRAM.EPD	CPU card RAM test EPD file
12) STDERAM.EPD	EPROM/RAM card RAM test EPD file
13) STDSDB.EPD	SDB interface EPD file

(e) Included files

1) PLMPAR.INC	Compiler Controls
2) LITS.INC	Global literals
3) PICLITS.INC	CPU card 8259A PIC literals
4) PITLITS.INC	CPU card 8254 PIT literals
5) IOLITS.INC	CPU card 8274 I/O literals
6) STATLITS.INC	CPU card diagnostic latch literals

2.5 SOURCE CODE LISTINGS

2.5.1 PLM86 files

2.5.1.1 STDEXEC.PLM

```
$ include (PLMPAR.INC)

/*****************************************************************************
 *                                                                           *
 *        MODULE NAME : EXEC                                                 *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Source Filename  : STDEXEC.PLM                                          *
 *                                                                           *
 *   Source Compiler  : PLM86                                                *
 *                                                                           *
 *   Operating System : DOS 3.10                                             *
 *                                                                           *
 *   Description      : Executive driving routine for                        *
 *                      the standardised code.                               *
 *                                                                           *
 *   Public procedures: None                                                 *
 *                                                                           *
 *   EPD files        : STDINIT.EPD                                          *
 *                                                                           *
 *   Include files    : PLMPAR.INC                                           *
 *                                                                           *
 *                                                                           *
 *****************************************************************************

 $ eject

 *****************************************************************************
 *                                                                           *
 *   HISTORY   Version 1.0 :                                                 *
 *                                                                           *
 *             Designed by : P.A. OLANDER    Date :  May 1989              *
 *             Description : Original                                        *
 *                                                                           *
 *                                                                           *
 *****************************************************************************/
```

```
EXEC: do;

/*************************************************************************
*                                                                        *
*                    Global variable database                            *
*                                                                        *
*************************************************************************/

declare EXECUTIVE                   label public;
declare REAL_MODE_CODE              label external;
/* REAL_MODE_CODE is a public label in STDERAM.ASM */

$ include (STDINIT.EPD)

declare FOREVER byte;

EXECUTIVE:
   disable;
   call INITIALISE_ALL;     /* Perform POST and initialisation */
   goto REAL_MODE_CODE;     /* Jump to address specified       */
                            /* in system description           */

end EXEC;
```

2.5.1.2 STDINIT.PLM

```
$ include (PLMPAR.INC)

/*****************************************************************************
 *                                                                           *
 *        MODULE NAME : INIT                                                 *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Source Filename  : STDINIT.PLM                                          *
 *                                                                           *
 *   Source Compiler  : PLM86                                                *
 *                                                                           *
 *   Operating System : DOS 3.10                                             *
 *                                                                           *
 *   Description      : Initialisation routines for the                      *
 *                      Andromeda HECS 1442 CPU card.                        *
 *                                                                           *
 *   Public procedures: INITIALISE_ALL                                       *
 *                      PRINT_RESULTS                                        *
 *                                                                           *
 *   EPD files        : STDIO.EPD                                            *
 *                      STDSTAT.EPD                                          *
 *                      STDPIC.EPD                                           *
 *                      STDPIT.EPD                                           *
 *                      STDINTS.EPD                                          *
 *                      STDRAM.EPD                                           *
 *                      STDERAM.EPD                                          *
 *                      STDSDB.EPD                                           *
 *                      STDBIT.EPD                                           *
 *                      STDCONVT.EPD                                         *
 *                                                                           *
 *   Include files    : PLMPAR.INC                                           *
 *                      LITS.INC                                             *
 *                                                                           *
 *                                                                           *
 *****************************************************************************

$ eject
```

```
/*****************************************************************************
*                                                                            *
*   HISTORY   Version 1.0 :                                                  *
*                                                                            *
*             Designed by : P.A. OLANDER    Date :  May 1989               *
*             Description : Original                                         *
*                                                                            *
*                                                                            *
*****************************************************************************/

INIT:           DO;

declare IO_BASE_ADDRESS  literally '0d8h';
declare IO_INTERRUPT_NO  literally   '80';
declare IO_BAUD_RATE     literally '9600';

$ eject
$ include (STDIO.EPD)
$ eject
$ include (LITS.INC)
$ eject
$ include (STDSTAT.EPD)
$ eject
$ include (STDPIC.EPD)
$ eject
$ include (STDPIT.EPD)
$ eject
$ include (STDINTS.EPD)
$ eject
$ include (STDRAM.EPD)
$ eject
$ include (STDERAM.EPD)
$ eject
$ include (STDSDB.EPD)
$ eject
$ include (STDBIT.EPD)
$ eject
$ include (STDCONVT.EPD)
$ eject
declare REAL_MODE_CODE              label external;
```

```
/*****************************************************************************
 *                                                                           *
 *                        INITIALISE PORT DATA                               *
 *                                                                           *
 * The values below initialise the 8274 MPSC as follows :                    *
 *                                                                           *
 * WR4 : Divide by 16 clock ; 8-bit sync ; 2 stop bits ; odd parity ;        *
 *       parity disabled                                                     *
 * WR1 : Disable wait ; Wait on transmit ; interrupt on all receive ;        *
 *       parity does not affect vector ; status affects vector ;             *
 *       no transmit interrupt ; no external interrupt                       *
 * WR2A: Pin-10 is RTS ; vectored interrupt ; 8086 mode ; receive priority ; *
 *       both channels interrupt mode                                        *
 * WR2B: This register is the vector table base address                      *
 * WR3 : Receive 8 bits/char ; no auto enables ; no hunt mode ;              *
 *       no receive CRC ; no address search ; no sync inhibit ;              *
 *       enable receiver                                                     *
 * WR5 : DTR on ; transmit 8 bits/char ; no break ; enable transmitter ;    *
 *       no SDLC CRC ; RTS on ; no transmit CRC                              *
 *                                                                           *
 *                                                                           *
 *****************************************************************************/

INITIALISE_PORT_DATA: procedure;

   call INITIALISE_IO_ADDRESSES (PORT_A,
                                 IO_BASE_ADDRESS,
                                 IO_INTERRUPT_NO);
   call INITIALISE_PORT( PORT_A,
                         IO_BAUD_RATE,
                         EIGHT_DATA_BITS,
                         TWO_STOP_BITS,
                         NO_PARITY,
                         TERMINAL);
   call INITIALISE_PORT( PORT_B,
                         IO_BAUD_RATE,
                         EIGHT_DATA_BITS,
                         TWO_STOP_BITS,
                         NO_PARITY,
                         TERMINAL);

   end INITIALISE_PORT_DATA;

$ eject
```

```
SET_INTERRUPT_TABLE: procedure;

 /*****************************************************************************
 *                                                                            *
 * This procedure initially sets all the vectors to point to an illegal       *
 * interrupt handler and then overwrites the appropraite location for each    *
 * vector needed.                                                             *
 *                                                                            *
 *****************************************************************************/

   declare I byte;
   declare INTERRUPT_VECTOR(256) pointer at (0);

   /* First fill the vector table with a default to the illegal interrupt
      handler.                                                              */

   do I = 0 to 256;
      INTERRUPT_VECTOR (I) = INTERRUPT$PTR (ILLEGAL_INT);
      end;

   /* Now overwrite the table with appropriate vectors                     */

   call SET$INTERRUPT (0, DIVIDE_ERROR);                 /* Intel reserved */
   call SET$INTERRUPT (1, SINGLE_STEP);                  /* Intel reserved */
   call SET$INTERRUPT (2, NMI);                          /* Intel reserved */
   call SET$INTERRUPT (3, BREAKPOINT);                   /* Intel reserved */
   call SET$INTERRUPT (4, INTO_OVERFLOW);                /* Intel reserved */
   call SET$INTERRUPT (5, BOUND_RANGE);                  /* Intel reserved */
   call SET$INTERRUPT (6, INVALID_OPCODE);               /* Intel reserved */
   call SET$INTERRUPT (7, PROC_EXT_NOT_AVAILABLE);       /* Intel reserved */
   call SET$INTERRUPT (8, DOUBLE_EXCEPTION);             /* Intel reserved */
   call SET$INTERRUPT (9, PROC_EXT_SEGMENT_OVERRUN);     /* Intel reserved */
   call SET$INTERRUPT (10, INVALID_TASK);                /* Intel reserved */
   call SET$INTERRUPT (11, SEGMENT_NOT_PRESENT);         /* Intel reserved */
   call SET$INTERRUPT (12, STACK_OVERRUN);               /* Intel reserved */
   call SET$INTERRUPT (13, GENERAL_PROTECTION);          /* Intel reserved */
   call SET$INTERRUPT (16, NCP_INTERRUPT);               /* Intel reserved */
   call SET$INTERRUPT (80, CHB_TX_EMPTY); /* Used by 8274 to transmit chars */
```

```
   /* Define the Master PIC interrupt handlers                          */

   call SET$INTERRUPT (96, TMOUT);        /* To test off board accesses     */
   call SET$INTERRUPT (97, MASTER_CLOCK); /* For Timer 1 & Master PIC tests */
   call SET$INTERRUPT (98,  MASTER_INTERRUPT_2);
   call SET$INTERRUPT (99,  MASTER_INTERRUPT_3);
   call SET$INTERRUPT (100, MASTER_INTERRUPT_4);

   /* Interrupt 5 on the Master PIC is linked to the 8274 and interrupt 6 is
      linked to the Slave PIC. Both these devices supply their own vectors. */

   call SET$INTERRUPT (103, MASTER_INTERRUPT_7);

   /* Define the Slave PIC interrupt handlers                           */

   call SET$INTERRUPT (128, SLAVE_INTERRUPT_0);
   call SET$INTERRUPT (129, SLAVE_INTERRUPT_1);
   call SET$INTERRUPT (130, SLAVE_INTERRUPT_2);
   call SET$INTERRUPT (131, TMAP_INTERRUPT); /* Hardwired to identify TMAP  */
   call SET$INTERRUPT (132, SLAVE_CLOCK); /* For Timer 2 & Slave PIC tests  */
   call SET$INTERRUPT (133, SLAVE_INTERRUPT_5);
   call SET$INTERRUPT (134, SLAVE_INTERRUPT_6);
   call SET$INTERRUPT (135, SLAVE_INTERRUPT_7);

   end SET_INTERRUPT_TABLE;

$ eject
```

```
DELAY_SLAVE_PROCS: procedure;

   /*************************************************************************
    *                                                                       *
    * This procedure applies only to a dual processor environment which has *
    * two CPU cards running identical boot code. One needs to be delayed    *
    * before any off-board writes take place.                               *
    *                                                                       *
    * The secondary processor slot has Multibus interrupt 7 strapped to     *
    * permanently interrupt in order for the firmware to identify which of  *
    * the two identical processor cards is going to be defined as the       *
    * secondary processor.                                                  *
    *                                                                       *
    *************************************************************************/

   disable;
   call OUTPUT_TO_LATCH (DELAYING_SLAVE_PROCS);
   output(PIC_SLAVE_8259A_ADR1)  = SLAVE_ICW1_8259A or LEVEL_TRIGGERED_MODE ;
   output(PIC_SLAVE_8259A_ADR2)  = SLAVE_ICW2_8259A                        ;
   output(PIC_SLAVE_8259A_ADR2)  = SLAVE_ICW3_8259A                        ;
   output(PIC_SLAVE_8259A_ADR2)  = SLAVE_ICW4_8259A                        ;
   output(PIC_SLAVE_8259A_ADR2)  = UNMASK_SLAVE_INT3                       ;
   output(PIC_MASTER_8259A_ADR2) = UNMASK_SLAVE                            ;
   enable;
   call TIME (1);   /* Delay for 100 microseconds to give pending
                       interrupt a chance to interrupt. The interrupt
                       handler will cause the secondary processor to delay
                       for a further 100 milliseconds.                   */

   end DELAY_SLAVE_PROCS;

$ eject
```

```
/*****************************************************************************
 *                                                                           *
 *                          POWER ON SELF TESTS                              *
 *                                                                           *
 * An indication is given below as to whether a test is regarded as critical *
 * or non critical. A critical test failure will cause the processor to shut *
 * down (if possible) and in some cases issue an error message via the RS232 *
 * link. A non critical failure will report its failed status via the RS232  *
 * link, but operation will continue.                                        *
 *                                                                           *
 *****************************************************************************/

POST: procedure ;

   declare SPIC_CAUSED_HALT (*) byte data
      (ESC, '[2J', ESC, '[15;0H', BEL, BEL, BEL,
       '**** CRITICAL ERROR DETECTED ****', CR, LF, LF,
       'SYSTEM HALTED DUE TO SLAVE PIC OR TIMER FAILURE', EOM);

   declare EROM_CAUSED_HALT (*) byte data
      (ESC, '[2J', ESC, '[15;0H', BEL, BEL, BEL,
       '**** CRITICAL ERROR DETECTED ****', CR, LF, LF,
       'SYSTEM HALTED DUE TO INCORRECT CHECKSUM DETECTED ON EPROM/RAM CARD',
        EOM);
   declare EPROM_RAM_NOT_PRESENT (*) byte data
      (ESC, '[2J', ESC, '[15;0H', BEL, BEL, BEL,
       '**** CRITICAL ERROR DETECTED ****', CR, LF, LF,
       'SYSTEM HALTED DUE TO ABSENT EPROM/RAM CARD',
        EOM);
   declare OFF_BOARD_ACCESS_CAUSED_HALT (*) byte data
      (ESC, '[2J', ESC, '[12;0H', BEL, BEL, BEL,
       '**** CRITICAL ERROR DETECTED ****', CR, LF, LF,
       'SYSTEM HALTED DUE TO MAIN PROCESSOR CARD UNABLE TO ACCESS', CR, LF,
       'EPROM/RAM CARD AND SDB CARD.', CR, LF, LF,
       'EITHER BOTH THESE CARDS ARE ABSENT OR THE MAIN PROCESSOR CARD', CR, LF,
       'HAS FAILED WHEN ATTEMPTING OFF BOARD ACCESSES',
        EOM);

   declare ERAM_CAUSED_HALT (*) byte data
      (ESC, '[2J', ESC, '[15;0H', BEL, BEL, BEL,
       '**** CRITICAL ERROR DETECTED ****', CR, LF, LF,
       'SYSTEM HALTED DUE TO RAM FAILURE ON EPROM/RAM CARD',
        EOM);

   declare SDB_RAM_CAUSED_HALT (*) byte data
      (ESC, '[2J', ESC, '[15;0H', BEL, BEL, BEL,
       '**** CRITICAL ERROR DETECTED ****', CR, LF, LF,
       'SYSTEM HALTED DUE TO GLOBAL RAM FAILURE ON SDB CARD',
        EOM);
```

```
   declare I byte;

$ eject

   TEST                          = TRUE     ;
   BIT_RESULT.CPU_BOARD          = UNTESTED ;
   BIT_RESULT.MASTER_PIC         = UNTESTED ;
   BIT_RESULT.SLAVE_PIC          = UNTESTED ;
   BIT_RESULT.TMOUT_CIRCUIT      = UNTESTED ;
   BIT_RESULT.PIT                = UNTESTED ;
   BIT_RESULT.COPROCESSOR        = UNTESTED ;
   BIT_RESULT.MPSC               = UNTESTED ;
   BIT_RESULT.ACCESS_TO_1611     = UNTESTED ;
   BIT_RESULT.ERAM               = UNTESTED ;
   BIT_RESULT.EROM               = UNTESTED ;
   BIT_RESULT.SDB_BOARD          = UNTESTED ;
   BIT_RESULT.SDB_SELFTEST       = UNTESTED ;
   BIT_RESULT.SDB_RAM            = UNTESTED ;
   BIT_RESULT.LOCAL_BUS          = UNTESTED ;

   TMOUT_INTERRUPT               = FALSE ;
   CLOCK_1_INTERRUPT             = FALSE ;
   CLOCK_2_INTERRUPT             = FALSE ;

$ eject

   enable;

   /* First perform all on-board tests */

   call OUTPUT_TO_LATCH (TMOUT_TEST_NO);
   call MPIC_AND_TMOUT_TEST     ; /* Critical test                  */
   if (BIT_RESULT.TMOUT_CIRCUIT = FAILED)
   or (BIT_RESULT.MASTER_PIC) = FAILED then
      halt                           ; /* Master PIC or Time-out failure  */

   call OUTPUT_TO_LATCH (PIT_TEST_NO);
   call MPIC_AND_PIT_TEST       ; /* Critical test                  */
   if BIT_RESULT.PIT = FAILED then
      halt                           ; /* Timer failure                  */

   call OUTPUT_TO_LATCH (MPSC_TEST_NO);
   BIT_RESULT.MPSC = MPSC_IO_TEST (VDU); /* Non critical test              */
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call OUTPUT_TO_LATCH (SLAVE_PIC_TEST_NO);
   call SPIC_MPIC_AND_PIT_TEST  ; /* Critical test                  */
   if BIT_RESULT.MPSC = PASSED then    /* Allow reporting to the RS232    */
      output (PIC_MASTER_8259A_ADR2) = UNMASK_MPSC  ;
```

```
   if (BIT_RESULT.PIT = FAILED)
   or (BIT_RESULT.SLAVE_PIC = FAILED) then
      do                              ;
      if BIT_RESULT.MPSC = PASSED then
         call WRITE_POLL (VDU, @SPIC_CAUSED_HALT);
      halt                            ; /* Slave PIC failure              */
      end                             ;
   call OUTPUT_TO_LATCH (COPROCESSOR_TEST_NO);
   call COPROCESSOR_TEST          ; /* Non critical test               */

$ eject

   call OUTPUT_TO_LATCH (OFF_BOARD_ACCESS_TEST_NO);
   call TEST_ACCESS_TO_1611       ; /* Critical test                   */
   if BIT_RESULT.ACCESS_TO_1611 = FAILED then
      do;
      if BIT_RESULT.MPSC = PASSED then
         do;
         if BIT_RESULT.EROM = NOT_PRESENT then
            call WRITE_POLL (VDU, @EPROM_RAM_NOT_PRESENT);
         else
            call WRITE_POLL (VDU, @OFF_BOARD_ACCESS_CAUSED_HALT);
         end;
      halt;
      end;

   /* Now perform off-board tests                                        */

   call OUTPUT_TO_LATCH (EROM_TEST_NO);
   call APPLICATIONS_ROM_TEST          ; /* Critical test               */
   if BIT_RESULT.EROM = FAILED then
      do                              ;
      if BIT_RESULT.MPSC = PASSED then
         call WRITE_POLL (VDU, @EROM_CAUSED_HALT);
      halt                            ; /* Off board EPROM failure       */
      end                             ;

$ eject
```

```
   if NO_OF_PROCS > 1 then
      do;
      PROCESSOR_ID = MAP            ; /* PROCESSOR_ID is a global flag   */
      call DELAY_SLAVE_PROCS        ; /* that is reset to SLAVE in the   */
      end                           ; /* SLAVE processor interrupt       */
                                      /* handler.                        */

   /*  Any subsystem requiring more than one CPU card should strap a
       permanent interrupt to the Slave PIC (IR3), in order to be able to
       determine whether the processor must delay (i.e. it is a slave,
       provided that the master is operational) or not delay (i.e. it is
       the master).                                                       */

   call OUTPUT_TO_LATCH (ERAM_TEST_NO);
   BIT_RESULT.ERAM = ERAM_TEST      ;    /* Critical test                */
   if BIT_RESULT.ERAM = FAILED then
      do                            ;    /* Global RAM test is critical  */
      if BIT_RESULT.MPSC = PASSED then
         call WRITE_POLL (VDU, @ERAM_CAUSED_HALT);
      halt                          ;
      end;
   call OUTPUT_TO_LATCH (SDB_SELFTEST_TEST_NO);
   call SDB_SELFTEST                ;    /* Non critical test            */
   if BIT_RESULT.SDB_BOARD <> ABSENT then
      do;
      call OUTPUT_TO_LATCH (SDB_RAM_TEST_NO);
      call SDB_RAM_TEST             ;
      end;

   call OUTPUT_TO_LATCH (LOCAL_BUS_TEST_NO);
   call LOCAL_BUS_TEST              ;    /* Non critical test            */
   call OUTPUT_TO_LATCH (RESET_LATCH);

   TEST = FALSE                     ;   /* Reset variable to indicate   */
                                        /* that POST is complete        */
   end POST;

$ eject
```

```
ANALYSE_BIT_RESULTS: procedure;

/*****************************************************************************
 *                                                                           *
 * ANALYSE_BIT_RESULTS simply records the overall results of the POST for    *
 * each board in the standard computing segment. If any functional unit of a *
 * board has failed, then the result is that the board itself has failed.    *
 *                                                                           *
 *****************************************************************************/

   if (BIT_RESULT.CPU = PASSED) and (BIT_RESULT.ROM            = PASSED)
                                and (BIT_RESULT.MASTER_PIC     = PASSED)
                                and (BIT_RESULT.SLAVE_PIC      = PASSED)
                                and (BIT_RESULT.TMOUT_CIRCUIT  = PASSED)
                                and (BIT_RESULT.PIT            = PASSED)
                                and (BIT_RESULT.COPROCESSOR    = PASSED)
                                and (BIT_RESULT.MPSC           = PASSED)
                                and (BIT_RESULT.RAM            = PASSED) then
       BIT_RESULT.CPU_BOARD = PASSED;
   else
      BIT_RESULT.CPU_BOARD = FAILED ;

   if (BIT_RESULT.SDB_SELFTEST = PASSED) and (BIT_RESULT.SDB_RAM = PASSED) then
      BIT_RESULT.SDB_BOARD = PASSED ;
   else
      BIT_RESULT.SDB_BOARD = FAILED ;

   end ANALYSE_BIT_RESULTS ;

$ eject
```

```
PRINT_RESULTS: procedure public;

/*****************************************************************************
 *                                                                           *
 * This procedures reports the results of the POST to the RS232 link. Based  *
 * on the values supplied on the ROM of the EPROM/RAM card, it also          *
 * determines what subsystem is resident on the EPROM/RAM card, as well as   *
 * the version number of that sub system's code, and reports this down the   *
 * RS232.                                                                    *
 *                                                                           *
 *****************************************************************************/

   declare ERASE_SCREEN (*) byte data
      /* Clears the screen and homes the cursor */
      (ESC, '[2J', ESC, '[0;0H', EOM) ;

   declare BIT_MSG (*) byte data
      (BEL,
       '***** BUILT IN TEST RESULTS FOR THE STANDARD COMPUTING SEGMENT *****',
       CR, LF, LF, LF, EOM);

   declare STD_SYSTEM_PASSED (*) byte data
      ('STANDARD COMPUTING SEGMENT...................PASSED',
       CR, LF, LF, EOM);

   declare CPU_BOARD_PASSED (*) byte data
      ('HECS 1442 CPU board.........................PASSED', CR, LF, EOM);
   declare EROM_RAM_PASSED (*) byte data
      ('EPROM/RAM 1611 card.........................PASSED', CR, LF, EOM);
   declare SDB_PASSED (*) byte data
      ('System Data Bus 6123 Controller card.........PASSED', CR, LF, EOM);

   declare STD_SYSTEM_FAILED (*) byte data
      ('STANDARD COMPUTING SEGMENT...................FAILED        ******',
       CR, LF, LF, EOM);

   declare CPU_BOARD_FAILED (*) byte data
      ('HECS 1442 CPU board.........................FAILED        ******',
       CR, LF, EOM);

   declare EROM_RAM_FAILED (*) byte data
      ('EPROM/RAM 1611 card.........................FAILED        ******',
       CR, LF, EOM);

   declare SDB_FAILED (*) byte data
      ('6123 Controller card........................FAILED        ******',
       CR, LF, EOM);
```

```
declare SYSTEM_PROMPT (*) byte data
   (CR, LF, LF,
    'System found on EPROM/RAM card :      ', EOM);

declare VERSION_NO_PROMPT (*) byte data
   (CR, LF, LF,
    'Version no :                          ', EOM);

declare EPROM_CHECKSUM_1442_PROMPT (*) byte data
   (CR, LF, LF,
    'CPU card EPROM CHECKSUM =             ', EOM);

declare EPROM_CHECKSUM_1611_PROMPT (*) byte data
   (CR, LF, LF,
    'EPROM/RAM card EPROM CHECKSUM =       ', EOM);

declare VERSION (4) byte;
declare SYSTEM  (4) byte;
declare EPROM_CHECKSUM_1442 (6) byte;
declare EPROM_CHECKSUM_1611 (6) byte;

$ eject

   output (PIC_MASTER_8259A_ADR2) = UNMASK_MPSC ;
   call WRITE_POLL (VDU, @ERASE_SCREEN) ;
   call WRITE_POLL (VDU, @BIT_MSG);

   if (BIT_RESULT.CPU_BOARD = PASSED) and (BIT_RESULT.ERAM      = PASSED)
                                      and (BIT_RESULT.EROM      = PASSED)
                                      and (BIT_RESULT.SDB_BOARD = PASSED) then
      call WRITE_POLL (VDU, @STD_SYSTEM_PASSED);
   else
      call WRITE_POLL (VDU, @STD_SYSTEM_FAILED);

   if BIT_RESULT.CPU_BOARD = PASSED then
      call WRITE_POLL (VDU, @CPU_BOARD_PASSED);
   else
      call WRITE_POLL (VDU, @CPU_BOARD_FAILED);

   if (BIT_RESULT.ERAM) and (BIT_RESULT.EROM = PASSED) then
      call WRITE_POLL (VDU, @EROM_RAM_PASSED);
   else
      call WRITE_POLL (VDU, @EROM_RAM_FAILED);

   if BIT_RESULT.SDB_BOARD = PASSED then
      call WRITE_POLL (VDU, @SDB_PASSED);
   else
      call WRITE_POLL (VDU, @SDB_FAILED);
```

```
   if (SYSTEM_NUMBER < 1) or (SYSTEM_NUMBER > 9) then
      call PRINT_ERROR;  /* incorrect initialisation of system description */
   else
      do;
      SYSTEM  (3) = EOM;
      call WRITE_POLL (VDU, @SYSTEM_PROMPT);
      call C040_BIN_BYTE_TO_ASCII_DEC (SYSTEM_NUMBER, @SYSTEM);
      call WRITE_POLL (VDU, @SYSTEM);

      VERSION (3) = EOM;
      call WRITE_POLL (VDU, @VERSION_NO_PROMPT);
      call C040_BIN_BYTE_TO_ASCII_DEC (SYSTEM_VERSION, @VERSION);
      call WRITE_POLL (VDU, @VERSION);

      EPROM_CHECKSUM_1442 (5) = EOM;
      call WRITE_POLL (VDU, @EPROM_CHECKSUM_1442_PROMPT);
      call C070_CONVERT_WORD_TO_HEX (OB_EPROM_CHECK, @EPROM_CHECKSUM_1442);
      call WRITE_POLL (VDU, @EPROM_CHECKSUM_1442);

      EPROM_CHECKSUM_1611 (5) = EOM;
      call WRITE_POLL (VDU, @EPROM_CHECKSUM_1611_PROMPT);
      call C070_CONVERT_WORD_TO_HEX (EPROM_CHECK, @EPROM_CHECKSUM_1611);
      call WRITE_POLL (VDU, @EPROM_CHECKSUM_1611);

      end;

   end PRINT_RESULTS ;

$ eject
```

```
INITIALISE_ALL: procedure public;

/*****************************************************************************
 *                                                                           *
 * This is the main program of the initialisation code. It firsts performs   *
 * CPU, stack, ROM and on-board RAM tests. If any of these fail, the program *
 * immediately halts, otherwise it continues to:                             *
 *                                                                           *
 * a) Initialise the Programmable Interval Timer                             *
 * b) Initialise the Programmable Interrupt Controller                       *
 * c) Initialise the Multi Protocol Serial Communications Chip               *
 * d) Initialise the Numeric Coprocessor                                     *
 * e) Perform the Power On Self Tests (POST). Any test defined to be critical *
 *    will cause the processor to halt, and a predefined error will be sent  *
 *    to the diagnostic latch and an error message will be sent to the RS232 *
 *    link (if possible).                                                    *
 * f) Record the results of the POST                                         *
 * g) Output the POST results via the RS232 link, if possible.               *
 *                                                                           *
 *****************************************************************************/

   call TEST_CORE                    ; /* CPU, stack, ROM & RAM tests       */

   call OUTPUT_TO_LATCH (INITIALISATION) ;
   call INITIALISE_8254_PIT          ; /* Initialises 8254 PIT              */
   call INITIALISE_8259A_PIC         ; /* Initialises 8259A PIC             */
   call INITIALISE_PORT_DATA         ; /* Initialises 8274 MPSC             */
   call SET_INTERRUPT_TABLE          ; /* Initialises Interrupt Vector Table */
   call INIT$REAL$MATH$UNIT          ; /* Initialises 80287                 */
   call SET$REAL$MODE (CTRL_287)     ; /* Numeric Coprocessor               */

   if NUMBER_OF_RUNS <> SECOND_RUN then
      do;
      call POST                      ; /* Standard computing segment BIT    */
      call ANALYSE_BIT_RESULTS       ; /* Summary of BIT results            */
      if BIT_RESULT.MPSC = PASSED then
         call PRINT_RESULTS          ; /* The status of each board is output */
                                       /* to the RS232.                     */
      end;
   else
      call DELAY_SLAVE_PROCS;

   output (PIC_MASTER_8259A_ADR2) = UNMASK_MPSC and UNMASK_TMOUT ;
   if NUMBER_OF_RUNS = SECOND_RUN then
      goto REAL_MODE_CODE        ; /* Warm start */

   end INITIALISE_ALL ;

end INIT ;
```

2.5.1.3 STDBIT.PLM

```
$ include (PLMPAR.INC)

/*****************************************************************************
 *                                                                           *
 *       MODULE NAME : BIT                                                   *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Source Filename  : STDBIT.PLM                                           *
 *                                                                           *
 *   Source Compiler  : PLM86                                                *
 *                                                                           *
 *   Operating System : DOS 3.10                                             *
 *                                                                           *
 *   Description      : Diagnostic routines for the                          *
 *                      standard computing segment                           *
 *                                                                           *
 *   Public procedures: CRC_RESULT                                           *
 *                      TEST_ROM_BLOCK                                       *
 *                      PRINT_ERROR                                          *
 *                      OFF_BOARD_ACCESS                                     *
 *                      TEST_CORE                                            *
 *                      ON_BOARD_EPROM_TEST                                  *
 *                      MPIC_AND_TMOUT_TEST                                  *
 *                      MPIC_AND_PIT_TEST                                    *
 *                      SPIC_MPIC_AND_PIT_TEST                               *
 *                      COPROCESSOR_TEST                                     *
 *                      TEST_ACCESS_TO_1611                                  *
 *                      APPLICATIONS_ROM_TEST                                *
 *                      SDB_SELFTEST                                         *
 *                      SDB_RAM_TEST                                         *
 *                      LOCAL_BUS_TEST                                       *
 *                                                                           *
 *   EPD files        : STDIO.EPD                                            *
 *                      STDSTAT.EPD                                          *
 *                      STDPIC.EPD                                           *
 *                      STDPIT.EPD                                           *
 *                      STDINTS.EPD                                          *
 *                      STDCPU.EPD                                           *
 *                      STDRAM.EPD                                           *
 *                      STDERAM.EPD                                          *
 *                      STDSDB.EPD                                           *
 *                                                                           *
 *   Include files    : PLMPAR.INC                                           *
 *                      LITS.INC                                             *
 *                                                                           *
 *****************************************************************************
```

```
 ****************************************************************************
 *                                                                          *
 *   HISTORY   Version 1.0 :                                                *
 *                                                                          *
 *             Designed by : P.A. OLANDER    Date :  August 1989          *
 *             Description : Original                                       *
 *                                                                          *
 *                                                                          *
 ****************************************************************************/

$ eject

BIT: do;

$ include (STDIO.EPD)
$ eject
$ include (LITS.INC)
$ eject
$ include (STDSTAT.EPD)
$ eject
$ include (STDPIC.EPD)
$ eject
$ include (STDPIT.EPD)
$ eject
$ include (STDINTS.EPD)
$ eject
$ include (STDCPU.EPD)
$ eject
$ include (STDRAM.EPD)
$ eject
$ include (STDERAM.EPD)
$ eject
$ include (STDSDB.EPD)
$ eject
```

```
/*****************************************************************************
 *                                                                           *
 *                          Public variables                                 *
 *                                                                           *
 *****************************************************************************/

declare CARD_UNDER_TEST       byte public;
declare ON_BOARD_VAR          word public;
declare OB_EPROM_CHECK        word public;
declare EPROM_CHECK           word public;

declare TEST                  byte public;

/*****************************************************************************
 *                                                                           *
 * This flag is set when tests are executed, so that the processor can       *
 * expect self-generated time-outs. These time-outs are necessary in order   *
 * to test the functionality of certain aspects of the hardware.             *
 *                                                                           *
 *****************************************************************************/

/*****************************************************************************
 *                                                                           *
 *                          Global variables                                 *
 *                                                                           *
 *****************************************************************************/

declare RETRIES               byte;

$ eject
```

```
/*****************************************************************************
 *                                                                           *
 *                          COMMON PROCEDURES                                *
 *                                                                           *
 *****************************************************************************/

CRC_RESULT: procedure (REM, INPUT_BYTE) word public;

   declare REM          word;
   declare INPUT_BYTE   byte;
   declare J            byte;

   do J = 1 to 8;
      INPUT_BYTE = scl(INPUT_BYTE, 1);
      REM        = scl(REM,1);
      if CARRY then
         REM = (REM xor CRC_POLYNOMIAL);
      end;

   return REM;

   end CRC_RESULT;
```

```
TEST_ROM_BLOCK: procedure (ROM_BLOCK_NO) word public;
   declare ROM_BLOCK_NO byte;
   declare REMAINDER word;
   declare I word;

   REMAINDER = 0;
   if (ROM_BLOCK_NO <> 8) and (ROM_BLOCK_NO <> 10) then
      do I = 0 to SEGMENT_MAX;
         do case ROM_BLOCK_NO - 1;
            REMAINDER = CRC_RESULT (REMAINDER, ROM_1(I));
            REMAINDER = CRC_RESULT (REMAINDER, ROM_2(I));
            REMAINDER = CRC_RESULT (REMAINDER, ROM_3(I));
            REMAINDER = CRC_RESULT (REMAINDER, ROM_4(I));
            REMAINDER = CRC_RESULT (REMAINDER, ROM_5(I));
            REMAINDER = CRC_RESULT (REMAINDER, ROM_6(I));
            REMAINDER = CRC_RESULT (REMAINDER, ROM_7(I));
            ; /* ROM_BLOCK = 8 is a special case */
            REMAINDER = CRC_RESULT (REMAINDER, ROM_9(I));
            end; /* case */
         end; /* do loop */
   else
     do I = 0 to SEGMENT_MAX_LESS_2 ;
        /* Both ROM_BLOCK = 8 and ROM_BLOCK = 10 are special cases
           due to the checksum residing at the top of EPROM */
        if ROM_BLOCK_NO = 8 then
           REMAINDER = CRC_RESULT (REMAINDER, ROM_8(I));
        else
           REMAINDER = CRC_RESULT (REMAINDER, ROM_10(I));
         end;

   return REMAINDER;

   end TEST_ROM_BLOCK;

$ eject
```

```
PRINT_ERROR: procedure public;

/* This procedure will output the following message if the 1611 EPROM/RAM card
   does not contain the required information in ROM and will then cause the
   processor to halt.                                                        */

   declare ERROR_MSG (*) byte data
      (ESC, '[2J', ESC, '[0;0H',
       'The system description has not been initialised correctly    ', CR, LF,
       'The following information should be provided on the EPROMS   ', CR, LF,
       'of the EPROM/RAM card by the system:                         ', CR, LF,
                                                                        LF,
       'ADDRESS DESCRIPTION OF VARIABLE   POSSIBLE VALUES            ', CR, LF,
       '------- -----------------------   ---------------            ', CR, LF,
       'dff00h  EPROM ID                  5ah                        ', CR, LF,
       'dff01h  System Number             Defined as per system      ', CR, LF,
       'dff02h  System Version            Defined as per system      ', CR, LF,
       'dff03h  EPROM/RAM option          1 = 512K EPROM, 128K RAM   ', CR, LF,
       '                                  2 = 384K EPROM, 256K RAM   ', CR, LF,
       '                                  3 = 256K EPROM, 384K RAM   ', CR, LF,
       'dff04h  Checksums Present         1 = EPROM checksums present', CR, LF,
       'dff05h  Number of 1442 CPU cards  Defined as per system      ', CR, LF,
       'dff06h  Real or Protected mode      5 = Real mode            ', CR, LF,
       '                                  0ah = Protected mode       ', CR, LF,
       'dff07h  Descriptors tables to     1 = Copy descriptors       ', CR, LF,
       '        be copied                 dff07h <> 1 = Bypass copy  ', CR, LF,
       'dff08h  Real mode link address    IP value                   ', CR, LF,
       'dff0ah  Real mode link address    CS value                   ', CR, LF,
       'dfffeh  EPROM checksum            Code dependent             ', CR, LF,
                                                                        LF,
       'Please initialise the above values, reset the system, and restart ...',
                                                                      EOM);

   if BIT_RESULT.MPSC = PASSED then
      call WRITE_POLL (VDU, @ERROR_MSG);

   call OUTPUT_TO_LATCH (EROM_TEST_NO) ;

   halt;   /* halt the processor if the system initialisation is not correct */

   end PRINT_ERROR;

$ eject
```

```
OFF_BOARD_ACCESS: procedure (ACTION, OFF_BOARD_VAR_PTR, VALUE_TO_WRITE) public;

 /*****************************************************************************
 *                                                                           *
 * This procedure attempts off-board read/writes at a location passed as a   *
 * parameter. If the CPU card times-out when attempting this access and      *
 * the bus had not been granted to the card, then it attempts to access the  *
 * location again. On time-outs with no BUS_GRANT, it will keep trying to    *
 * access the location until the pre-defined maximum number of retries is    *
 * exceeded. In the event of the bus request never being granted, the        *
 * processor reports its inability to be granted the bus and halts.          *
 *                                                                           *
 *****************************************************************************/

   declare ACTION                                  word ;
   declare VALUE_TO_WRITE                          word ;
   declare OFF_BOARD_VAR_PTR                     pointer ;
   declare OFF_BOARD_VAR based OFF_BOARD_VAR_PTR   word ;

   declare UNABLE_TO_ACCESS_BUS (*) byte data
      (ESC, '[2J', ESC,'[15;0H', BEL, BEL, BEL,
       'Critical error detected : ', CR, LF, LF,
       'System halted due to the CPU card not being granted the bus.',
       CR, LF, LF, EOM);
```

```
TMOUT_INTERRUPT = FALSE;
BUS_GRANT = BUS_NOT_GRANTED;
RETRIES = 0;

call UNMASK_PIC (TMOUT_INTERRUPT_NO) ;

if ACTION = WRITE_IT then
   OFF_BOARD_VAR = VALUE_TO_WRITE;
else
   ON_BOARD_VAR = OFF_BOARD_VAR;

do while (TMOUT_INTERRUPT = TRUE)
   and   (BUS_GRANT       = BUS_NOT_GRANTED)
   and   (RETRIES <= MAX_RETRIES) ;

   TMOUT_INTERRUPT= FALSE;             /* Reset flag */

   /* Try another off-board access  */

   if ACTION = WRITE_IT then
      OFF_BOARD_VAR = VALUE_TO_WRITE;
   else
      ON_BOARD_VAR = OFF_BOARD_VAR;

   RETRIES = RETRIES + 1;

   end;

CARD_UNDER_TEST   = PRESENT;     /* Assume success until proven otherwise */

if RETRIES > MAX_RETRIES then    /* Something is holding the bus */
   do;
   call OUTPUT_TO_LATCH (MULTIBUS_TEST_NO);
   call WRITE_POLL (VDU, @UNABLE_TO_ACCESS_BUS);
   halt;
   end;
else
   if TMOUT_INTERRUPT = TRUE then
      /* The bus had been granted eventually, but the
         card under test is absent from the system.   */
      CARD_UNDER_TEST = ABSENT;

end OFF_BOARD_ACCESS;

$ eject
```

```
TEST_CORE: procedure public;

/*****************************************************************************
 *                                                                           *
 * Tests the basic operating kernel on the processor board in order          *
 * to provide operator confidence in the system. If any of these tests fail, *
 * it is considered to be a critical failure. The tests for the basic kernel *
 * are a CPU and stack test, an on-board ROM test and an on_board RAM test.  *
 *                                                                           *
 * The philosophy here is somewhat a "chicken and egg" situation, because if *
 * the CPU or stack is faulty, or if this part of the ROM is corrupt, the    *
 * code is unlikely to have executed thus far anyway. In the execution of    *
 * self-tests, however, it is imperative to assume the functionality of some *
 * fundamental core of the hardware under test.                              *
 *                                                                           *
 *****************************************************************************/

  call OUTPUT_TO_LATCH (CPU_TEST_NO)     ; /* Diagnostic latch initialised */
  BIT_RESULT.CPU = CPU_TEST              ; /* Invoke CPU test function     */
  if BIT_RESULT.CPU = FAILED then
     halt                                ; /* CPU or stack failure         */

  call OUTPUT_TO_LATCH (ROM_TEST_NO)     ;
  BIT_RESULT.ROM = UNTESTED              ;
  call ON_BOARD_EPROM_TEST               ;
  if BIT_RESULT.ROM = FAILED then
     halt                                ; /* On board ROM failure         */

  call OUTPUT_TO_LATCH (RAM_TEST_NO) ;
  BIT_RESULT.RAM        =  MEM_TEST      ;
  if BIT_RESULT.RAM = FAILED then         /* On board RAM failure         */
     halt;

  end TEST_CORE;
```

```
ON_BOARD_EPROM_TEST: procedure public;

   /*************************************************************************
    *                                                                       *
    * This procedure calculates a ROM checksum according to the following   *
    * method ( ref. "COMPUTER NETWORKS", A. TANENBAUM, p. 128 - 132 ) :     *
    *                                                                       *
    * It considers the ROM based code to be one binary number               *
    * and then by using the CRC_CCITT polynomial X**16 + X**12 + X**5 + 1   *
    * (i.e. 1 0001 0000 0010 0001 binary) as a divisor, calculates the      *
    * remainder. This remainder is then defined to be the checksum.         *
    *                                                                       *
    * So EPROM CODE / POLYNOMIAL = DIVIDEND remainder CHECKSUM              *
    *                                                                       *
    *************************************************************************/

   declare ROM_BLOCK_NO   byte;

   OB_EPROM_CHECK = 0;

   do ROM_BLOCK_NO = 9 to 10;
      OB_EPROM_CHECK = OB_EPROM_CHECK + TEST_ROM_BLOCK (ROM_BLOCK_NO);
      end;
   if OB_CSUM_PRESENT = PRESENT then
      do;
      if OB_EPROM_CHECK = OB_EPROM_CHECKSUM then
         BIT_RESULT.ROM = PASSED;
      else
         BIT_RESULT.ROM = FAILED;
      end;

   end ON_BOARD_EPROM_TEST;

$ eject
```

```
MPIC_AND_TMOUT_TEST: procedure public;

   /*************************************************************************
    *                                                                       *
    * Tests the Master PIC and time-out circuitry by performing a write to  *
    * an invalid port. The result should be to cause a time-out interrupt   *
    * to be generated.                                                      *
    *                                                                       *
    *************************************************************************/

   TMOUT_INTERRUPT = FALSE;
   BUS_GRANT       = BUS_NOT_GRANTED;
   RETRIES         = 0;
   call UNMASK_PIC (TMOUT_INTERRUPT_NO)  ;
   output (INVALID_PORT)          = ZERO_ONES     ;

   do while (TMOUT_INTERRUPT = TRUE)
      and   (BUS_GRANT = BUS_NOT_GRANTED)
      and   (RETRIES <= MAX_RETRIES);
   /* Keep trying while something else is holding the Multibus until the number
      of retries exceeds the predefined maximum */

      output (INVALID_PORT)          = ZERO_ONES     ;
      RETRIES = RETRIES + 1;
      end; /* while */

   if TMOUT_INTERRUPT = TRUE then
      do;
      BIT_RESULT.TMOUT_CIRCUIT = PASSED ;
      BIT_RESULT.MASTER_PIC    = PASSED ;
      TMOUT_INTERRUPT          = FALSE  ;   /* Reset flag */
      end;
   else /* Master PIC or time-out circuitry has failed */
      do;
      BIT_RESULT.MASTER_PIC = FAILED ;
      call MPIC_AND_PIT_TEST       ;

      /* This tests the Master PIC in another manner so therefore the code
         can determine whether the Master PIC or the time-out failed */

      if BIT_RESULT.MASTER_PIC = PASSED then
         do;
         BIT_RESULT.TMOUT_CIRCUIT = FAILED ;
         call OUTPUT_TO_LATCH (TMOUT_TEST_NO);
         end;
      else
         call OUTPUT_TO_LATCH (MASTER_PIC_TEST_NO);
      end;

   end MPIC_AND_TMOUT_TEST;
```

```
$ eject

MPIC_AND_PIT_TEST: procedure public;

   /*************************************************************************
    *                                                                       *
    * Tests the Master PIC and timer 0 by enabling timer 0 and then delaying *
    * long enough for timer 0 to cause an interrupt.                        *
    *                                                                       *
    *************************************************************************/

   call UNMASK_PIC (CLOCK_1_INTERRUPT_NO);
   call TIME (110) ; /* Delay for 11 ms, waiting for PIT OUT0 to interrupt */
   call MASK_PIC (CLOCK_1_INTERRUPT_NO)  ;
   if CLOCK_1_INTERRUPT = TRUE then
      do;
      BIT_RESULT.PIT        = PASSED ;
      BIT_RESULT.MASTER_PIC = PASSED ;
      end;
   else
      do;
      BIT_RESULT.PIT        = FAILED ;
      call OUTPUT_TO_LATCH (PIT_TEST_NO);
      end;

   end MPIC_AND_PIT_TEST;

$ eject
```

```
SPIC_MPIC_AND_PIT_TEST: procedure public;

   /*************************************************************************
    *                                                                       *
    * Tests the Slave PIC and timer 1 by enabling timer 1 and then delaying *
    * long enough for timer 1 to cause an interrupt.                        *
    *                                                                       *
    *************************************************************************/

   call UNMASK_PIC (SLAVE_INTERRUPT_NO)     ;
   call UNMASK_PIC (CLOCK_2_INTERRUPT_NO)   ;
   call TIME (150) ; /* Delay for 15 ms, waiting for PIT OUT1 to interrupt  */

   /* Both the master and slave interrupts are masked immediately
      on entry to the interrupt handler. The interrupt handler also
      sends a specific end of interrupt to the Slave PIC.               */

   if CLOCK_2_INTERRUPT = TRUE then
      do;
      BIT_RESULT.SLAVE_PIC  = PASSED ;
      BIT_RESULT.MASTER_PIC = PASSED ;
      end;
   else
      do;
      BIT_RESULT.SLAVE_PIC  = FAILED ;
      BIT_RESULT.PIT        = FAILED ;
      call OUTPUT_TO_LATCH (SLAVE_PIC_TEST_NO);
      end;

   end SPIC_MPIC_AND_PIT_TEST;

$ eject
```

```
COPROCESSOR_TEST: procedure public;

   /*************************************************************************
    *                                                                       *
    * Tests the 80287 Numeric Coprocessor by performing a real number       *
    * calculation with a known result and then checking whether the actual  *
    * result is within an acceptable range when compared to the expected    *
    * result.                                                               *
    *                                                                       *
    *************************************************************************/

   declare (A,B,C,D,E) real;
   declare (ACTUAL_RESULT, EXPECTED_RESULT) real;

   A = 123.123                      ;
   B = 234.234                      ;
   C = 345.345                      ;
   D = 456.456                      ;
   E = 567.567                      ;
   EXPECTED_RESULT = 2166.352116       ;
   ACTUAL_RESULT   = 10.0*((((A+B)*C)-D)/E)      ;

   if (ACTUAL_RESULT - EXPECTED_RESULT < 0.0004)  then
      BIT_RESULT.COPROCESSOR = PASSED ;
   else
      do;
      BIT_RESULT.COPROCESSOR = FAILED ;
      call OUTPUT_TO_LATCH (COPROCESSOR_TEST_NO);
      end;

   end COPROCESSOR_TEST;

$ eject
```

```
TEST_ACCESS_TO_1611: procedure public;

   /*****************************************************************************
    *                                                                           *
    * This procedure tests accesses to the EPROM/RAM card by reading a          *
    * location on this card. If a time-out is generated (this implies that      *
    * either the EPROM/RAM card is not present, or the CPU card is unable       *
    * to access an off-board location), the routine then attempts to read a     *
    * value from the SDB card (i.e. a Multibus read). If a time-out is          *
    * generated again, then the routine assumes that the CPU card is unable     *
    * to access an off-board location, rather than that both the SDB and the    *
    * EPROM/RAM card are absent.                                                *
    *                                                                           *
    *****************************************************************************/

   declare ON_BOARD_VAR word;

   call OFF_BOARD_ACCESS (READ, @EPROM_CHECKSUM, READ);

   if TMOUT_INTERRUPT = TRUE then    /* Both a Local bus           */
      do;                            /* and Multibus read failed   */
      BIT_RESULT.ACCESS_TO_1611 = FAILED;
      call OFF_BOARD_ACCESS (READ, @SDB_TEST_STATUS, READ);  /* Multibus read */

      if TMOUT_INTERRUPT = TRUE then /* No off board accesses are possible */
         call OUTPUT_TO_LATCH (OFF_BOARD_ACCESS_TEST_NO) ;
      else                           /* EPROM/RAM card not present */
         do;
         call OUTPUT_TO_LATCH (EPROM_RAM_ABSENT) ;
         BIT_RESULT.EROM = NOT_PRESENT;
         BIT_RESULT.ERAM = NOT_PRESENT;
         end;
      end;
   else
      BIT_RESULT.ACCESS_TO_1611 = PASSED;

   end TEST_ACCESS_TO_1611;

$ eject
```

```
APPLICATIONS_ROM_TEST: procedure public;

   /*************************************************************************
    *                                                                       *
    * This procedure calculates a ROM checksum of the applications code     *
    * resident on the EPROM/RAM card in a similar manner to that of the     *
    * on-board checksum calculation.                                        *
    *                                                                       *
    * It considers the ROM based code to be one binary number               *
    * and then by using the CRC_CCITT polynomial X**16 + X**12 + X**5 + 1   *
    * (i.e. 1 0001 0000 0010 0001 binary) as a divisor, calculates the      *
    * remainder. This remainder is then defined to be the checksum.         *
    *                                                                       *
    * Thus, EPROM CODE / POLYNOMIAL = DIVIDEND remainder CHECKSUM           *
    *                                                                       *
    *************************************************************************/

   declare ROM_BLOCK_NO byte;
   declare STARTING_BLOCK byte;
```

```
   EPROM_CHECK = 0;
   if     (EPROM_ID = 5ah)
      and (EPROM_RAM_OPTION > 0) and (EPROM_RAM_OPTION < 4) then
      do;
      do case EPROM_RAM_OPTION - 1;
         /* Determine how large the ROM is */
         STARTING_BLOCK = 1 ;            /* Largest ROM option allowed     */
         STARTING_BLOCK = 3 ;            /* EPROM_RAM_OPTION = 2           */
         STARTING_BLOCK = 5 ;            /* Smallest ROM option allowed    */
         end;

      do ROM_BLOCK_NO = STARTING_BLOCK to 8 ;
         EPROM_CHECK  = EPROM_CHECK + TEST_ROM_BLOCK (ROM_BLOCK_NO) ;
         end;

      if CSUMS_PRESENT = PRESENT then
         do;
         if EPROM_CHECK = EPROM_CHECKSUM then
            BIT_RESULT.EROM = PASSED;
         else
            BIT_RESULT.EROM = FAILED;
         end;    /* if CSUMS_PRESENT = PRESENT */
      end;       /* if     (EPROM_ID = 5ah)
                    and (EPROM_RAM_OPTION > 0) and (EPROM_RAM_OPTION < 4) */
   else          /* If no EPROM_ID has been found or             */
                 /* no EPROM_RAM_OPTION has been identified then */
                 /* system description is incomplete.            */
      call PRINT_ERROR;

   if BIT_RESULT.EROM <> PASSED then
      call OUTPUT_TO_LATCH (EROM_TEST_NO);

   end APPLICATIONS_ROM_TEST;

$ eject
```

```
SDB_SELFTEST: procedure public;

  /*************************************************************************
   *                                                                       *
   * During POST, this procedure checks the self test result reported by   *
   * the SDB card. If called during off-line diagnostics mode, the         *
   * procedure will issue a reset to the SDB card and then delay for 100 ms *
   * before checking the result of the SDB self test.                      *
   *                                                                       *
   *************************************************************************/

  declare ON_BOARD_VAR word;
```

```
   call OFF_BOARD_ACCESS (READ, @SDB_TEST_STATUS, READ);

   if TMOUT_INTERRUPT = TRUE then
      do;
      BIT_RESULT.SDB_BOARD    = ABSENT;
      BIT_RESULT.SDB_SELFTEST = ABSENT;
      BIT_RESULT.SDB_RAM      = ABSENT;
      end;
   else
      do;
      if BIT_RESULT.SDB_SELFTEST = UNTESTED then
         if SDB_TEST_STATUS = 2 then
            BIT_RESULT.SDB_SELFTEST = PASSED ;
         else /* test has either taken too long,
                 failed or reported an invalid state */
            do;
            call OUTPUT_TO_LATCH (SDB_SELFTEST_TEST_NO) ;
            BIT_RESULT.SDB_SELFTEST = FAILED ;
            end;
      else /* this test has already been invoked, so issue a reset to SDB board */
         do;
         outword (SDB_PORT) = SDB_RESET;
         /* SDB self-test should only take about 50 ms, so delay for 100 ms */
         call TIME (1000);

         if SDB_TEST_STATUS <> 2 or SDB_TEST_STATUS = BUSY then
         /* the test has either failed, taken too long
            or reported an invalid state */
            do;
            call OUTPUT_TO_LATCH (SDB_SELFTEST_TEST_NO);
            BIT_RESULT.SDB_SELFTEST = FAILED ;
            end;
         else
            BIT_RESULT.SDB_SELFTEST = PASSED ;
         end;
      end; /* else */

   if BIT_RESULT.SDB_SELFTEST <> PASSED then
      call OUTPUT_TO_LATCH (SDB_SELFTEST_TEST_NO);

   end SDB_SELFTEST;

$ eject
```

```
SDB_RAM_TEST: procedure public;

   /*************************************************************************
    *                                                                       *
    * Tests the SDB Dual Port User RAM by performing an alternate write and *
    * read of 0101 0101 0101 0101 and 1010 1010 1010 1010 binary.           *
    *                                                                       *
    * If a time-out interrupt occurs or what is read is not what was written *
    * then the SDB RAM is reported as having failed.                        *
    *                                                                       *
    *************************************************************************/

   declare I word;

   BIT_RESULT.SDB_RAM = PASSED;   /* Be optimistic */
   I = SDB_RAM_BUFFER_MIN;
   do while (I < SDB_RAM_BUFFER_MAX) and (BIT_RESULT.SDB_RAM <> FAILED) ;
      call OFF_BOARD_ACCESS (WRITE_IT, @SDB_RAM_BUFFER(I), WORD_OF_ZERO_ONES);
      call OFF_BOARD_ACCESS (READ, @SDB_RAM_BUFFER(I), READ);

      if (TMOUT_INTERRUPT = TRUE)
      or (ON_BOARD_VAR <> WORD_OF_ZERO_ONES) then
         do;
         BIT_RESULT.SDB_RAM = FAILED;
         TMOUT_INTERRUPT = FALSE;
         end;
      else
         do;
         SDB_RAM_BUFFER(I) = WORD_OF_ONE_ZEROS ;
         call OFF_BOARD_ACCESS (WRITE_IT, @SDB_RAM_BUFFER(I), WORD_OF_ONE_ZEROS);
         call OFF_BOARD_ACCESS (READ, @SDB_RAM_BUFFER(I), READ);
         if (TMOUT_INTERRUPT = TRUE)
         or (ON_BOARD_VAR <> WORD_OF_ONE_ZEROS) then
            do;
            BIT_RESULT.SDB_RAM = FAILED;
            TMOUT_INTERRUPT = FALSE;
            end;
         I = I + 1;
         end;
      end; /* while */

   if BIT_RESULT.SDB_RAM <> PASSED then
      call OUTPUT_TO_LATCH (SDB_RAM_TEST_NO);

   end SDB_RAM_TEST;

$ eject
```

```
LOCAL_BUS_TEST: procedure public;

  /*************************************************************************
   *                                                                       *
   * This tests the functionality of the local bus. The only way to        *
   * determine via which bus off-board accesses are being performed at run *
   * time in the standard computing segment, is by performing a speed test *
   * over both busses (Multibus runs much slower than the local bus).      *
   *                                                                       *
   * The routine operates as follows:                                      *
   *                                                                       *
   * The timer is enabled and for 10 ms a single location is read from the *
   * 1611 EPROM/RAM card (a local bus read). For each local bus read, a    *
   * counter is incremented. Following this, a single location is read for *
   * 10 ms from the 1553 SDB card (a Multibus read). For each Multibus     *
   * read, a second counter is incremented. The two counter values are then *
   * compared and the test passes if                                       *
   *                                                                       *
   * LOCAL_BUS_COUNTER - MULTIBUS_COUNTER    >    SPEED_THRESHOLD          *
   *                                                                       *
   * where SPEED_THRESHOLD is a predefined value, determined by the CPU    *
   * clock speed and the EPROM/RAM memory device access times.             *
   *                                                                       *
   *************************************************************************/

  declare ON_BOARD_VAR             word;
  declare MULTIBUS_COUNTER  dword;
  declare LOCAL_BUS_COUNTER dword;
```

```
   MULTIBUS_COUNTER  = 0;
   LOCAL_BUS_COUNTER = 0;
   output (PIC_MASTER_8259A_ADR2) =     UNMASK_MPSC
                                    and UNMASK_TMOUT
                                    and UNMASK_CLOCK_1; /* Enable the clock */
   CLOCK_1_INTERRUPT = FALSE;
   do while CLOCK_1_INTERRUPT = FALSE;             /* Synchronise the clock */
      end;

   /* The Master clock interrupt handler will set CLOCK_1_INTERRUPT to true */

   /* The stopwatch has now started */

   CLOCK_1_INTERRUPT = FALSE;
   do while CLOCK_1_INTERRUPT = FALSE;           /* Run the loop for 10 ms */
      ON_BOARD_VAR = EPROM_CHECKSUM;           /* Read a known Local bus location */
      LOCAL_BUS_COUNTER = LOCAL_BUS_COUNTER + 1;
      end;

   /* The loop is exited on the first clock interrupt.   */

   /* The stopwatch has now stopped for the first 10 ms, */
   /* but is counting down for the second 10 ms.         */

   CLOCK_1_INTERRUPT = FALSE;
   do while CLOCK_1_INTERRUPT = FALSE;     /* Run this loop for 10 ms also */
      ON_BOARD_VAR = SDB_TEST_STATUS;           /* Read a known Multibus location */
      MULTIBUS_COUNTER = MULTIBUS_COUNTER + 1;
      end;

   /* Exits on second clock interrupt */

   /* The stopwatch has now stopped */

   output (PIC_MASTER_8259A_ADR2) =     UNMASK_MPSC
                                    and UNMASK_TMOUT ; /* Mask off the clock */
   CLOCK_1_INTERRUPT = FALSE;                   /* Reset the interrupt flag */

   if LOCAL_BUS_COUNTER - MULTIBUS_COUNTER > SPEED_THRESHOLD then
      /* An acceptable threshold for the difference between the busses */
      BIT_RESULT.LOCAL_BUS = PASSED;
   else
      do;
      call OUTPUT_TO_LATCH (LOCAL_BUS_TEST_NO);
      BIT_RESULT.LOCAL_BUS = FAILED;
      end;

   end LOCAL_BUS_TEST;

end BIT;
```

2.5.1.4 STDINTS.PLM

```
$ include (PLMPAR.INC)

/*****************************************************************************
 *                                                                           *
 *        MODULE NAME : INTS                                                 *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Source Filename  : STDINTS.PLM                                          *
 *                                                                           *
 *   Source Compiler  : PLM86                                                *
 *                                                                           *
 *   Operating System : DOS 3.10                                             *
 *                                                                           *
 *   Description      : A suite of interrupt handlers to handle the          *
 *                      Built In Tests on the standardised CPU card.         *
 *                                                                           *
 *****************************************************************************
```

```
*****************************************************************************
*                                                                           *
*   Public procedures: DIVIDE_ERROR                                         *
*                      SINGLE_STEP                                          *
*                      NMI                                                  *
*                      BREAKPOINT                                           *
*                      INTO_OVERFLOW                                        *
*                      BOUND_RANGE                                          *
*                      INVALID_OPCODE                                       *
*                      PROC_EXT_NOT_AVAILABLE                               *
*                      DOUBLE_EXCEPTION                                     *
*                      PROC_EXT_SEGMENT_OVERRUN                             *
*                      INVALID_TASK                                         *
*                      SEGMENT_NOT_PRESENT                                  *
*                      STACK_OVERRUN                                        *
*                      GENERAL_PROTECTION                                   *
*                      NCP_INTERRUPT                                        *
*                      ILLEGAL_INTERRUPT                                    *
*                      TMOUT                                                *
*                      MASTER_CLOCK                                         *
*                      MASTER_INTERRUPT_2                                   *
*                      MASTER_INTERRUPT_3                                   *
*                      MASTER_INTERRUPT_4                                   *
*                      MASTER_INTERRUPT_7                                   *
*                      SLAVE_INTERRUPT_0                                    *
*                      SLAVE_INTERRUPT_1                                    *
*                      SLAVE_INTERRUPT_2                                    *
*                      TMAP_INTERRUPT                                       *
*                      SLAVE_CLOCK                                          *
*                      SLAVE_INTERRUPT_5                                    *
*                      SLAVE_INTERRUPT_6                                    *
*                      SLAVE_INTERRUPT_7                                    *
*                                                                           *
*   EPD files        : STDIO.EPD                                            *
*                      STDSTAT.EPD                                          *
*                      STDPIC.EPD                                           *
*                      STDRAM.EPD                                           *
*                      STDCONVT.EPD                                         *
*                                                                           *
*   Include files    : PLMPAR.INC                                           *
*                      LITS.INC                                             *
*                                                                           *
*****************************************************************************

$ eject
```

```
/*****************************************************************************
 *                                                                           *
 *   HISTORY   Version 1.0 :                                                 *
 *                                                                           *
 *             Designed by : P.A. OLANDER    Date :  July 1989             *
 *             Description : Original                                        *
 *                                                                           *
 *****************************************************************************/
$ eject

INTS:       DO;

declare LOW                 literally '0'           ;
declare TMAP                literally '0ah'         ;
declare BUS_GRANT_MASK      literally '00000010b'   ;

$ include (STDIO.EPD)
$ eject
$ include (STDSTAT.EPD)
$ eject
$ include (STDPIC.EPD)
$ eject
$ include (STDRAM.EPD)
$ eject
$ include (STDCONVT.EPD)
$ eject
$ include (LITS.INC)
$ eject

declare REC_ERR_MESS(*) byte data
        ( ESC, '[18;25H', 'CH.A REC.ERR [F,O,P]:        ', EOM );

/*****************************************************************************
 *                                                                           *
 *                         Global variables                                  *
 *                                                                           *
 *****************************************************************************/

declare TMOUT_INTERRUPT         byte public;
declare TMOUT_STATUS            byte public;
declare BUS_GRANT               byte public;

declare RESERVED_INTERRUPT      byte public;
declare ILLEGAL_INTERRUPT       byte public;
declare CLOCK_1_INTERRUPT       byte public;
declare CLOCK_2_INTERRUPT       byte public;
declare TEST                    byte external;

declare VAR_MESSAGE (80)        byte ;
$ eject
```

```
/*****************************************************************************
 *                                                                           *
 *                        Public interrupt handlers                          *
 *                                                                           *
 *****************************************************************************/

DIVIDE_ERROR: procedure interrupt 0 public;

   declare MSG (*) byte data
      ('Divide by zero detected', CR, LF, EOM);

   RESERVED_INTERRUPT = 0;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end DIVIDE_ERROR;

/*****************************************************************************/

SINGLE_STEP: procedure interrupt 1 public;

   declare MSG (*) byte data
      ('Single step interrupt has occurred', CR, LF, EOM);

   RESERVED_INTERRUPT = 1;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end SINGLE_STEP;

/*****************************************************************************/

NMI: procedure interrupt 2 public;

   declare MSG (*) byte data
      ('Non maskable interrupt has occurred', CR, LF, EOM);

   RESERVED_INTERRUPT = 2;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end NMI;

$ eject
```

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

BREAKPOINT: procedure interrupt 3 public;

   declare MSG (*) byte data
      ('Breakpoint interrupt detected', CR, LF, EOM);

   RESERVED_INTERRUPT = 3;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end BREAKPOINT;

/*****************************************************************************/

INTO_OVERFLOW: procedure interrupt 4 public;

   declare MSG (*) byte data
      ('INTO detected overflow exception', CR, LF, EOM);

   RESERVED_INTERRUPT = 4;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end INTO_OVERFLOW;

/*****************************************************************************/

BOUND_RANGE: procedure interrupt 5 public;

   declare MSG (*) byte data
      ('Bound range exceeded exception has occurred', CR, LF, EOM);

   RESERVED_INTERRUPT = 5;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end BOUND_RANGE;

$ eject
```

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

INVALID_OPCODE: procedure interrupt 6 public;

   declare MSG (*) byte data
      ('Invalid opcode exception has occurred', CR, LF, EOM);

   RESERVED_INTERRUPT = 6;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end INVALID_OPCODE;

/*****************************************************************************/

PROC_EXT_NOT_AVAILABLE: procedure interrupt 7 public;

   declare MSG (*) byte data
      ('Processor extension not available exception has occurred', CR, LF, EOM);

   RESERVED_INTERRUPT = 7;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end PROC_EXT_NOT_AVAILABLE;

/*****************************************************************************/

DOUBLE_EXCEPTION: procedure interrupt 8 public;

   declare MSG (*) byte data
      ('Double exception detected', CR, LF, EOM);

   RESERVED_INTERRUPT = 8;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end DOUBLE_EXCEPTION;

$ eject
```

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

PROC_EXT_SEGMENT_OVERRUN: procedure interrupt 9 public;

   declare MSG (*) byte data
      ('Processor extension segment overrun interrupt has occurred', CR, LF, EOM);

   RESERVED_INTERRUPT = 9;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end PROC_EXT_SEGMENT_OVERRUN;

/*****************************************************************************/

INVALID_TASK: procedure interrupt 10 public;

   declare MSG (*) byte data
      ('Invalid task state segment exception has occurred', CR, LF, EOM);

   RESERVED_INTERRUPT = 10;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end INVALID_TASK;

/*****************************************************************************/

SEGMENT_NOT_PRESENT: procedure interrupt 11 public;

   declare MSG (*) byte data
      ('Segment not present exception has occurred', CR, LF, EOM);

   RESERVED_INTERRUPT = 11;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end SEGMENT_NOT_PRESENT;

$ eject

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

```
STACK_OVERRUN: procedure interrupt 12 public;

   declare MSG (*) byte data
      ('Stack segment overrun exception has occurred', CR, LF, EOM);

   RESERVED_INTERRUPT = 12;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end STACK_OVERRUN;

/*****************************************************************************/

GENERAL_PROTECTION: procedure interrupt 13 public;

   declare MSG (*) byte data
      ('General protection exception has occurred', CR, LF, EOM);

   RESERVED_INTERRUPT = 13;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end GENERAL_PROTECTION;

$ eject

/*****************************************************************************/

NCP_INTERRUPT: procedure interrupt 16 public;

   declare MSG (*) byte data
      ('Numeric Coprocessor interrupt has occurred', CR, LF, EOM);

   RESERVED_INTERRUPT = 16;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end NCP_INTERRUPT;

$ eject
```

```
/*****************************************************************************
 *                                                                           *
 *                      Default Illegal Interrupt :                          *
 *                                                                           *
 * ILLEGAL_INT is a default interrupt handler. The interrupt vector table    *
 * is initialised in such a manner that all interrupts initially use this    *
 * default as their vector. Any interrupts declared thereafter thus          *
 * overwrite the illegal interrupt vector.                                   *
 *                                                                           *
 *****************************************************************************/

ILLEGAL_INT: procedure interrupt 33 public;

   declare MSG (*) byte data
      ('An illegal interrupt has occurred', CR, LF, EOM);

   ILLEGAL_INTERRUPT = TRUE;
   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end ILLEGAL_INT;

$ eject
```

```
/*****************************************************************************
 *                                                                           *
 *                        Interrupts on Master PIC                           *
 *                                                                           *
 *****************************************************************************/

TMOUT: procedure interrupt 96 public;

/* Master interrupt 0 */

   declare HALT_WITH_BUS_GRANT (*) byte data
      (ESC, '[2J', ESC, '[0;0H', BEL, BEL, BEL,
       'Processor has halted due to a MULTIBUS address not responding.',
       CR, LF, LF,
       'The bus HAD been granted to the processor.', EOM);

   declare HALT_WITHOUT_BUS_GRANT (*) byte data
      (ESC, '[2J', ESC, '[0;0H', BEL, BEL, BEL,
       'Processor has halted due to the MULTIBUS being held.',
       CR, LF, LF,
       'The bus HAD NOT been granted to the processor.', EOM);

   TMOUT_STATUS = INPUT_FROM_LATCH;
   BUS_GRANT = TMOUT_STATUS and BUS_GRANT_MASK ;

   if TEST <> TRUE then /* time-out occurred during normal applications */
      do;
      if BUS_GRANT = LOW then /* bus has been granted (active low) */
         do;
         output (STATUS_LATCH_ADR) = BUS_GRANTED ;
         if BIT_RESULT.MPSC = PASSED then
            call WRITE_POLL (VDU, @HALT_WITH_BUS_GRANT);
         end;
      else
         do;
         output (STATUS_LATCH_ADR) = BUS_NOT_GRANTED ;
         if BIT_RESULT.MPSC = PASSED then
            call WRITE_POLL (VDU, @HALT_WITHOUT_BUS_GRANT);
         end;
      halt;
      end;
   else
      TMOUT_INTERRUPT = TRUE;

   end TMOUT;

$ eject

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

```
MASTER_CLOCK: procedure interrupt 97 public;

/* Master interrupt 1 */

/* This procedure is called every 10 ms when OUT0 of the PIT is unmasked     */

   CLOCK_1_INTERRUPT = TRUE ;
   end MASTER_CLOCK;

/*****************************************************************************/

MASTER_INTERRUPT_2: procedure interrupt 98 public;

   declare MSG (*) byte data
      ('Master PIC interrupt 2 has occurred', CR, LF, EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end MASTER_INTERRUPT_2;

MASTER_INTERRUPT_3: procedure interrupt 99 public;

   declare MSG (*) byte data
      ('Master PIC interrupt 3 has occurred', CR, LF, EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end MASTER_INTERRUPT_3;

MASTER_INTERRUPT_4: procedure interrupt 100 public;

   declare MSG (*) byte data
      ('Master PIC interrupt 4 has occurred', CR, LF, EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end MASTER_INTERRUPT_4;
```

```
/* In the standard computing segment, Master PIC interrupts 5 and 6 are   */
/* linked to devices that supply their own vectors, i.e.                  */
/* IR5 : Serial interface ; IR6 : Slave PIC interrupt                     */

MASTER_INTERRUPT_7: procedure interrupt 103 public;

   declare MSG (*) byte data
      ('Master PIC interrupt 7 has occurred', CR, LF, EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   halt;

   end MASTER_INTERRUPT_7;

$ eject

/*****************************************************************************
 *                                                                           *
 *                        Interrupts on Slave PIC                            *
 *                                                                           *
 *****************************************************************************/

SLAVE_INTERRUPT_0: procedure interrupt 128 public;

   declare MSG (*) byte data
      ('SLAVE PIC interrupt 0 has occurred', CR, LF, EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   call ISSUE_EOI (API_INTERRUPT_NO);
   halt;

   end SLAVE_INTERRUPT_0;

SLAVE_INTERRUPT_1: procedure interrupt 129 public;

   declare MSG (*) byte data
      ('SLAVE PIC interrupt 1 has occurred', CR, LF, EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   call ISSUE_EOI (SCMB_INTERRUPT_NO);
   halt;

   end SLAVE_INTERRUPT_1;
```

```
SLAVE_INTERRUPT_2: procedure interrupt 130 public;

   declare MSG (*) byte data
      ('SLAVE PIC interrupt 2 has occurred', CR, LF, EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   call ISSUE_EOI (INTERRUPT_NO_10);
   halt;

   end SLAVE_INTERRUPT_2;

TMAP_INTERRUPT: procedure interrupt 131 public;

 /* This interupt handler is system specific. It caters for a dual
    processor environment in which one of the standardised CPU cards
    is located in a slot which has Multibus interrupt 7 tied low on
    the backplane.

    Since both CPU boards are executing the same code, the handler
    delays for 2 seconds to allow the first processor to get ahead
    and then re-initialises the Slave PIC to accept edge triggered
    interrupts only.                                                  */

   PROCESSOR_ID = TMAP;   /* Set ID to secondary processor            */
   call TIME (20000);
   output (PIC_MASTER_8259A_ADR2) = UNMASK_MPSC;
   output (PIC_SLAVE_8259A_ADR1) = SLAVE_ICW1_8259A;
   output (PIC_SLAVE_8259A_ADR2) = SLAVE_ICW2_8259A;
   output (PIC_SLAVE_8259A_ADR2) = SLAVE_ICW3_8259A;
   output (PIC_SLAVE_8259A_ADR2) = SLAVE_ICW4_8259A;
   output (PIC_SLAVE_8259A_ADR2) = MASK_ALL_INTS;

   end TMAP_INTERRUPT;

$ eject

SLAVE_CLOCK: procedure interrupt 132 public;

/* This procedure is called every 7 microsecs when
   OUT1 of the PIT is unmasked                                        */

   call MASK_PIC (SLAVE_INTERRUPT_NO);
   call MASK_PIC (CLOCK_2_INTERRUPT_NO);
   CLOCK_2_INTERRUPT = TRUE ;
   call ISSUE_EOI (CLOCK_2_INTERRUPT_NO);

   end SLAVE_CLOCK;
```

```
SLAVE_INTERRUPT_5: procedure interrupt 133 public;

   declare MSG (*) byte data
      ('SLAVE PIC interrupt 5 has occurred', CR, LF, EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   call ISSUE_EOI (DRAM_INTERRUPT_NO);
   halt;

   end SLAVE_INTERRUPT_5;

$ eject

SLAVE_INTERRUPT_6: procedure interrupt 134 public;

   declare MSG (*) byte data
      ('SLAVE PIC interrupt 6 has occurred', CR, LF, EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   call ISSUE_EOI (INTERRUPT_NO_13);
   halt;

   end SLAVE_INTERRUPT_6;

SLAVE_INTERRUPT_7: procedure interrupt 135 public;

   declare MSG (*) byte data
      ('SLAVE PIC interrupt 7 has occurred', CR, LF, EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @MSG);
   call ISSUE_EOI (INTERRUPT_NO_14);
   halt;

   end SLAVE_INTERRUPT_7;

end INTS;
```

2.5.1.5 STDPIC.PLM

```
$ include (PLMPAR.INC)

/*****************************************************************************
 *                                                                           *
 *        MODULE NAME : PIC                                                  *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Source Filename  : STDPIC.PLM                                           *
 *                                                                           *
 *   Source Compiler  : PLM86                                                *
 *                                                                           *
 *   Operating System : DOS 3.10                                             *
 *                                                                           *
 *   Description      : Standardised routines for the 8259A                  *
 *                      programmable interrupt controller.                   *
 *                                                                           *
 *   Public procedures: INITIALISE_8259A_PIC                                 *
 *                      MASK_PIC                                             *
 *                      UNMASK_PIC                                           *
 *                      ISSUE_EOI                                            *
 *                                                                           *
 *   EPD files        : None                                                 *
 *                                                                           *
 *   Include files    : PLMPAR.INC                                           *
 *                      PICLITS.INC                                          *
 *                                                                           *
 *****************************************************************************

 $ eject

 *****************************************************************************
 *                                                                           *
 *   HISTORY   Version 1.0 :                                                 *
 *                                                                           *
 *             Designed by : P.A. OLANDER    Date : June 1989              *
 *             Description : Original                                        *
 *                                                                           *
 *                                                                           *
 *****************************************************************************/

 $ eject
```

```
PIC:            do;

$ include (PICLITS.INC)

declare MASTER_INTERRUPT_MASK byte;
declare SLAVE_INTERRUPT_MASK byte;

INITIALISE_8259A_PIC : procedure public;

    /*************************************************************************
    *                                                                        *
    * Initialises the Programmable Interrupt Controller as follows:         *
    *                                                                        *
    * 1) Master PIC                                                         *
    *                                                                        *
    *    i) to trigger on edge transitions                                   *
    *   ii) with an interrupt base of 96 and interval of 8                   *
    *  iii) input line IR6 is linked to the 8274 MPSC and input line IR5 is  *
    *       linked to the slave PIC and both these devices supply their own  *
    *       vectors when providing interrupts                                *
    *   iv) buffered and not special fully nested                            *
    *    v) to provide an automatic end of interrupt (EOI)                   *
    *   vi) to operate in 8086 mode                                          *
    *                                                                        *
    * 2) Slave PIC                                                          *
    *                                                                        *
    *    i) to trigger on edge transitions                                   *
    *   ii) with an interrupt base of 128 and interval of 8                  *
    *  iii) cascaded to input line IR5 of the master PIC                     *
    *   iv) buffered and not special fully nested                            *
    *    v) to expect a normal end of interrupt                              *
    *   vi) to operate in 8086 mode                                          *
    *                                                                        *
    *************************************************************************/
```

```
   /* Master Programmable Interrupt Controller initialisation.              */

   output(PIC_MASTER_8259A_ADR1) = MASTER_ICW1_8259A  ;
   output(PIC_MASTER_8259A_ADR2) = MASTER_ICW2_8259A  ;
   output(PIC_MASTER_8259A_ADR2) = MASTER_ICW3_8259A  ;
   output(PIC_MASTER_8259A_ADR2) = MASTER_ICW4_8259A  ;
   output(PIC_MASTER_8259A_ADR2) = MASK_ALL_INTS      ;
   MASTER_INTERRUPT_MASK         = MASK_ALL_INTS      ;

   /* Initially mask all interrupts */

   /* Slave Programmable Interrupt Controller initialisation  */

   output(PIC_SLAVE_8259A_ADR1)  = SLAVE_ICW1_8259A  ;
   output(PIC_SLAVE_8259A_ADR2)  = SLAVE_ICW2_8259A  ;
   output(PIC_SLAVE_8259A_ADR2)  = SLAVE_ICW3_8259A  ;
   output(PIC_SLAVE_8259A_ADR2)  = SLAVE_ICW4_8259A  ;
   output(PIC_SLAVE_8259A_ADR2)  = MASK_ALL_INTS     ;
   SLAVE_INTERRUPT_MASK          = MASK_ALL_INTS      ;

   /* Initially mask all interrupts */

   end INITIALISE_8259A_PIC ;

$ eject
```

```
UNMASK_PIC: procedure (INTERRUPT_NO) public;

/*****************************************************************************
 *                                                                           *
 * This procedure takes the PIC's current mask status and unmasks additional *
 * interrupts.                                                               *
 *                                                                           *
 *****************************************************************************/

   declare INTERRUPT_NO             byte;
   declare INT_TO_UNMASK            byte;

   if (INTERRUPT_NO >= 0) and (INTERRUPT_NO <= 7) then
      /* it is a master interrupt */
      do;
      INT_TO_UNMASK = rol (UNMASK_TMOUT, INTERRUPT_NO);
      MASTER_INTERRUPT_MASK = MASTER_INTERRUPT_MASK and INT_TO_UNMASK;
      output (PIC_MASTER_8259A_ADR2) = MASTER_INTERRUPT_MASK;
      end;
   else
      do;
      if (INTERRUPT_NO >= 8) and (INTERRUPT_NO <= 15) then
         /* it is a slave interrupt */
         do;
         INTERRUPT_NO = INTERRUPT_NO - 8;
         INT_TO_UNMASK = rol (UNMASK_TMOUT, INTERRUPT_NO);
         SLAVE_INTERRUPT_MASK = SLAVE_INTERRUPT_MASK and INT_TO_UNMASK;
         output (PIC_SLAVE_8259A_ADR2) = SLAVE_INTERRUPT_MASK;
         end;
      end;

   end UNMASK_PIC;

$ eject
```

```
MASK_PIC: procedure (INTERRUPT_NO) public;

/*****************************************************************************
 *                                                                           *
 * This procedure takes the PIC's current mask status and masks additional   *
 * interrupts.                                                               *
 *                                                                           *
 *****************************************************************************/

   declare INTERRUPT_NO             byte;
   declare INT_TO_MASK              byte;

   if (INTERRUPT_NO >= 0) and (INTERRUPT_NO <= 7) then
      /* it is a master interrupt */
      do;
      INT_TO_MASK = rol (MASK_TMOUT, INTERRUPT_NO);
      MASTER_INTERRUPT_MASK = MASTER_INTERRUPT_MASK or INT_TO_MASK;
      output (PIC_MASTER_8259A_ADR2) = MASTER_INTERRUPT_MASK;
      end;
   else
      do;
      if (INTERRUPT_NO >= 8) and (INTERRUPT_NO <= 15) then
         /* it is a slave interrupt */
         do;
         INTERRUPT_NO = INTERRUPT_NO - 8;
         INT_TO_MASK = rol (MASK_TMOUT, INTERRUPT_NO);
         SLAVE_INTERRUPT_MASK = SLAVE_INTERRUPT_MASK or INT_TO_MASK;
         output (PIC_SLAVE_8259A_ADR2) = SLAVE_INTERRUPT_MASK;
         end;
      end;

   end MASK_PIC;
```

```
ISSUE_EOI: procedure (INTERRUPT_NO) public;


/*****************************************************************************
 *                                                                           *
 * This procedure issues an end of interrupt signal to the PIC for the       *
 * appropriate interrupt.                                                    *
 *                                                                           *
 *****************************************************************************/

   declare INTERRUPT_NO byte;
   declare EOI          byte;

   if (INTERRUPT_NO >= 0) and (INTERRUPT_NO <= 7) then
      do;
      if ( (MASTER_ICW4_8259A) and (AUTO_EOI_MASK) )  <> AUTO_EOI_MASK then
         /* the Master PIC has not been configured in auto EOI mode */
         do;
         EOI = EOI_MASK or INTERRUPT_NO;
         output (PIC_MASTER_8259A_ADR2) = EOI;
         end;
      end;
   else
      do;
      if (INTERRUPT_NO >= 8) and (INTERRUPT_NO <= 15) then
         do;
         INTERRUPT_NO = INTERRUPT_NO - 8;
         EOI = EOI_MASK or INTERRUPT_NO;
         output (PIC_SLAVE_8259A_ADR1) = EOI;
         end;
      end;

   end ISSUE_EOI;

end PIC;
```

2.5.1.6 STDPIT.PLM

```
$ include (PLMPAR.INC)

/*****************************************************************************
 *                                                                           *
 *        MODULE NAME : PIT                                                  *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Source Filename  : STDPIT.PLM                                           *
 *                                                                           *
 *   Source Compiler  : PLM86                                                *
 *                                                                           *
 *   Operating System : DOS 3.10                                             *
 *                                                                           *
 *   Description      : Standardised routines for the 8254                   *
 *                      programmable interval timer.                         *
 *                                                                           *
 *   Public procedures: INITIALISE_8254_PIT                                  *
 *                                                                           *
 *   EPD files        : None                                                 *
 *                                                                           *
 *   Include files    : PLMPAR.INC                                           *
 *                      PITLITS.INC                                          *
 *                                                                           *
 *****************************************************************************

$ eject

 *****************************************************************************
 *                                                                           *
 *   HISTORY   Version 1.0 :                                                 *
 *                                                                           *
 *             Designed by : P.A. OLANDER    Date : June 1989              *
 *             Description : Original                                        *
 *                                                                           *
 *                                                                           *
 *****************************************************************************/
$ eject
```

```
PIT:            do;

$ include (PITLITS.INC)

INITIALISE_8254_PIT : procedure public;

    /*************************************************************************
    *                                                                        *
    * Initialises the Programmable Interval Timer to the following values:  *
    *                                                                        *
    * Control registers are initialised to read/write. They accept the       *
    * least significant byte first and then the most significant byte,       *
    * and are all set to act as 16-bit counters.                             *
    *                                                                        *
    * Counter 0 and Counter 1 are initialised to behave as per mode 2,       *
    * i.e. as Rate Generators. In this mode, the timers function like        *
    * divide by N counters, typically used for Real Time Clock interrupts.   *
    * On the Standard CPU card, these two timers are strapped to the         *
    * Master PIC and the Slave PIC, respectively.                            *
    *                                                                        *
    * The initial values loaded into counters 0 and 1 are 3000h or           *
    * 12288 decimal. This will cause the timer to respond every 10 ms,       *
    * since :                                                                *
    *                                                                        *
    * Timer response = Initial counter value / Clock frequency               *
    *                = 12288 / 1.2288 MHz                                    *
    *                = 10 ms                                                 *
    *                                                                        *
    * Counter 2, however, is initialised to behave as per mode 3, i.e. in    *
    * Square Wave Mode. This counter is tied to the 8274 MPSC, which is      *
    * used to drive the RS232 link.                                          *
    *                                                                        *
    * Counter 2 is initialised with a value of 8, since the baud rate        *
    * factor on the MPSC is defined to be a "divide-by-16" clock, thus :     *
    *                                                                        *
    * Counter value  = Clock frequency / (Baud rate * Baud rate factor)      *
    *                = 1.2288 Mhz / (9600 * 16)                              *
    *                = 8                                                     *
    *                                                                        *
    *************************************************************************/
```

```
   /* Load counter 0 initial value */

   output(CTR_CTRL_ADR)  = CTR0_8254_CTRL ;
   output(CTR0_8254_ADR) = CTR0_8254_VAL_LSB ;
   output(CTR0_8254_ADR) = CTR0_8254_VAL_MSB ;

   /* Load counter 1 initial value */

   output(CTR_CTRL_ADR)  = CTR1_8254_CTRL ;
   output(CTR1_8254_ADR) = CTR1_8254_VAL_LSB ;
   output(CTR1_8254_ADR) = CTR1_8254_VAL_MSB ;

   /* Load counter 2 initial value */

   output(CTR_CTRL_ADR)  = CTR2_8254_CTRL ;
   output(CTR2_8254_ADR) = CTR2_8254_VAL_LSB ;
   output(CTR2_8254_ADR) = CTR2_8254_VAL_MSB ;

   end INITIALISE_8254_PIT ;

end PIT;
```

2.5.1.7 STDSTAT.PLM

```
$ include (PLMPAR.INC)

/*****************************************************************************
 *                                                                           *
 *        MODULE NAME : STATUS_LATCH                                         *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Source Filename  : STDSTAT.PLM                                          *
 *                                                                           *
 *   Source Compiler  : PLM86                                                *
 *                                                                           *
 *   Operating System : DOS 3.10                                             *
 *                                                                           *
 *   Description      : Standardised routines to drive the                   *
 *                      diagnostic status latch.                             *
 *                                                                           *
 *   Public procedures: OUTPUT_TO_LATCH                                      *
 *                      INPUT_FROM_LATCH                                     *
 *                                                                           *
 *   EPD files        : None                                                 *
 *                                                                           *
 *   Include files    : PLMPAR.INC                                           *
 *                      STATLITS.INC                                         *
 *                                                                           *
 *****************************************************************************

$ eject

 *****************************************************************************
 *                                                                           *
 *   HISTORY   Version 1.0 :                                                 *
 *                                                                           *
 *             Designed by : P.A. OLANDER    Date : June 1989              *
 *             Description : Original                                        *
 *                                                                           *
 *                                                                           *
 *****************************************************************************/
$ eject
```

```
STATUS_LATCH: do;

$ include (STATLITS.INC)

OUTPUT_TO_LATCH: procedure (TEST_NO) public;

/* Writes the current test number to the diagnostic latch */

   declare TEST_NO byte;

   output (STATUS_LATCH_ADR) = TEST_NO;

   end OUTPUT_TO_LATCH;

$ eject

INPUT_FROM_LATCH: procedure byte public;

/* Reads the status latch in order to determine the time-out status */

   declare TMOUT_STATUS byte;

   TMOUT_STATUS = input (STATUS_LATCH_ADR);
   return TMOUT_STATUS;

   end INPUT_FROM_LATCH;

end STATUS_LATCH;
```

2.5.1.8 STDDIAG.PLM

```
$ include (PLMPAR.INC)

/*****************************************************************************
 *                                                                           *
 *       MODULE NAME : DIAG                                                  *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Source Filename  : STDDIAG.PLM                                          *
 *                                                                           *
 *   Source Compiler  : PLM86                                                *
 *                                                                           *
 *   Operating System : DOS 3.10                                             *
 *                                                                           *
 *   Description      : Off-line diagnostic routines for                     *
 *                      standard computing segment.                          *
 *                                                                           *
 *   Public procedures: DIAGNOSTICS                                          *
 *                                                                           *
 *   EPD files        : STDIO.EPD                                            *
 *                      STDCPU.EPD                                           *
 *                      STDRAM.EPD                                           *
 *                      STDERAM.EPD                                          *
 *                      STDBIT.EPD                                           *
 *                      STDINIT.EPD                                          *
 *                                                                           *
 *   Include files    : PLMPAR.INC                                           *
 *                      LITS.INC                                             *
 *                                                                           *
 *****************************************************************************

$ eject

 *****************************************************************************
 *                                                                           *
 *   HISTORY   Version 1.0 :                                                 *
 *                                                                           *
 *             Designed by : P.A. OLANDER    Date : July 1989              *
 *             Description : Original                                        *
 *                                                                           *
 *                                                                           *
 *****************************************************************************/

$ eject
```

```
/*****************************************************************************
*                                                                            *
* The standardised off-line built in tests are divided into procedures that  *
* perform tests on each of the cards in the standard computing segment. Each *
* card thus has an associated testing routine together with the              *
* corresponding display driving routine.                                     *
*                                                                            *
* The procedures are :                                                       *
*                                                                            *
* PRINT_1442_RESULTS :  Transmits the results of the CPU card tests down the *
*                       RS232 serial port.                                   *
* PRINT_1611_RESULTS :  Transmits the results of the EPROM/RAM card tests    *
*                       down the RS232 serial port.                          *
* PRINT_6123_RESULTS :  Transmits the results of the serial data bus         *
*                       controller card tests down the RS232 serial port.    *
* OFF_LINE_1442_TESTS : Performs off-line BIT on the CPU card.               *
* OFF_LINE_1611_TESTS : Performs off-line BIT on the EPROM/RAM card.         *
* OFF_LINE_6123_TESTS : Performs off-line BIT on the system data bus         *
*                       controller card.                                     *
* DIAGNOSTICS         : Drives the off-line testing of the standard          *
*                       computing segment.                                   *
*                                                                            *
*****************************************************************************/

DIAG: do;

$ include (STDIO.EPD)
$ eject
$ include (LITS.INC)
$ eject
$ include (STDBIT.EPD)
$ eject
$ include (STDCPU.EPD)
$ eject
$ include (STDRAM.EPD)
$ eject
$ include (STDERAM.EPD)
$ eject
$ include (STDINIT.EPD)
$ eject

declare ILLEGAL_INPUT (*) byte data
   (ESC, '[24;0H', BEL, 'ILLEGAL INPUT', EOM);
   declare TEST_NOT_IMPLEMENTED (*) byte data
      (ESC, '[0;50H', BEL, 'TEST NOT IMPLEMENTED', EOM);
   declare SPACES_TOP (*) byte data
      (ESC, '[0;50H', '                    ', EOM);
   declare SPACES_BOTTOM (*) byte data
      (ESC, '[24;0H', '              ', EOM);
```

```
   declare CONTINUE (*) byte data
      (ESC, '[24;0H', '<ESC> returns', EOM);

   declare PASSED_1 (*) byte data
      (ESC, '[1;50H', 'PASSED          ', EOM);
   declare PASSED_2 (*) byte data
      (ESC, '[2;50H', 'PASSED          ', EOM);
   declare PASSED_3 (*) byte data
      (ESC, '[3;50H', 'PASSED          ', EOM);
   declare PASSED_4 (*) byte data
      (ESC, '[4;50H', 'PASSED          ', EOM);
   declare PASSED_5 (*) byte data
      (ESC, '[5;50H', 'PASSED          ', EOM);
   declare PASSED_6 (*) byte data
      (ESC, '[6;50H', 'PASSED          ', EOM);
   declare PASSED_7 (*) byte data
      (ESC, '[7;50H', 'PASSED          ', EOM);
   declare PASSED_8 (*) byte data
      (ESC, '[8;50H', 'PASSED          ', EOM);
   declare PASSED_9 (*) byte data
      (ESC, '[9;50H', 'PASSED          ', EOM);
   declare PASSED_10 (*) byte data
      (ESC, '[10;50H', 'PASSED          ', EOM);
   declare PASSED_11 (*) byte data
      (ESC, '[11;50H', 'PASSED          ', EOM);
   declare PASSED_12 (*) byte data
      (ESC, '[12;50H', 'PASSED          ', EOM);
   declare PASSED_13 (*) byte data
      (ESC, '[13;50H', 'PASSED          ', EOM);
   declare PASSED_14 (*) byte data
      (ESC, '[14;50H', 'PASSED          ', EOM);
   declare PASSED_15 (*) byte data
      (ESC, '[15;50H', 'PASSED          ', EOM);
   declare PASSED_16 (*) byte data
      (ESC, '[16;50H', 'PASSED          ', EOM);
   declare PASSED_17 (*) byte data
      (ESC, '[17;50H', 'PASSED          ', EOM);
   declare PASSED_18 (*) byte data
      (ESC, '[18;50H', 'PASSED          ', EOM);
   declare PASSED_19 (*) byte data
      (ESC, '[19;50H', 'PASSED          ', EOM);
   declare PASSED_20 (*) byte data
      (ESC, '[20;50H', 'PASSED          ', EOM);
   declare PASSED_21 (*) byte data
      (ESC, '[21;50H', 'PASSED          ', EOM);
   declare PASSED_22 (*) byte data
      (ESC, '[22;50H', 'PASSED          ', EOM);

$ eject
```

```
declare FAILED_1 (*) byte data
   (ESC, '[1;50H', '**** FAILED     ', EOM);
declare FAILED_2 (*) byte data
   (ESC, '[2;50H', '**** FAILED     ', EOM);
declare FAILED_3 (*) byte data
   (ESC, '[3;50H', '**** FAILED     ', EOM);
declare FAILED_4 (*) byte data
   (ESC, '[4;50H', '**** FAILED     ', EOM);
declare FAILED_5 (*) byte data
   (ESC, '[5;50H', '**** FAILED     ', EOM);
declare FAILED_6 (*) byte data
   (ESC, '[6;50H', '**** FAILED     ', EOM);
declare FAILED_7 (*) byte data
   (ESC, '[7;50H', '**** FAILED     ', EOM);
declare FAILED_8 (*) byte data
   (ESC, '[8;50H', '**** FAILED     ', EOM);
declare FAILED_9 (*) byte data
   (ESC, '[9;50H', '**** FAILED     ', EOM);
declare FAILED_10 (*) byte data
   (ESC, '[10;50H', '**** FAILED     ', EOM);
declare FAILED_11 (*) byte data
   (ESC, '[11;50H', '**** FAILED     ', EOM);
declare FAILED_12 (*) byte data
   (ESC, '[12;50H', '**** FAILED     ', EOM);
declare FAILED_13 (*) byte data
   (ESC, '[13;50H', '**** FAILED     ', EOM);
declare FAILED_14 (*) byte data
   (ESC, '[14;50H', '**** FAILED     ', EOM);
declare FAILED_15 (*) byte data
   (ESC, '[15;50H', '**** FAILED     ', EOM);
declare FAILED_16 (*) byte data
   (ESC, '[16;50H', '**** FAILED     ', EOM);
declare FAILED_17 (*) byte data
   (ESC, '[17;50H', '**** FAILED     ', EOM);
declare FAILED_18 (*) byte data
   (ESC, '[18;50H', '**** FAILED     ', EOM);
declare FAILED_19 (*) byte data
   (ESC, '[19;50H', '**** FAILED     ', EOM);
declare FAILED_20 (*) byte data
   (ESC, '[20;50H', '**** FAILED     ', EOM);
declare FAILED_21 (*) byte data
   (ESC, '[21;50H', '**** FAILED     ', EOM);
declare FAILED_22 (*) byte data
   (ESC, '[22;50H', '**** FAILED     ', EOM);
```

```
   declare BUSY_3 (*) byte data
      (ESC, '[3;50H', 'BUSY TESTING', EOM);
   declare BUSY_4 (*) byte data
      (ESC, '[4;50H', 'BUSY TESTING', EOM);
   declare BUSY_5 (*) byte data
      (ESC, '[5;50H', 'BUSY TESTING', EOM);
   declare BUSY_6 (*) byte data
      (ESC, '[6;50H', 'BUSY TESTING', EOM);
   declare BUSY_10 (*) byte data
      (ESC, '[10;50H', 'BUSY TESTING', EOM);
   declare BUSY_11 (*) byte data
      (ESC, '[11;50H', 'BUSY TESTING', EOM);
   declare BUSY_12 (*) byte data
      (ESC, '[12;50H', 'BUSY TESTING', EOM);
   declare BUSY_13 (*) byte data
      (ESC, '[13;50H', 'BUSY TESTING', EOM);
   declare BUSY_14 (*) byte data
      (ESC, '[14;50H', 'BUSY TESTING', EOM);
   declare BUSY_16 (*) byte data
      (ESC, '[16;50H', 'BUSY TESTING', EOM);

declare LETTER  byte;

$ eject
```

```
/*****************************************************************************
*                                                                           *
*                        Display driving procedures                         *
*                                                                           *
*****************************************************************************/

PRINT_1442_RESULTS: procedure;
   declare DISPLAY (*) byte data

      ('RESULTS OF HECS 1442 CPU BOARD BIT.............', CR, LF, LF,
       'CPU.....................................', CR, LF,
       'ROM.....................................', CR, LF,
       'MASTER PIC..............................', CR, LF,
       'SLAVE_PIC...............................', CR, LF,
       'TIME OUT................................', CR, LF,
       'PIT.....................................', CR, LF,
       'MPSC....................................', CR, LF,
       'NUMERIC COPROCESSOR.....................', CR, LF,
       'RAM.....................................', CR, LF, LF,
       '<ESC> returns to the 1442 CPU BOARD MENU', EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @DISPLAY);

   if BIT_RESULT.CPU_BOARD = PASSED then
      call WRITE_POLL (VDU, @PASSED_1);
   else
      call WRITE_POLL (VDU, @FAILED_1);

   if BIT_RESULT.CPU = PASSED then
      call WRITE_POLL (VDU, @PASSED_3);
   else
      call WRITE_POLL (VDU, @FAILED_3);
   if BIT_RESULT.ROM = PASSED then
      call WRITE_POLL (VDU, @PASSED_4);
   else
      call WRITE_POLL (VDU, @FAILED_4);
   if BIT_RESULT.MASTER_PIC = PASSED then
      call WRITE_POLL (VDU, @PASSED_5);
   else
      call WRITE_POLL (VDU, @FAILED_5);
```

```
   if BIT_RESULT.SLAVE_PIC = PASSED then
      call WRITE_POLL (VDU, @PASSED_6);
   else
      call WRITE_POLL (VDU, @FAILED_6);
   if BIT_RESULT.TMOUT_CIRCUIT = PASSED then
      call WRITE_POLL (VDU, @PASSED_7);
   else
      call WRITE_POLL (VDU, @FAILED_7);
   if BIT_RESULT.PIT = PASSED then
      call WRITE_POLL (VDU, @PASSED_8);
   else
      call WRITE_POLL (VDU, @FAILED_8);
   if BIT_RESULT.MPSC = PASSED then
      call WRITE_POLL (VDU, @PASSED_9);
   else
      call WRITE_POLL (VDU, @FAILED_9);
   if BIT_RESULT.COPROCESSOR = PASSED then
      call WRITE_POLL (VDU, @PASSED_10);
   else
      call WRITE_POLL (VDU, @FAILED_10);
   if BIT_RESULT.RAM = PASSED then
      call WRITE_POLL (VDU, @PASSED_11);
   else
      call WRITE_POLL (VDU, @FAILED_11);

   end PRINT_1442_RESULTS;

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

```
PRINT_1611_RESULTS: procedure;

   declare DISPLAY (*) byte data

      ('RESULTS OF HECS 1611 EPROM/RAM BOARD BIT.....', CR, LF, LF,
       'ROM.....', CR, LF,
       'RAM.....', CR, LF, LF,
       '<ESC> returns to the 1611 EPROM/RAM BOARD MENU', EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @DISPLAY);

   if (BIT_RESULT.EROM = PASSED) and (BIT_RESULT.ERAM = PASSED) then
      call WRITE_POLL (VDU, @PASSED_1);
   else
      call WRITE_POLL (VDU, @FAILED_1);

   if BIT_RESULT.EROM = PASSED then
      call WRITE_POLL (VDU, @PASSED_3);
   else
      call WRITE_POLL (VDU, @FAILED_3);
   if BIT_RESULT.ERAM = PASSED then
      call WRITE_POLL (VDU, @PASSED_4);
   else
      call WRITE_POLL (VDU, @FAILED_4);

   end PRINT_1611_RESULTS;

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

```
PRINT_6123_RESULTS: procedure;

   declare DISPLAY (*) byte data

      ('RESULTS OF HECS 6123 1553 SDB BIT.............', CR, LF, LF,
       '6123 Controller card self test................', CR, LF,
       'RAM.......................................', CR, LF, LF,
       '<ESC> returns to the 6123 CONTROLLER CARD MENU', EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @DISPLAY);

   if BIT_RESULT.SDB_BOARD = PASSED then
      call WRITE_POLL (VDU, @PASSED_1);
   else
      call WRITE_POLL (VDU, @FAILED_1);

   if BIT_RESULT.SDB_SELFTEST = PASSED then
      call WRITE_POLL (VDU, @PASSED_3);
   else
      call WRITE_POLL (VDU, @FAILED_3);
   if BIT_RESULT.SDB_RAM = PASSED then
      call WRITE_POLL (VDU, @PASSED_4);
   else
      call WRITE_POLL (VDU, @FAILED_4);

   end PRINT_6123_RESULTS;

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

```
/*****************************************************************************
*                                                                           *
*      Standard computing segment off-line diagnostic test routines.        *
*                                                                           *
*****************************************************************************/

OFF_LINE_1442_TESTS: procedure;

declare CPU_BOARD_MENU (*) byte data
   ('HECS 1442 CPU BOARD BUILT IN TESTS', CR, LF, LF,
    'A) Print HECS 1442 Built In Test results', CR, LF,
    'B) Test entire CPU board', CR, LF,
    'C) Test CPU', CR, LF,
    'D) Test on board ROM', CR, LF,
    'E) Test Master Programmable Interrupt Controller', CR, LF,
    'F) Test Slave Programmable Interrupt Controller', CR, LF,
    'G) Test Time out facility', CR, LF,
    'H) Test Programmable Interval Timer', CR, LF,
    'I) Test Numeric Coprocessor', CR, LF,
    'J) Test Multi Protocol Serial Controller', CR, LF,
    'K) Test on board RAM', CR, LF, LF,
    '<ESC> returns to the main menu', EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @CPU_BOARD_MENU);
   LETTER = INPUT_CHARACTER (KB) ;

   do while (LETTER <> ESC) and ((LETTER < 'A') or (LETTER > 'K'));
      call WRITE_POLL (VDU, @ILLEGAL_INPUT);
      call TIME (2000);
      call WRITE_POLL (VDU, @SPACES_BOTTOM);
      call TIME (2000);
      call WRITE_POLL (VDU, @ILLEGAL_INPUT);
      LETTER = INPUT_CHARACTER (KB);
      end; /* while LETTER < 'A' or LETTER > 'K' */
```

```
do while LETTER <> ESC;

   do case LETTER - 'A';

      do;
         call PRINT_1442_RESULTS;                              /* A */
         LETTER = INPUT_CHARACTER (KB);
         do while (LETTER <> ESC);
            call WRITE_POLL (VDU, @ILLEGAL_INPUT);
            call TIME (2000);
            call WRITE_POLL (VDU, @SPACES_BOTTOM);
            call TIME (2000);
            call WRITE_POLL (VDU, @ILLEGAL_INPUT);
            LETTER = INPUT_CHARACTER (KB);
            end;
         end;

      do;
         call WRITE_POLL (VDU, @BUSY_4);                       /* B */
         BIT_RESULT.CPU = CPU_TEST;
         call ON_BOARD_EPROM_TEST;
         call MPIC_AND_TMOUT_TEST;
         call MPIC_AND_PIT_TEST;
         call SPIC_MPIC_AND_PIT_TEST;
         BIT_RESULT.MPSC = MPSC_IO_TEST (VDU);
         call COPROCESSOR_TEST;
         BIT_RESULT.RAM = MEM_TEST;
         if (BIT_RESULT.CPU = PASSED) and (BIT_RESULT.ROM           = PASSED)
                                      and (BIT_RESULT.MASTER_PIC    = PASSED)
                                      and (BIT_RESULT.SLAVE_PIC     = PASSED)
                                      and (BIT_RESULT.TMOUT_CIRCUIT = PASSED)
                                      and (BIT_RESULT.PIT           = PASSED)
                                      and (BIT_RESULT.COPROCESSOR   = PASSED)
                                      and (BIT_RESULT.MPSC          = PASSED)
                                      and (BIT_RESULT.RAM           = PASSED) then

            do;
            BIT_RESULT.CPU_BOARD = PASSED;
            call WRITE_POLL (VDU, @PASSED_4);
            end;
         else
            do;
            BIT_RESULT.CPU_BOARD = FAILED;
            call WRITE_POLL (VDU, @FAILED_4);
            end;
         end;     /* B */
```

```
do;
   BIT_RESULT.CPU = CPU_TEST;                          /* C */
   if BIT_RESULT.CPU = PASSED then
      call WRITE_POLL (VDU, @PASSED_5);
   else
      call WRITE_POLL (VDU, @FAILED_5);
   end;

do;
   call ON_BOARD_EPROM_TEST;                           /* D */
   if BIT_RESULT.ROM = PASSED then
      call WRITE_POLL (VDU, @PASSED_6);
   else
      call WRITE_POLL (VDU, @FAILED_6);
   end;

do;
   call MPIC_AND_TMOUT_TEST;                           /* E */
   if BIT_RESULT.MASTER_PIC = FAILED then
      call MPIC_AND_PIT_TEST;
   if BIT_RESULT.MASTER_PIC = PASSED then
      call WRITE_POLL (VDU, @PASSED_7);
   else
      call WRITE_POLL (VDU, @FAILED_7);
   end;

do;
   call SPIC_MPIC_AND_PIT_TEST;                        /* F */
   if BIT_RESULT.SLAVE_PIC = PASSED then
      call WRITE_POLL (VDU, @PASSED_8);
   else
      call WRITE_POLL (VDU, @FAILED_8);
   end;

do;
   call MPIC_AND_TMOUT_TEST;                           /* G */
   if BIT_RESULT.TMOUT_CIRCUIT = PASSED then
      call WRITE_POLL (VDU, @PASSED_9);
   else
      call WRITE_POLL (VDU, @FAILED_9);
   end;

do;
   call MPIC_AND_PIT_TEST;                             /* H */
   if BIT_RESULT.PIT = PASSED then
      call WRITE_POLL (VDU, @PASSED_10);
   else
      call WRITE_POLL (VDU, @FAILED_10);
   end;
```

```
      do;
         call COPROCESSOR_TEST;                                  /* I */
         if BIT_RESULT.COPROCESSOR = PASSED then
            call WRITE_POLL (VDU, @PASSED_11);
         else
            call WRITE_POLL (VDU, @FAILED_11);
         end;

      do;
         BIT_RESULT.MPSC = MPSC_IO_TEST (VDU);                   /* J */
         if BIT_RESULT.MPSC = PASSED then
            call WRITE_POLL (VDU, @PASSED_12);
         else
            call WRITE_POLL (VDU, @FAILED_12);
         end;

      do;
         call WRITE_POLL (VDU, @BUSY_13);                        /* K */
         BIT_RESULT.RAM = MEM_TEST;
         if BIT_RESULT.RAM = PASSED then
            call WRITE_POLL (VDU, @PASSED_13);
         else
            call WRITE_POLL (VDU, @FAILED_13);
         end;
      end; /* case */

      if LETTER = ESC then /* print routine was chosen */
         do;
         call WRITE_POLL (VDU, @CLEARSCREEN);
         call WRITE_POLL (VDU, @CPU_BOARD_MENU);
         end;

      LETTER = INPUT_CHARACTER (KB);

      do while (LETTER <> ESC) and ((LETTER < 'A') or (LETTER > 'K'));
         call WRITE_POLL (VDU, @ILLEGAL_INPUT);
         call TIME (2000);
         call WRITE_POLL (VDU, @SPACES_BOTTOM);
         call TIME (2000);
         call WRITE_POLL (VDU, @ILLEGAL_INPUT);
         LETTER = INPUT_CHARACTER (KB);
         end; /* while LETTER < 'A' or LETTER > 'K' */

      end; /* while LETTER <> ESC */

   end OFF_LINE_1442_TESTS;

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

```
OFF_LINE_1611_TESTS: procedure;

   declare EPROM_RAM_BOARD_MENU (*) byte data
      ('HECS 1611 EPROM/RAM CARD BUILT IN TESTS', CR, LF, LF,
       'A) Print HECS 1611 EPROM/RAM card Built In Test results', CR, LF,
       'B) Test entire EPROM/RAM card', CR, LF,
       'C) Test ROM on EPROM/RAM card', CR, LF,
       'D) Test RAM on EPROM/RAM card', CR, LF, LF,
       '<ESC> returns to the main menu', EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @EPROM_RAM_BOARD_MENU);
   LETTER = INPUT_CHARACTER (KB);

   do while (LETTER <> ESC) and ((LETTER < 'A') or (LETTER > 'D'));
      call WRITE_POLL (VDU, @ILLEGAL_INPUT);
      call TIME (2000);
      call WRITE_POLL (VDU, @SPACES_BOTTOM);
      call TIME (2000);
      call WRITE_POLL (VDU, @ILLEGAL_INPUT);
      LETTER = INPUT_CHARACTER (KB);
      end; /* while LETTER < 'A' or LETTER > 'D' */

   do while LETTER <> ESC;

      do case LETTER - 'A';

         do;
            call PRINT_1611_RESULTS;                          /* A */
            LETTER = INPUT_CHARACTER (KB);
            do while (LETTER <> ESC);
               call WRITE_POLL (VDU, @ILLEGAL_INPUT);
               call TIME (2000);
               call WRITE_POLL (VDU, @SPACES_BOTTOM);
               call TIME (2000);
               call WRITE_POLL (VDU, @ILLEGAL_INPUT);
               LETTER = INPUT_CHARACTER (KB);
               end;
            end;

         do;
           call WRITE_POLL (VDU, @BUSY_4);                    /* B */
           BIT_RESULT.ERAM = ERAM_TEST;
           call APPLICATIONS_ROM_TEST;
           if (BIT_RESULT.ERAM = PASSED) and (BIT_RESULT.EROM = PASSED) then
              call WRITE_POLL (VDU, @PASSED_4);
           else
              call WRITE_POLL (VDU, @FAILED_4);
           end;
```

```
      do;
         call APPLICATIONS_ROM_TEST;                          /* C */
         if BIT_RESULT.EROM = PASSED then
            call WRITE_POLL (VDU, @PASSED_5);
         else
            call WRITE_POLL (VDU, @FAILED_5);
         end;

      do;
         call WRITE_POLL (VDU, @BUSY_6);                      /* D */
         BIT_RESULT.ERAM = ERAM_TEST;
         if BIT_RESULT.ERAM = PASSED then
            call WRITE_POLL (VDU, @PASSED_6);
         else
            call WRITE_POLL (VDU, @FAILED_6);
         end;
       end; /* case */

       if LETTER = ESC then /* print routine was chosen */
          do;
          call WRITE_POLL (VDU, @CLEARSCREEN);
          call WRITE_POLL (VDU, @EPROM_RAM_BOARD_MENU);
          end;

    LETTER = INPUT_CHARACTER (KB);

    do while (LETTER <> ESC) and ((LETTER < 'A') or (LETTER > 'D'));
       call WRITE_POLL (VDU, @ILLEGAL_INPUT);
       call TIME (2000);
       call WRITE_POLL (VDU, @SPACES_BOTTOM);
       call TIME (2000);
       call WRITE_POLL (VDU, @ILLEGAL_INPUT);
       LETTER = INPUT_CHARACTER (KB);
       end; /* while LETTER < 'A' or LETTER > 'D' */
    end; /* while */

 end OFF_LINE_1611_TESTS;

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

```
OFF_LINE_6123_TESTS: procedure;

   declare SDB_BOARD_MENU (*) byte data
      ('HECS 6123 CONTROLLER CARD BUILT IN TESTS', CR, LF, LF,
       'A) Print HECS 6123 controller card Built In Test results', CR, LF,
       'B) Test entire 6123 controller card', CR, LF,
       'C) Invoke 6123 controller card self test', CR, LF,
       'D) Test RAM on 6123 controller card', CR, LF, LF,
       '<ESC> returns to the main menu', EOM);

   call WRITE_POLL (VDU, @CLEARSCREEN);
   call WRITE_POLL (VDU, @SDB_BOARD_MENU);
   LETTER = INPUT_CHARACTER (KB);

   do while (LETTER <> ESC) and ((LETTER < 'A') or (LETTER > 'D'));
      call WRITE_POLL (VDU, @ILLEGAL_INPUT);
      call TIME (2000);
      call WRITE_POLL (VDU, @SPACES_BOTTOM);
      call TIME (2000);
      call WRITE_POLL (VDU, @ILLEGAL_INPUT);
      LETTER = INPUT_CHARACTER (KB);
      end; /* while LETTER < 'A' or LETTER > 'D' */

   do while LETTER <> ESC;

      do case LETTER - 'A';

         do;
            call PRINT_6123_RESULTS;                          /* A */
            LETTER = INPUT_CHARACTER (KB);
            do while (LETTER <> ESC);
               call WRITE_POLL (VDU, @ILLEGAL_INPUT);
               call TIME (2000);
               call WRITE_POLL (VDU, @SPACES_BOTTOM);
               call TIME (2000);
               call WRITE_POLL (VDU, @ILLEGAL_INPUT);
               LETTER = INPUT_CHARACTER (KB);
               end;
            end;
```

```
do;
   call WRITE_POLL (VDU, @BUSY_4);                    /* B */
   call SDB_SELFTEST;
   call SDB_RAM_TEST;
   if (BIT_RESULT.SDB_SELFTEST = PASSED)
   and (BIT_RESULT.SDB_RAM = PASSED) then
      do;
      BIT_RESULT.SDB_BOARD = PASSED;
      call WRITE_POLL (VDU, @PASSED_4);
      end;
   else
      do;
      BIT_RESULT.SDB_BOARD = FAILED;
      call WRITE_POLL (VDU, @FAILED_4);
      end;
   end;

do;
   call SDB_SELFTEST;                                 /* C */
   if BIT_RESULT.SDB_SELFTEST = PASSED then
      call WRITE_POLL (VDU, @PASSED_5);
   else
      call WRITE_POLL (VDU, @FAILED_5);
   end;

do;
   call WRITE_POLL (VDU, @BUSY_6);                    /* D */
   call SDB_RAM_TEST;
   if BIT_RESULT.SDB_RAM = PASSED then
      call WRITE_POLL (VDU, @PASSED_6);
   else
      call WRITE_POLL (VDU, @FAILED_6);
   end;
end; /* case */
```

```
         if LETTER = ESC then /* print routine was chosen */
            do;
            call WRITE_POLL (VDU, @CLEARSCREEN);
            call WRITE_POLL (VDU, @SDB_BOARD_MENU);
            end;

      LETTER = INPUT_CHARACTER (KB);

      do while (LETTER <> ESC) and ((LETTER < 'A') or (LETTER > 'D'));
         call WRITE_POLL (VDU, @ILLEGAL_INPUT);
         call TIME (2000);
         call WRITE_POLL (VDU, @SPACES_BOTTOM);
         call TIME (2000);
         call WRITE_POLL (VDU, @ILLEGAL_INPUT);
         LETTER = INPUT_CHARACTER (KB);
         end; /* while LETTER < 'A' or LETTER > 'D' */
      end; /* while LETTER <> ESC */

   end OFF_LINE_6123_TESTS;

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

```
DIAGNOSTICS: procedure public;

/*****************************************************************************
*                                                                            *
*         Menu driving routine for the off-line diagnostics of the           *
*         standard computing segment.                                        *
*                                                                            *
*****************************************************************************/

  declare FOREVER byte;
  declare DIAGNOSTIC_MENU (*) byte data
     ('OFF LINE DIAGNOSTIC BUILT IN TESTS FOR THE STANDARD COMPUTING SEGMENT',
       CR, LF, LF,
      'A) Display BIT results', CR, LF,
      'B) Test HECS 1442 CPU board', CR, LF,
      'C) Test HECS 1611 EPROM/RAM card', CR, LF,
      'D) Test HECS 6123 1553 System Data Bus board', CR, LF, LF,
      'Choose a letter...', EOM);

  TEST = TRUE;
  enable;
  call WRITE_POLL (VDU, @CLEARSCREEN);
  call WRITE_POLL (VDU, @DIAGNOSTIC_MENU);
  LETTER = INPUT_CHARACTER (KB);

  do while (LETTER < 'A') or (LETTER > 'D');
     call WRITE_POLL (VDU, @ILLEGAL_INPUT);
     call TIME (2000);
     call WRITE_POLL (VDU, @SPACES_BOTTOM);
     call TIME (2000);
     call WRITE_POLL (VDU, @ILLEGAL_INPUT);
     LETTER = INPUT_CHARACTER (KB);
     end; /* while LETTER < 'A' or LETTER > 'D' */
```

```
   do case LETTER - 'A';
      do;
         call PRINT_RESULTS;                                  /* A */
         call WRITE_POLL (VDU, @CONTINUE);
         call TIME (5000);
         call WRITE_POLL (VDU, @SPACES_BOTTOM); /* Causes ESC message */
         call TIME (2000);                      /* to blink           */
         call WRITE_POLL (VDU, @CONTINUE);
         LETTER = INPUT_CHARACTER (KB);
         do while LETTER <> ESC;
            call WRITE_POLL (VDU, @ILLEGAL_INPUT);
            call TIME (2000);
            call WRITE_POLL (VDU, @SPACES_BOTTOM);
            call TIME (2000);
            call WRITE_POLL (VDU, @ILLEGAL_INPUT);
            LETTER = INPUT_CHARACTER (KB);
            end; /* while LETTER <> ESC */
         end;
      call OFF_LINE_1442_TESTS;                               /* B */
      call OFF_LINE_1611_TESTS;                               /* C */
      call OFF_LINE_6123_TESTS;                               /* D */

      end;  /* case */

   if LETTER <> ESC then
      call TIME (5000);   /* Delay for result to be displayed */

   TEST = FALSE;

   end DIAGNOSTICS;

end DIAG;
```

2.5.1.9 STDIO.PLM

```
$ include (PLMPAR.INC)
$ nointvector
/*****************************************************************************
 *                                                                           *
 *        MODULE NAME : IO_8274                                              *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Source Filename  : STDIO.PLM                                            *
 *                                                                           *
 *   Source Compiler  : PLM86                                                *
 *                                                                           *
 *   Operating System : DOS 3.10                                             *
 *                                                                           *
 *   Description      : Standard I/O routines for the 8274 MPSC chip,        *
 *                      comprising the following routines :                  *
 *                                                                           *
 *      a)  INITIALISE_IO_ADDRESSES - Sets up the raw state of the MPSC.     *
 *      b)  INITIALISE_PORT - Configures MPSC registers to user defined      *
 *                            values.                                        *
 *      c)  MPSC_IO_TEST - Tests the 8274 MPSC device.                       *
 *      d)  ENABLE_PORT - Enables Tx and Rx of the serial port.              *
 *      e)  DISABLE_PORT - Disbales Tx and Rx of the serial port.            *
 *      f)  CLEAR_BUFFER - Clears an input buffer of unread characters.      *
 *      g)  TRANSMIT_CHARACTER - Called by a transmit buffer empty           *
 *             interrupt the routine transfers a character from the          *
 *             write buffer to the port.                                     *
 *      h)  RECEIVE_CHARACTER - Called by a receive character interrupt      *
 *             the routine transfers the character to the input buffer.      *
 *      i)  GET_MODE - returns the mode of a serial link.                    *
 *      j)  SET_MODE - sets the mode of a serial link.                       *
 *      k)  WRITE - Transfers the contents of a buffer to a write buffer     *
 *             and links the buffer to the port data.                        *
 *      l)  WRITE_POLL - Writes a buffer to a port using a polled mode       *
 *             of operation.                                                 *
 *      m)  INPUT_CHARACTER - Clears the input buffers and waits for a       *
 *             character to be entered (poll mode).                          *
 *      n)  GET_CHARACTER - Returns the next character in the input buffer.  *
 *      o)  LAST_CHARACTER_READ - Returns the value of the last character    *
 *             received by the 8274 (ie. last character in the buffer).      *
 *      p)  GET_BYTE - Clears the input buffer, and waits for a <CR>, then   *
 *             returns the numeric value of the next numeric string in the   *
 *             input buffer.                                                 *
 *      q)  GET_WORD - As GET_BYTE but returning a word value.               *
 *                                                                           *
 *****************************************************************************
$ eject
```

```
*****************************************************************************
*                                                                           *
*      r)  BUFFER_READY - Returns a 00H if the buffer is empty.             *
*             Returns a 01H if data is available in a buffer.               *
*             Returns a 02H if a CR is in the buffer.                       *
*      s)  CHB_TX_EMPTY - Interrupt handler for channel B transmissions.    *
*      t)  CHA_TX_EMPTY - Interrupt handler for channel A transmissions.    *
*      u)  CHA_RX_AVAIL - Interrupt handler for characters available on A.  *
*      v)  CHB_RX_AVAIL - Interrupt handler for characters available on B.  *
*      w)  CHA_RX_ERROR - Interrupt handler for reception errors on A.      *
*      y)  CHB_RX_ERROR - Interrupt handler for reception errors on B.      *
*                                                                           *
*****************************************************************************

*****************************************************************************
*                                                                           *
*   Public procedures: INITIALISE_IO_ADDRESSES                              *
*                      INITIALISE_PORT                                      *
*                      MPSC_IO_TEST                                         *
*                      ENABLE_PORT                                          *
*                      DISABLE_PORT                                         *
*                      CLEAR_BUFFER                                         *
*                      TRANSMIT_CHARACTER                                   *
*                      RECEIVE_CHARACTER                                    *
*                      GET_MODE                                             *
*                      SET_MODE                                             *
*                      WRITE                                                *
*                      WRITE_POLL                                           *
*                      INPUT_CHARACTER                                      *
*                      GET_CHARACTER                                        *
*                      LAST_CHARACTER_READ                                  *
*                      GET_BYTE                                             *
*                      GET_WORD                                             *
*                      BUFFER_READY                                         *
*                      CHB_TX_EMPTY                                         *
*                      CHA_TX_EMPTY                                         *
*                      CHA_RX_AVAIL                                         *
*                      CHB_RX_AVAIL                                         *
*                      CHA_RX_ERROR                                         *
*                      CHB_RX_ERROR                                         *
*                                                                           *
*   EPD files        : STDPIC.EPD                                           *
*                      STDCONVT.EPD                                         *
*                                                                           *
*   Include files    : IOLITS.INC                                           *
*                                                                           *
*                                                                           *
*****************************************************************************/
$ eject
```

```
/*****************************************************************************
*                                                                            *
*   HISTORY   Version 1.0 :                                                  *
*                                                                            *
*             Designed by : A.J. POLMANS    Date :  February 1989          *
*                           P.A. OLANDER                                     *
*             Description : Original                                         *
*                                                                            *
*                                                                            *
*****************************************************************************/
$ eject

io_8274:  do;

$ include (IOLITS.INC)
$ eject

/*****************************************************************************/

declare MPSC_TEST_IN_PROGRESS byte;
declare MPSC_TEST_RESULT      byte;

declare VAR_MESSAGE (80)      byte ;

/*****************************************************************************/


/*********************************/
/*                               */
/*  Entities Publicly Declared   */
/*                               */
/*********************************/

$ include(STDPIC.EPD )
$ eject
$ include (STDCONVT.EPD)
$ eject
```

```
/*******************************/
/*                             */
/*   Local variables           */
/*                             */
/*******************************/

/*--- Port Data ---*/

declare PORT_DATA( NO_OF_PORTS ) structure
      ( CONTROL_PORT           word,    /* Address of 8274 control port    */
        DATA_PORT              word,    /* Address of 8274 data port       */

        NO_OF_DATA_BITS        byte,

        INTERRUPT_NO           byte,

        BUFFER_ADDRESS         pointer, /* Address of buffer being output  */
        LAST_BUFFER_ADDRESS    pointer, /* Address of last buffer in queue */
        NEXT_CHARACTER         byte,    /* Position of next character in   */
                                        /*     buffer to be printed.       */
        MODE                   byte,    /* Sets up mode of port :          */
                                        /*     0 = terminal mode           */
                                        /*     1 = data mode               */
        INPUT_FLAG             byte,    /* Status of input buffer.         */
                                        /*     0 = no valid data in buffer */
                                        /*     1 = some valid data         */
                                        /*     2 = <CR> received in buffer */
        NO_OF_CHARACTERS       byte,    /* No. of characters received in   */
                                        /*     the input buffer.           */
        READ_POINTER           byte,    /* Pointer to position of next     */
                                        /*     character to be read out    */
                                        /*     of the input buffer.        */
        LAST_CHARACTER_ECHOED  byte,    /* Last character in the input     */
                                        /*     buffer that has been echoed */
        INPUT_BUFFER( 80 )     byte );  /* Array of input characters.      */

/*--- Buffer Areas ---*/

declare BUFFER( NO_OF_BUFFERS ) structure
        ( OUTPUT_STRING( 80 )  byte,
          NEXT_BUFFER          pointer );


declare NEXT_FREE_BUFFER pointer;
declare LAST_FREE_BUFFER pointer;

$eject
/*****************************************************************************/
```

```
INITIALISE_IO_ADDRESSES : procedure( PORT_NO,
                                     BASE_ADDRESS,
                                     INTERRUPT_NO  )    public;

   declare  PORT_NO              byte;
   declare  BASE_ADDRESS         word;
   declare  INTERRUPT_NO         byte;

   output( BASE_ADDRESS + 4 ) = 018H; /* Channel reset */
   output( BASE_ADDRESS + 6 ) = 018H; /* Channel reset */

   PORT_DATA( PORT_NO ).DATA_PORT         = BASE_ADDRESS;
   PORT_DATA( PORT_NO ).CONTROL_PORT      = BASE_ADDRESS + 4;

   PORT_DATA( PORT_NO + 1 ).DATA_PORT     = BASE_ADDRESS + 2;
   PORT_DATA( PORT_NO + 1 ).CONTROL_PORT  = BASE_ADDRESS + 6;

   PORT_DATA( PORT_NO     ).INTERRUPT_NO  = INTERRUPT_NO;
   PORT_DATA( PORT_NO + 1 ).INTERRUPT_NO  = INTERRUPT_NO;

   if INTERRUPT_NO = 0 then
      do;   /* Not using interrupts */
      end;  /* Not using interrupts */
   else
      do;   /* Using interrupts */
      disable;

      /* Set up the interrupt vector for the chip on WR2 of */
      /* channel B.                                         */

      output( BASE_ADDRESS + 6 ) = 2;
      output( BASE_ADDRESS + 6 ) = INTERRUPT_NO;

      /* Set up the interrupt mode for the chip on WR2 of   */
      /* channel A.  Pin 10 = RTS                           */
      /*             Vectored Interrupt                     */
      /*             8086 mode                              */
      /*             Priority RxA > RxB > TxA > TxB         */
      /*             Both channels interrupt driven         */

      output( BASE_ADDRESS + 4 ) = 2;
      output( BASE_ADDRESS + 4 ) = 00110100b;

      enable;
      end;  /* Using interrupts */

   MPSC_TEST_IN_PROGRESS = FALSE;

   end INITIALISE_IO_ADDRESSES;
```

```
$ eject

/*************************************************************************/
/*                                                                       */
/*     INITIALISE_PORT : Writes the control characters required to set   */
/*         up a 8274 MPSC from the table set up in the port data area    */
/*         BUFFER(I).                                                    */
/*                                                                       */
/*     INPUT : A pointer to the correct buffer area.                     */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*     Any output in progress at the time of initialisation is aborted   */
/*     and the buffers restored to the free buffer list.                 */
/*                                                                       */
/*************************************************************************/

INITIALISE_PORT : procedure( PORT_NO,
                             BAUD_RATE,
                             NO_OF_DATA_BITS,
                             NO_OF_STOP_BITS,
                             PARITY_TYPE,
                             MODE ) public;

   declare PORT_NO         byte;
   declare BAUD_RATE       byte;
   declare NO_OF_DATA_BITS byte;
   declare NO_OF_STOP_BITS byte;
   declare PARITY_TYPE     byte;
   declare MODE            byte;

   /*--------------------------------------*/

   declare B_ADDRESS pointer;
   declare BUFFER based B_ADDRESS structure
      ( OUTPUT_STRING(80) byte,
        NEXT_BUFFER       pointer );

   /*--------------------------------------*/
```

```
PORT_DATA( PORT_NO ).NO_OF_DATA_BITS = NO_OF_DATA_BITS;

/*   Set up * 16 clock rate            */
/*          parity and no of stop bits  */
/*          as per parameters.          */

output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 4;
output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 01000000b or
                                              NO_OF_STOP_BITS or
                                              PARITY_TYPE;

/*   Set up Rx register as follows              */
/*            no of data bits as per parameters  */
/*            Rx enabled                         */

output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 3;
output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = NO_OF_DATA_BITS or
                                              00000001b;

/*   Set up Tx register as follows              */
/*            no of data bits as per parameters  */
/*            Rx enabled                         */
/*            no control lines active            */

output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 5;
output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = shr( NO_OF_DATA_BITS, 1 ) or
                                              00001000b;

if PORT_DATA( PORT_NO ).INTERRUPT_NO = 0 then
   do;  /* Disable Interrupts */

   /*   Set up EXT INTERRUPT - Disabled    */
   /*          TX  INTERRUPT - Disabled    */
   /*          RX  INTERRUPT - Disabled    */

   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 1;
   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 00000000b;
   end;
else
   do;  /* Enable Interrupts */

   /*   Set up EXT INTERRUPT - Disabled    */
   /*          TX  INTERRUPT - Enabled     */
   /*          VARIABLE INTERRUPT VECTOR   */
   /*          INT on all Rx characters    */

   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 1;
   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 00011110b;
   end;
```

```
   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 028H; /* Reset Tx interrupt pending */
   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 030H; /* Reset Error conditions */

   PORT_DATA( PORT_NO ).NEXT_CHARACTER = 0;
   PORT_DATA( PORT_NO ).INPUT_FLAG = 0;
   PORT_DATA( PORT_NO ).INPUT_BUFFER(0) = 0;
   PORT_DATA( PORT_NO ).NO_OF_CHARACTERS = 0;
   PORT_DATA( PORT_NO ).LAST_CHARACTER_ECHOED = 0;
   PORT_DATA( PORT_NO ).READ_POINTER = 0;

   /*******************************************************/
   /*                                                     */
   /*     If a list of buffers was allocated to the port, */
   /*     then chain them on to the end of the free       */
   /*     buffer list.                                    */
   /*                                                     */
   /*******************************************************/

   if PORT_DATA( PORT_NO ).BUFFER_ADDRESS <> nil then
      do;
      B_ADDRESS = LAST_FREE_BUFFER;
      BUFFER.NEXT_BUFFER = PORT_DATA( PORT_NO ).BUFFER_ADDRESS;
      LAST_FREE_BUFFER = PORT_DATA( PORT_NO ).LAST_BUFFER_ADDRESS;
      PORT_DATA( PORT_NO ).BUFFER_ADDRESS = nil;
      PORT_DATA( PORT_NO ).LAST_BUFFER_ADDRESS = nil;
      end;

   end INITIALISE_PORT;

$ eject
```

```
/*************************************************************************/
/*                                                                       */
/*     MPSC_IO_TEST : Called to test the operation of the 8274           */
/*                    multiple protocol serial controller.               */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : A test result.                                           */
/*                1 ==> Test passed                                      */
/*                2 ==> Test failed                                      */
/*                                                                       */
/*************************************************************************/

MPSC_IO_TEST : procedure ( PORT_NO ) byte public;

   declare PORT_NO  byte;

   /*   Unmask the Interrupt Controller for the chip, */
   /*   then send a bel to the keyboard, and wait   */
   /*   for an interrupt to be returned.            */

   MPSC_TEST_IN_PROGRESS = TRUE;
   MPSC_TEST_RESULT      = FALSE;

   call UNMASK_PIC( MPSC_INTERRUPT_NO );
   output( PORT_DATA( PORT_NO ).DATA_PORT ) = BEL;
   call TIME( 10 );   /* 1 millisecond delay */

   MPSC_TEST_IN_PROGRESS = FALSE;

   if MPSC_TEST_RESULT = TRUE then
      return( 1 );
   else
      return( 2 );

   end MPSC_IO_TEST;

$ eject
```

```
/*************************************************************************/
/*                                                                       */
/*     ENABLE_PORT : Called to enable the Tx and Rx of the port.         */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

ENABLE_PORT : procedure ( PORT_NO ) public;

   declare PORT_NO  byte;

   /*   Set up Rx register as follows              */
   /*            no of data bits as per parameters  */
   /*            Rx enabled                         */

   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 3;
   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = PORT_DATA( PORT_NO ).NO_OF_DATA_BITS
                                                 or 00000001b;

   /*   Set up Tx register as follows              */
   /*            no of data bits as per parameters  */
   /*            Rx enabled                         */
   /*            no control lines active            */


   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 5;
   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = shr( PORT_DATA( PORT_NO ).NO_OF_DATA_BITS, 1 )
                                                 or 00001000b;

   end ENABLE_PORT;

$ eject
```

```
/*************************************************************************/
/*                                                                       */
/*     DISABLE_PORT : Called to disable the Tx and Rx of the port.       */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

DISABLE_PORT : procedure ( PORT_NO ) public;

   declare PORT_NO  byte;

   /*   Set up Rx register as follows              */
   /*            no of data bits as per parameters  */
   /*            Rx enabled                         */

   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 3;
   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = PORT_DATA( PORT_NO ).NO_OF_DATA_BITS
                                                 and 11111110b;

   /*   Set up Tx register as follows              */
   /*            no of data bits as per parameters  */
   /*            Rx enabled                         */
   /*            no control lines active            */


   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 5;
   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = shr( PORT_DATA( PORT_NO ).NO_OF_DATA_BITS, 1 )
                                                 and 11110111b;

   end DISABLE_PORT;

$ eject
```

```
/*************************************************************************/
/*                                                                       */
/*     CLEAR_BUFFER : Called to clear an input buffer of unread          */
/*         characters.  If the routine LAST_READ_CHARACTER is used, the  */
/*         buffer is never emptied and must be done so explicitly        */
/*         by the user.                                                  */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

CLEAR_BUFFER : procedure ( PORT_NO ) public;

   declare PORT_NO  byte;

   disable;
   PORT_DATA( PORT_NO ).NO_OF_CHARACTERS      = 0;
   PORT_DATA( PORT_NO ).READ_POINTER          = 0;
   PORT_DATA( PORT_NO ).LAST_CHARACTER_ECHOED = 0;
   PORT_DATA( PORT_NO ).INPUT_FLAG            = 0;
   enable;

   end CLEAR_BUFFER;

$ eject
```

```
/************************************************************************/
/*                                                                      */
/*     TRANSMIT_CHARACTER : Called as a direct result of a Tx buffer    */
/*         empty interrupt, this routine send the next character to the */
/*         8274, and if this emptys the transmit buffer, the next       */
/*         buffer is linked to the port data area.  If no characters    */
/*         are waiting to be sent, the Tx pending flag in the 8274 is   */
/*         reset.                                                       */
/*                                                                      */
/*     INPUT : A pointer to the correct port data area.                 */
/*                                                                      */
/*     OUTPUT : none.                                                   */
/*                                                                      */
/************************************************************************/

TRANSMIT_CHARACTER : procedure( PORT_NO ) public;

   declare PORT_NO     byte;

   declare B_ADDRESS pointer;
   declare BUFFER based B_ADDRESS structure
      ( OUTPUT_STRING(80) byte,
        NEXT_BUFFER       pointer );

   declare TEMP_ADDRESS pointer;

   /* Is there a character to be echoed ?? */
   if PORT_DATA( PORT_NO ).MODE = TERMINAL and
         PORT_DATA( PORT_NO ).LAST_CHARACTER_ECHOED < PORT_DATA( PORT_NO ).NO_OF_CHARACTERS then
      do;
      PORT_DATA( PORT_NO ).LAST_CHARACTER_ECHOED = PORT_DATA( PORT_NO ).LAST_CHARACTER_ECHOED + 1;
      if PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).LAST_CHARACTER_ECHOED ) >= ' ' then
         output( PORT_DATA( PORT_NO ).DATA_PORT ) =
              PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).LAST_CHARACTER_ECHOED );
      end;

   else
     if PORT_DATA( PORT_NO ).BUFFER_ADDRESS = nil then
       /* No characters to be transmitted so reset Tx pending */
       output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 028H;  /* Reset Tx pending flag */

      else
         do; /* Send next character */
         B_ADDRESS = PORT_DATA( PORT_NO ).BUFFER_ADDRESS;
         if BUFFER.OUTPUT_STRING( PORT_DATA( PORT_NO ).NEXT_CHARACTER ) = EOM then
            do; /* Go on to next buffer */
            PORT_DATA( PORT_NO ).BUFFER_ADDRESS = BUFFER.NEXT_BUFFER;
```

```
            /* Restore the used buffer to the end of the free buffer list */
            BUFFER.NEXT_BUFFER = nil;
            if NEXT_FREE_BUFFER = nil then
               do;
               NEXT_FREE_BUFFER = B_ADDRESS;
               LAST_FREE_BUFFER = B_ADDRESS;
               end;
            else
               do;
               TEMP_ADDRESS = B_ADDRESS;
               B_ADDRESS = LAST_FREE_BUFFER;
               BUFFER.NEXT_BUFFER = TEMP_ADDRESS;
               LAST_FREE_BUFFER = TEMP_ADDRESS;
               end;

            if PORT_DATA( PORT_NO ).BUFFER_ADDRESS = nil then
               do; /* No next buffer */
               /* Set end pointer to nil */
               PORT_DATA( PORT_NO ).LAST_BUFFER_ADDRESS = nil;
               /* Reset Tx interrupt pending flag */
               output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 028H;
               end; /* No next buffer */
            else
               do; /* Send first character from next buffer */
               B_ADDRESS = PORT_DATA( PORT_NO ).BUFFER_ADDRESS;
               output( PORT_DATA( PORT_NO ).DATA_PORT ) = BUFFER.OUTPUT_STRING(0);
               PORT_DATA( PORT_NO ).NEXT_CHARACTER = 1;
               end; /* Send first character from next buffer */

            end; /* Go on to next buffer */

         else
            do;  /* Send next character from same buffer */
            output( PORT_DATA( PORT_NO ).DATA_PORT ) =
            BUFFER.OUTPUT_STRING( PORT_DATA( PORT_NO ).NEXT_CHARACTER );
            PORT_DATA( PORT_NO ).NEXT_CHARACTER = PORT_DATA( PORT_NO ).NEXT_CHARACTER + 1;
            end; /* Send next character from same buffer */
         end; /* Send next character */

   /* Now reset the interrupts down the chain of PICs etc. */
   output( PORT_DATA( PORT_NO ).CONTROL_PORT and 0FCh ) = 038H; /* WR0 function 7 */
   call ISSUE_EOI( MPSC_INTERRUPT_NO );

   end TRANSMIT_CHARACTER;

$ eject
```

```
/*************************************************************************/
/*                                                                       */
/*     RECEIVE_CHARACTER : Called as a direct result of a Rx buffer      */
/*         available interrupt, this routine transfers the received      */
/*         character to the INPUT_BUFFER in PORT_DATA( PORT_NO ).        */
/*         If the character is a <CR>, the INPUT_FLAG is set to 2.       */
/*            else INPUT_FLAG is set to 1.                               */
/*         If the input character is a <DEL> or a <BACK_SPACE> then      */
/*         a backspace, space, backspace is transmitted and the char     */
/*         deleted from the buffer.                                      */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

RECEIVE_CHARACTER : procedure( PORT_NO ) public;

   declare IN_CHAR byte;

   declare PORT_NO      byte;

   /*********************************************************************/
   /*                                                                   */
   /*           Receive char from terminal                              */
   /*                                                                   */
   /*********************************************************************/

   RECEIVE_CHAR_FROM_TERMINAL: procedure;

   if IN_CHAR = BACK_SPACE or IN_CHAR = DEL then
      do; /* BACK_SPACE or DEL */
      if  PORT_DATA( PORT_NO ).NO_OF_CHARACTERS > 0 then
         do;
         PORT_DATA( PORT_NO ).NO_OF_CHARACTERS = PORT_DATA( PORT_NO ).NO_OF_CHARACTERS - 1;
         PORT_DATA( PORT_NO ).LAST_CHARACTER_ECHOED=PORT_DATA( PORT_NO ).LAST_CHARACTER_ECHOED-1;
         if PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).NO_OF_CHARACTERS ) >= ' ' then
            call WRITE( PORT_NO, @RUBOUT_MESSAGE );
         if PORT_DATA( PORT_NO ).NO_OF_CHARACTERS = 0 then
            PORT_DATA( PORT_NO ).INPUT_FLAG = 0;
         end;
      end; /* BACK_SPACE or DEL */
```

```
else if IN_CHAR = CR then
   do;
   PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).NO_OF_CHARACTERS ) = CR;
   PORT_DATA( PORT_NO ).INPUT_FLAG = 2;
   end;

else if IN_CHAR = LF then
   do;
   /*  Ignore line feeds in terminal mode  */
   end;

else if PORT_DATA( PORT_NO ).NO_OF_CHARACTERS < 79 then
   do;
   PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).NO_OF_CHARACTERS ) = IN_CHAR;
   if PORT_DATA( PORT_NO ).BUFFER_ADDRESS = nil then
      do;  /* Initiate echo of character */
      output( PORT_DATA( PORT_NO ).DATA_PORT ) = IN_CHAR;
      PORT_DATA( PORT_NO ).LAST_CHARACTER_ECHOED=PORT_DATA( PORT_NO ).LAST_CHARACTER_ECHOED+1;
      end;  /* Initiate echo of character */

   PORT_DATA( PORT_NO ).NO_OF_CHARACTERS = PORT_DATA( PORT_NO ).NO_OF_CHARACTERS + 1;
   PORT_DATA( PORT_NO ).INPUT_FLAG = 1;
   end;

end RECEIVE_CHAR_FROM_TERMINAL;

/*************************************************************************/
/*                                                                       */
/*           Receive char from data terminal                             */
/*                                                                       */
/*************************************************************************/

RECEIVE_DATA_CHAR: procedure;

/* Increment write pointer ( No of characters ) */
if PORT_DATA( PORT_NO ).NO_OF_CHARACTERS = 79 then
   PORT_DATA( PORT_NO ).NO_OF_CHARACTERS = 1;
else
   PORT_DATA( PORT_NO ).NO_OF_CHARACTERS = PORT_DATA( PORT_NO ).NO_OF_CHARACTERS + 1;

/* Check for overflow of the buffer */
if PORT_DATA( PORT_NO ).NO_OF_CHARACTERS = PORT_DATA( PORT_NO ).READ_POINTER then
   do; /* Reset write pointer to what is was */
```

```
      if PORT_DATA( PORT_NO ).NO_OF_CHARACTERS = 1 then
         PORT_DATA( PORT_NO ).NO_OF_CHARACTERS = 79;
      else
         PORT_DATA( PORT_NO ).NO_OF_CHARACTERS = PORT_DATA( PORT_NO ).NO_OF_CHARACTERS - 1;
      return;
      end; /* Reset write pointer */

   /* Store the character */
   PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).NO_OF_CHARACTERS ) = IN_CHAR;

   /* Set the flag */
   PORT_DATA( PORT_NO ).INPUT_FLAG = 1;

   /* If the read pointer is at zero, set it to the first element */
   if PORT_DATA( PORT_NO ).READ_POINTER = 0 then
      PORT_DATA( PORT_NO ).READ_POINTER = 1;

   end RECEIVE_DATA_CHAR;

   /*************************************************************************/


   IN_CHAR = input( PORT_DATA( PORT_NO ).DATA_PORT );
   if PORT_DATA( PORT_NO ).MODE = TERMINAL then
      do;
      if ( PORT_DATA( PORT_NO ).INPUT_FLAG <> 2 ) then
         call RECEIVE_CHAR_FROM_TERMINAL;
      end;
   else
      call RECEIVE_DATA_CHAR;


   /* Now reset the interrupts down the chain of PICs etc. */

   output( PORT_DATA( PORT_NO ).CONTROL_PORT and 0FCh ) = 038H; /* WR0 function 7 */
   call ISSUE_EOI( MPSC_INTERRUPT_NO );

   end RECEIVE_CHARACTER;

$ eject
```

```
/*************************************************************************/
/*                                                                       */
/*     GET_MODE: Returns the mode of a serial port.                      */
/*         0 = TERMINAL MODE                                             */
/*         1 = DATA MODE                                                 */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : The function returns a byte.                             */
/*                                                                       */
/*************************************************************************/

GET_MODE: procedure( PORT_NO ) byte public;

   declare PORT_NO  byte;

   return PORT_DATA( PORT_NO ).MODE;
   end;

$ eject
/*************************************************************************/
/*                                                                       */
/*     SET_MODE: Called to set the mode of a serial port.                */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*             The byte value of the new mode.                           */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

SET_MODE: procedure( PORT_NO, NEW_MODE ) public;

   declare PORT_NO  byte;
   declare NEW_MODE byte;

   PORT_DATA( PORT_NO ).MODE = NEW_MODE;
   end;

$ eject
```

```
/*************************************************************************/
/*                                                                       */
/*     WRITE : Transfers data from a user character string to a write    */
/*         buffer, and queues the write buffer for output.               */
/*         If a free buffer is not available the routine waits for a     */
/*         buffer to become free before continuing.                      */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO)          */
/*             A pointer to the user data string (DATA_ADDRESS)          */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

WRITE : procedure ( PORT_NO, DATA_ADDRESS ) public;

   declare PORT_NO  byte;
   declare DATA_ADDRESS pointer;
   declare ( DATA_STRING based DATA_ADDRESS )(80) byte;

   /*-------------------------------------------*/

   declare B_ADDRESS pointer;
   declare BUFFER based B_ADDRESS structure
      ( OUTPUT_STRING(80) byte,
        NEXT_BUFFER       pointer );

   declare I byte;
   declare FINISHED byte;
   declare BASE_INDEX word;   /* Index to the character at which the
                                 buffer is to started */
   declare TEMP_ADDRESS pointer;

   /*-------------------------------------------*/

   BASE_INDEX = 0;
   FINISHED = false;
   do while not FINISHED;
      do while NEXT_FREE_BUFFER = nil;
         /* Do nothing until a buffer becomes free */
         end;

      /* Get an empty buffer from the top of the free buffer list */
      disable;
      B_ADDRESS = NEXT_FREE_BUFFER;
      NEXT_FREE_BUFFER = BUFFER.NEXT_BUFFER;
      if NEXT_FREE_BUFFER = nil then
         LAST_FREE_BUFFER = nil;
      BUFFER.NEXT_BUFFER = nil;
      enable;
```

```
      /* Now transfer the data into the buffer */
      I = 0;
      do while DATA_STRING(BASE_INDEX+I) <> EOM and I < 79;
         BUFFER.OUTPUT_STRING(I) = DATA_STRING(BASE_INDEX+I);
         I = I + 1;
         end;
      BUFFER.OUTPUT_STRING(I) = EOM;

      if DATA_STRING(BASE_INDEX+I) <> EOM then
         BASE_INDEX = BASE_INDEX + 79;
      else
         FINISHED = true;

      /* Add the full buffer to the PORT_DATA buffer list */
      disable;
      if PORT_DATA( PORT_NO ).BUFFER_ADDRESS = nil then
         do;  /* No buffers currently being printed */
         PORT_DATA( PORT_NO ).BUFFER_ADDRESS = B_ADDRESS;
         PORT_DATA( PORT_NO ).LAST_BUFFER_ADDRESS = B_ADDRESS;
         /* Initiate transmission of first character */
         output( PORT_DATA( PORT_NO ).DATA_PORT ) = BUFFER.OUTPUT_STRING(0);
         PORT_DATA( PORT_NO ).NEXT_CHARACTER = 1;
         end; /* No buffers currently being printed */
      else
         do;  /* Add to end of list */
         TEMP_ADDRESS = B_ADDRESS;
         B_ADDRESS = PORT_DATA( PORT_NO ).LAST_BUFFER_ADDRESS;
         BUFFER.NEXT_BUFFER = TEMP_ADDRESS;
         PORT_DATA( PORT_NO ).LAST_BUFFER_ADDRESS = TEMP_ADDRESS;
         end; /* Add to end of list */

      enable;
      end; /* while LENGTH_OF_DATA > 0 */
   end WRITE;

$ eject
```

```
/*************************************************************************/
/*                                                                       */
/*     WRITE_POLL : Writes a character string pointed to by DATA_ADDRESS */
/*          to the port pointed to by PORT_NO.  The routine              */
/*          waits for all characters to be transmitted before returning. */
/*          Tx empty interrupts are disabled for the duration of the     */
/*          of the routine.                                              */
/*          The port is polled continually waiting for each character    */
/*          to be sent before the next on is output.                     */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*             A pointer to the user data string (DATA_ADDRESS)          */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

WRITE_POLL : procedure ( PORT_NO, DATA_ADDRESS ) public;

   declare PORT_NO  byte;
   declare B_ADDRESS pointer;
   declare BUFFER based B_ADDRESS structure
      ( OUTPUT_STRING(80) byte,
        NEXT_BUFFER       pointer  );

   /*-------------------------------------------*/

   declare I word;
   declare DATA_ADDRESS pointer;
   declare CHARACTER based DATA_ADDRESS(*) byte;

   /*-------------------------------------------*/

   /* Disable 8274 channel interrupts for duration of test */

   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 01H;
   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 04H; /* WR1 */

   /* If output buffers are allocated to the port, put them      */
   /* back on the free buffer list.                              */

   if PORT_DATA( PORT_NO ).BUFFER_ADDRESS <> nil then
      do;
      B_ADDRESS = LAST_FREE_BUFFER;
      BUFFER.NEXT_BUFFER = PORT_DATA( PORT_NO ).BUFFER_ADDRESS;
      LAST_FREE_BUFFER   = PORT_DATA( PORT_NO ).LAST_BUFFER_ADDRESS;
      PORT_DATA( PORT_NO ).BUFFER_ADDRESS = nil;
      PORT_DATA( PORT_NO ).LAST_BUFFER_ADDRESS = nil;
      end;
```

```
   I = 0;
   do while CHARACTER(I) <> EOM;
      do while(( input( PORT_DATA( PORT_NO ).CONTROL_PORT ) and 04H ) <> 04H );
         end;
      output( PORT_DATA( PORT_NO ).DATA_PORT ) = CHARACTER(I);
      I = I + 1;
      end;

   /* Wait for all characters to be transmitted */

   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 01H;
   do while(( input( PORT_DATA( PORT_NO ).CONTROL_PORT ) and 01H ) <> 01H );
      output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 01H;
      end;

   /* Re-enable Tx and Rx interrupts */

   if PORT_DATA( PORT_NO ).INTERRUPT_NO <> 0 then
      do;
      output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 001H;
      output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 01EH; /* WR1 */
      end;

   end WRITE_POLL;

$ eject
```

```
/*************************************************************************/
/*                                                                       */
/*     INPUT_CHARACTER : Clears the input buffer, then waits for a       */
/*         character to be received.  It is a function call.             */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*                                                                       */
/*     OUTPUT : the function call returns a character.                   */
/*                                                                       */
/*************************************************************************/

INPUT_CHARACTER: procedure ( PORT_NO ) byte public;

   declare PORT_NO  byte;

   /*-----------------------------------------------*/

   /* Disable Rx interrupts (otherwise they steal the character!) */

   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 01H;
   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 06H; /* WR1 */

   /* Clear the buffer in the 8274 */

   do while( input( PORT_DATA( PORT_NO ).CONTROL_PORT ) and 01H ) = 01H;
      PORT_DATA( PORT_NO ).INPUT_BUFFER(0) = input( PORT_DATA( PORT_NO ).DATA_PORT );
      end;
   call CLEAR_BUFFER( PORT_NO );

   /* Wait for and then get the character */

   do while( input( PORT_DATA( PORT_NO ).CONTROL_PORT ) and 01H ) <> 01H;
      end;
   PORT_DATA( PORT_NO ).INPUT_BUFFER(0) = input( PORT_DATA( PORT_NO ).DATA_PORT );

   /* Re-enable Rx interrupts */

   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 01H;
   output( PORT_DATA( PORT_NO ).CONTROL_PORT ) = 1EH; /* WR1 */

   return PORT_DATA( PORT_NO ).INPUT_BUFFER(0);

   end INPUT_CHARACTER;

$ eject
```

```
/*************************************************************************/
/*                                                                       */
/*     GET_CHARACTER : Returns the next character in the input buffer.   */
/*         If no characters are waiting then the buffer is cleared and   */
/*         the program waits for an input string terminated by a <CR>.   */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*                                                                       */
/*     OUTPUT : the function call returns a character.                   */
/*                                                                       */
/*************************************************************************/

GET_CHARACTER: procedure ( PORT_NO ) byte public;

   declare PORT_NO   byte;
   declare IN_CHAR byte;

   /*-------------------------------------------*/

   if PORT_DATA( PORT_NO ).MODE = TERMINAL then
      do;

      /* Wait for data to be stored in the buffer */

      do while ( PORT_DATA( PORT_NO ).INPUT_FLAG = 0 );
         end;

      IN_CHAR = PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).READ_POINTER );
      PORT_DATA( PORT_NO ).READ_POINTER = PORT_DATA( PORT_NO ).READ_POINTER + 1;

      /* If buffer is now empty, clear the Input Buffer */

      if PORT_DATA( PORT_NO ).NO_OF_CHARACTERS <= PORT_DATA( PORT_NO ).READ_POINTER then
         call CLEAR_BUFFER( PORT_NO );
      return IN_CHAR;
      end; /* PORT_MODE = TERMINAL */
```

```
   else
      do;  /* PORT_MODE = DATA */
      /* Check for underflow */
      if PORT_DATA( PORT_NO ).INPUT_FLAG = 0 then
         return 0;
      /* Get element */
      IN_CHAR = PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).READ_POINTER );
      /* Check if queue is empty */
      if PORT_DATA( PORT_NO ).NO_OF_CHARACTERS = PORT_DATA( PORT_NO ).READ_POINTER then
         call CLEAR_BUFFER( PORT_NO );
      else
         if PORT_DATA( PORT_NO ).READ_POINTER = 79 then
            PORT_DATA( PORT_NO ).READ_POINTER = 0;
         else
            PORT_DATA( PORT_NO ).READ_POINTER = PORT_DATA( PORT_NO ).READ_POINTER + 1;
      return IN_CHAR;
      end; /* PORT_MODE = DATA */

   end GET_CHARACTER;

$ eject
/*****************************************************************************/
/*                                                                           */
/*     LAST_CHARACTER_READ: Returns the content of the input buffer      */
/*         of the specified port.  It is a function call.                */
/*                                                                           */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*                                                                           */
/*     OUTPUT : the function call returns a character.                   */
/*                                                                           */
/*****************************************************************************/

LAST_CHARACTER_READ: procedure ( PORT_NO ) byte public;

   declare PORT_NO  byte;

   /*-------------------------------------------*/

   if PORT_DATA( PORT_NO ).INPUT_FLAG = 0 then
      return 0;
   else if PORT_DATA( PORT_NO ).INPUT_FLAG = 2 then
      return CR;
   else
      return PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).NO_OF_CHARACTERS - 1 );

   end LAST_CHARACTER_READ;

$ eject
```

```
/************************************************************************/
/*                                                                      */
/*     GET_BYTE : Clears the input buffer, waits for a <CR> and then    */
/*         returns the value of the 1st numeric string in the buffer.   */
/*                                                                      */
/*     INPUT : A pointer to the port data area.                         */
/*                                                                      */
/*     OUTPUT : the function call returns a numeric byte value.         */
/*                                                                      */
/************************************************************************/

GET_BYTE: procedure( PORT_NO, MIN_VALUE, MAX_VALUE ) byte public;

   declare PORT_NO        byte;
   declare MIN_VALUE      byte;
   declare MAX_VALUE      byte;
   declare NUMBER         byte;
   declare I              byte;
   declare INPUT_CORRECT  byte;
   declare NEXT_CHARACTER byte;

   /*-------------------------------------------*/

   INPUT_CORRECT = FALSE;
   MAIN_LOOP: do while INPUT_CORRECT = FALSE;

      /* Clear the input buffer */

      call CLEAR_BUFFER( PORT_NO );

      /* and wait for a <CR> */

      do while PORT_DATA( PORT_NO ).INPUT_FLAG <> 2;
         end;

      NEXT_CHARACTER = PORT_DATA( PORT_NO ).INPUT_BUFFER(0);

      /* Step over non-numeric characters */

      do while ( NEXT_CHARACTER < '0' or NEXT_CHARACTER > '9' ) and
            PORT_DATA( PORT_NO ).READ_POINTER < PORT_DATA( PORT_NO ).NO_OF_CHARACTERS;
         PORT_DATA( PORT_NO ).READ_POINTER = PORT_DATA( PORT_NO ).READ_POINTER + 1;
         NEXT_CHARACTER = PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).READ_POINTER );
         end;
```

```
      NUMBER = 0;
      do while NEXT_CHARACTER >= '0' and NEXT_CHARACTER <= '9' and
           PORT_DATA( PORT_NO ).READ_POINTER < PORT_DATA( PORT_NO ).NO_OF_CHARACTERS;
         if NUMBER <= 25 then
            do;
            NUMBER = NUMBER * 10 + ( NEXT_CHARACTER - '0' );
            INPUT_CORRECT = TRUE; /* If at least 1 numeric char received */
            end;
         else
            do;
            INPUT_CORRECT = FALSE;
            end;
         PORT_DATA( PORT_NO ).READ_POINTER = PORT_DATA( PORT_NO ).READ_POINTER + 1;
         NEXT_CHARACTER = PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).READ_POINTER );
         end;

      if INPUT_CORRECT = TRUE and
               NUMBER >= MIN_VALUE and NUMBER <= MAX_VALUE then
         return NUMBER;
      else
         do;
         INPUT_CORRECT = FALSE;
         do I = 1 to PORT_DATA( PORT_NO ).NO_OF_CHARACTERS;
            if PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).NO_OF_CHARACTERS - I ) >= ' '
               then call WRITE_POLL( PORT_NO, @RUBOUT_MESSAGE );
            end;
         call CLEAR_BUFFER( PORT_NO );
         end;
      end MAIN_LOOP;

   /* Allow other characters to come in */

   call CLEAR_BUFFER( PORT_NO );

   end GET_BYTE;

$ eject
```

```
/*************************************************************************/
/*                                                                       */
/*     GET_WORD : Clears the input buffer, waits for a <CR> and then     */
/*         returns the value of the 1st numeric string in the buffer.    */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*                                                                       */
/*     OUTPUT : the function call returns a numeric word value.          */
/*                                                                       */
/*************************************************************************/

GET_WORD: procedure( PORT_NO, MIN_VALUE, MAX_VALUE ) word public;

   declare PORT_NO         byte;
   declare MIN_VALUE       word;
   declare MAX_VALUE       word;
   declare NUMBER          word;
   declare I               byte;
   declare INPUT_CORRECT   byte;
   declare NEXT_CHARACTER  byte;

   /*-------------------------------------------*/

   INPUT_CORRECT = FALSE;

   MAIN_LOOP: do while INPUT_CORRECT = FALSE;

      /* Clear the input buffer */

      call CLEAR_BUFFER( PORT_NO );

      /* and wait for a <CR> */

      do while PORT_DATA( PORT_NO ).INPUT_FLAG <> 2;
         end;
      NEXT_CHARACTER = PORT_DATA( PORT_NO ).INPUT_BUFFER(0);

      /* Step over non-numeric characters */

      do while ( NEXT_CHARACTER < '0' or NEXT_CHARACTER > '9' ) and
            PORT_DATA( PORT_NO ).READ_POINTER < PORT_DATA( PORT_NO ).NO_OF_CHARACTERS;
         PORT_DATA( PORT_NO ).READ_POINTER = PORT_DATA( PORT_NO ).READ_POINTER + 1;
         NEXT_CHARACTER = PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).READ_POINTER );
         end;
```

```
      NUMBER = 0;
      do while NEXT_CHARACTER >= '0' and NEXT_CHARACTER <= '9' and
           PORT_DATA( PORT_NO ).READ_POINTER < PORT_DATA( PORT_NO ).NO_OF_CHARACTERS;
         if NUMBER <= 25 then
            do;
            NUMBER = NUMBER * 10 + ( NEXT_CHARACTER - '0' );
            INPUT_CORRECT = TRUE; /* If at least 1 numeric char received */
            end;
         else
            do;
            INPUT_CORRECT = FALSE;
            end;
         PORT_DATA( PORT_NO ).READ_POINTER = PORT_DATA( PORT_NO ).READ_POINTER + 1;
         NEXT_CHARACTER = PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).READ_POINTER );
         end;

      if INPUT_CORRECT = TRUE and
         NUMBER >= MIN_VALUE  and
         NUMBER <= MAX_VALUE then
         return NUMBER;
      else
         do;
         INPUT_CORRECT = FALSE;
         do I = 1 to PORT_DATA( PORT_NO ).NO_OF_CHARACTERS;
            if PORT_DATA( PORT_NO ).INPUT_BUFFER( PORT_DATA( PORT_NO ).NO_OF_CHARACTERS - I ) >= ' '
               then call WRITE_POLL( PORT_NO, @RUBOUT_MESSAGE );
            end;
         call CLEAR_BUFFER( PORT_NO );
         end;
      end MAIN_LOOP;

   /* Allow other characters to come in */

   call CLEAR_BUFFER( PORT_NO );
   end GET_WORD;

$ eject
```

```
/*************************************************************************/
/*                                                                       */
/*     BUFFER_READY: Returns 00H if the input buffer is empty.           */
/*         Returns 01H if data is available in the input buffer.         */
/*         Returns 02H if a <CR> has been received in the input buffer.  */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*                                                                       */
/*     OUTPUT : the function call returns a byte value.                  */
/*                                                                       */
/*************************************************************************/

BUFFER_READY: procedure( PORT_NO ) byte public;

   declare PORT_NO     byte;

   /*-------------------------------------------*/

   return PORT_DATA( PORT_NO ).INPUT_FLAG;

   end BUFFER_READY;

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

```
/*************************************************************************/
/*                                                                       */
/*                  MPSC interrupt handlers                              */
/*                                                                       */
/*************************************************************************/


CHB_TX_EMPTY:   procedure interrupt public;

   if MPSC_TEST_IN_PROGRESS = TRUE then /* MPSC BIT has been invoked */
      do;
      MPSC_TEST_RESULT = TRUE;
      output( PORT_DATA( PORT_B ).CONTROL_PORT ) = 28h;
      output( PORT_DATA( PORT_A  ).CONTROL_PORT ) = EOI_8274;
      call ISSUE_EOI( MPSC_INTERRUPT_NO );
      end;
   else
      call TRANSMIT_CHARACTER (PORT_B) ;

   end CHB_TX_EMPTY;

/*************************************************************************/

CHA_TX_EMPTY:   procedure interrupt public;

   call TRANSMIT_CHARACTER( PORT_A );
   end CHA_TX_EMPTY;

/*************************************************************************/

CHA_RX_AVAIL:   procedure interrupt public;

    call RECEIVE_CHARACTER( PORT_A );
    end CHA_RX_AVAIL;

/*************************************************************************/

CHB_RX_AVAIL:   procedure interrupt public;

    call RECEIVE_CHARACTER( PORT_B );
    end CHB_RX_AVAIL;

$ eject

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

```
CHA_RX_ERROR: procedure interrupt public;

    declare ( DUMMY_READ, STATUS ) byte;

    dummy_read = input( PORT_DATA(PORT_A).DATA_PORT );

    status = input( PORT_DATA(PORT_A).CONTROL_PORT ) and 70H;
    call MOVB( @REC_ERR_MESS, @VAR_MESSAGE, size( REC_ERR_MESS ) );
    VAR_MESSAGE(11) = 'A';
    call C050_BIN_BYTE_TO_ASCII_BIN ( status, @VAR_MESSAGE(30) );
    call WRITE( PORT_A, @VAR_MESSAGE );

    output( PORT_DATA( PORT_A ).CONTROL_PORT ) = ERROR_RESET_CODE;

    output( PORT_DATA( PORT_A ).CONTROL_PORT ) = EOI_8274;
    call ISSUE_EOI( MPSC_INTERRUPT_NO );

    end CHA_RX_ERROR;

/*****************************************************************************/

CHB_RX_ERROR: procedure interrupt public;

   declare ( DUMMY_READ, STATUS ) byte;

    dummy_read = input( PORT_DATA( PORT_B ).DATA_PORT );

    status = input( PORT_DATA( PORT_B ).CONTROL_PORT ) and 70H;
    call MOVB( @REC_ERR_MESS, @VAR_MESSAGE, size( REC_ERR_MESS ) );
    VAR_MESSAGE(11) = 'B';
    call C050_BIN_BYTE_TO_ASCII_BIN ( status, @VAR_MESSAGE(30) );

    call write( PORT_B, @VAR_MESSAGE );

    output( PORT_DATA( PORT_B ).CONTROL_PORT ) = ERROR_RESET_CODE;

    output( PORT_DATA( PORT_A ).CONTROL_PORT ) = EOI_8274;
    call ISSUE_EOI( MPSC_INTERRUPT_NO );

    end CHB_RX_ERROR;


end io_8274;
```

2.5.1.10 STDCONVT.PLM

```
$ include (PLMPAR.INC)

/*****************************************************************************
 *                                                                           *
 *        MODULE NAME : CONV                                                 *
 *                                                                           *
 *****************************************************************************
 *                                                                           *
 *   Source Filename  : CONVERT.PLM                                          *
 *                                                                           *
 *   Source Compiler  : PLM86                                                *
 *                                                                           *
 *   Operating System : DOS 3.10                                             *
 *                                                                           *
 *   Description      : Subroutines to perform various conversions           *
 *                                                                           *
 *   Public procedures: C010_BIN_WORD_TO_ASCII_OCTAL                         *
 *                      C020_BIN_BYTE_TO_ASCII_OCTAL                         *
 *                      C030_BIN_WORD_TO_ASCII_DEC                           *
 *                      C040_BIN_BYTE_TO_ASCII_DEC                           *
 *                      C050_BIN_BYTE_TO_ASCII_BIN                           *
 *                      C060_BIN_WORD_TO_BCD_VAL                             *
 *                      C070_CONVERT_WORD_TO_HEX                             *
 *                                                                           *
 *   EPD files        : None                                                 *
 *                                                                           *
 *   Include files    : None                                                 *
 *                                                                           *
 *                                                                           *
 *****************************************************************************

$ eject

 *****************************************************************************
 *                                                                           *
 *   HISTORY   Version 1.0 :                                                 *
 *                                                                           *
 *             Designed by : P.A. OLANDER    Date :  September 1988        *
 *             Description : Original                                        *
 *                                                                           *
 *                                                                           *
 *****************************************************************************/
```

```
CONV:                do;

/*************************************************************************/
/*                  CONVERT A BINARY WORD TO ASCII OCTAL                 */
/*************************************************************************/

C010_BIN_WORD_TO_ASCII_OCTAL: Procedure(BINARY_NUMBER,ASCII_ARRAY_PTR) public;

    declare BINARY_NUMBER word;
    declare ASCII_ARRAY_PTR pointer;
    declare CHARACTER based ASCII_ARRAY_PTR(1) byte;
    declare I byte;
    declare DIGIT byte;

    do I = 0 to 5;
       DIGIT = BINARY_NUMBER mod 8;
       CHARACTER(5-I) = DIGIT + 030h;
       BINARY_NUMBER = BINARY_NUMBER/8;
       end ;

    /* Replace leading zeroes with spaces */

    I = 0;
    do while I < 5 and CHARACTER(I) = '0';
       CHARACTER(I) = ' ';
       I = I + 1;
       end;

    end C010_BIN_WORD_TO_ASCII_OCTAL;

/*************************************************************************/
/*                  CONVERT A BINARY BYTE TO ASCII OCTAL                 */
/*************************************************************************/

C020_BIN_BYTE_TO_ASCII_OCTAL: Procedure (BINARY_NUMBER,ASCII_ARRAY_PTR) public;

   declare BINARY_NUMBER byte;
   declare ASCII_ARRAY_PTR pointer;
   declare CHARACTER based ASCII_ARRAY_PTR(1) byte;
   declare I byte;
   declare DIGIT byte;

   do I = 0 to 2;
      DIGIT = BINARY_NUMBER mod 8;
      CHARACTER(2-I) = DIGIT + 030h;
      BINARY_NUMBER = BINARY_NUMBER/8;
      end ;

   end C020_BIN_BYTE_TO_ASCII_OCTAL;
```

```
/*************************************************************************/
/*                  CONVERT A BINARY WORD TO ASCII DECIMAL                */
/*************************************************************************/

C030_BIN_WORD_TO_ASCII_DEC: Procedure (BINARY_NUMBER,ASCII_ARRAY_PTR) public;

   declare BINARY_NUMBER word;
   declare ASCII_ARRAY_PTR pointer;
   declare CHARACTER based ASCII_ARRAY_PTR(1) byte;
   declare I byte;
   declare DIGIT byte;

   do I = 0 to 5;
      DIGIT = BINARY_NUMBER mod 10;
      CHARACTER(5-I) = DIGIT + 030h;
      BINARY_NUMBER = BINARY_NUMBER/10;
      end ;

   /* Replace leading zeroes with spaces */

   I = 0;
   do while I < 5 and CHARACTER(I) = '0';
      CHARACTER(I) = ' ';
      I = I + 1;
      end;

   end C030_BIN_WORD_TO_ASCII_DEC;
```

```
/*************************************************************************/
/*                  CONVERT A BINARY BYTE TO ASCII DECIMAL               */
/*************************************************************************/

C040_BIN_BYTE_TO_ASCII_DEC: Procedure( BINARY_NUMBER, ASCII_ARRAY_PTR ) public;

    declare BINARY_NUMBER byte;
    declare ASCII_ARRAY_PTR pointer;
    declare CHARACTER based ASCII_ARRAY_PTR(1) byte;
    declare I byte;
    declare DIGIT byte;

    do I = 0 to 2;
       DIGIT = BINARY_NUMBER mod 10;
       CHARACTER(2-I) = DIGIT + 030h;
       BINARY_NUMBER = BINARY_NUMBER/10;
       end ;

    /* Replace leading zeroes with spaces */

    I = 0;
    do while I < 2 and CHARACTER(I) = '0';
       CHARACTER(I) = ' ';
       I = I + 1;
       end;

    end C040_BIN_BYTE_TO_ASCII_DEC;


/*************************************************************************/
/*                    CONVERT A BINARY BYTE TO ASCII BINARY              */
/*************************************************************************/

C050_BIN_BYTE_TO_ASCII_BIN: Procedure (BINARY_NUMBER,ASCII_ARRAY_PTR) public;

    declare BINARY_NUMBER byte;
    declare ASCII_ARRAY_PTR pointer;
    declare CHARACTER based ASCII_ARRAY_PTR(1) byte;
    declare I byte;
    declare DIGIT byte;

    do I = 0 to 7;
       DIGIT = BINARY_NUMBER mod 2;
       CHARACTER(7-I) = DIGIT + 030h;
       BINARY_NUMBER = BINARY_NUMBER/2;
       end ;

    end C050_BIN_BYTE_TO_ASCII_BIN;
```

```
/*************************************************************************/
/*            CONVERT A WORD VALUE TO  4-DIGIT BCD VALUE                 */
/*************************************************************************/

C060_WORD_TO_BCD_VAL: Procedure(BINARY_NUMBER) word public;

   declare (BCD_VALUE,DIGIT) word;
   declare (i,BINARY_NUMBER) byte;

   BCD_VALUE = 00;

   do I = 0 to 1;
      DIGIT = BINARY_NUMBER mod 10;
      BCD_VALUE = BCD_VALUE or rol(DIGIT,8*I);
      BINARY_NUMBER = BINARY_NUMBER/10;
      end ;

   return BCD_VALUE;

   end C060_WORD_TO_BCD_VAL;

/*************************************************************************/
/*       CONVERT A WORD VALUE TO 4 ASCII CHARACTERS IN HEX FORM          */
/*************************************************************************/

C070_CONVERT_WORD_TO_HEX: Procedure( INPUT_WORD, ASCII_ADDRESS ) public;

   declare INPUT_WORD word;
   declare ASCII_ADDRESS pointer;

   declare ( CHARACTER based ASCII_ADDRESS )(*) byte;

   declare ASCII_VALUE(*) byte data ( '0123456789ABCDEF' );
   declare I byte;

   do I = 0 to 3;
      CHARACTER( 3-I ) = ASCII_VALUE( INPUT_WORD mod 16 );
      INPUT_WORD = INPUT_WORD / 16;
      end;

   end C070_CONVERT_WORD_TO_HEX;

end CONV;
```

2.5.2 ASM86 files

2.5.2.1 STDCPU.ASM

```
$ mod186
$ debug
$ xref

;  /*************************************************************************
;   *                                                                       *
;   *        MODULE NAME : TEST_CPU                                         *
;   *                                                                       *
;   *************************************************************************
;   *                                                                       *
;   *   Source Filename  : STDCPU.ASM                                       *
;   *                                                                       *
;   *   Source Compiler  : ASM86                                            *
;   *                                                                       *
;   *   Operating System : DOS 3.10                                         *
;   *                                                                       *
;   *   Description      :  Tests the 8086 microprocessor family for :      *
;   *                                                                       *
;   *                         a) general purpose data movement              *
;   *                         b) shifts and rotates                         *
;   *                         c) arithemetic ability and                    *
;   *                         d) stack manipulation                         *
;   *                                                                       *
;   *   Public procedures: CPU_TEST                                         *
;   *                                                                       *
;   *   EPD files        : None                                             *
;   *                                                                       *
;   *   Include files    : None                                             *
;   *                                                                       *
;   *                                                                       *
;   *************************************************************************

$ eject

;   *************************************************************************
;   *                                                                       *
;   *   HISTORY   Version 1.0 :                                             *
;   *                                                                       *
;   *             Designed by : P.A. OLANDER    Date :  May 1989          *
;   *             Description : Original                                    *
;   *                                                                       *
;   *                                                                       *
;   *************************************************************************/

$ eject
```

```
NAME        TEST_CPU

ZERO        equ   0000H              ; All zeros
ONES        equ   0FFFFH             ; All ones
ZONE        equ   0101010101010101B  ; Alternating zero and ones
ONEZ        equ   1010101010101010B  ; Alternating ones and zeros
ZONEL       equ   01010101B          ; Byte of 0's and 1's
ZONEH       equ   01010101B          ; Byte of 0's and 1's
ONEZL       equ   10101010B          ; Byte of 1's and 0's

STACK       segment stack 'STACK'
            dw 100 dup (?)
STACK       ends

; Test general purpose data movement

CODE        segment public 'CODE'
            public  CPU_TEST
            assume  CS:CODE,SS:STACK
CPU_TEST    proc    far
            mov     AX,STACK         ; Set up stack segment
            mov     SS,AX
            mov     AX,ZONE          ; A = 0101 0101 0101 0101
            mov     BX,ONEZ          ; B = 1010 1010 1010 1010
            mov     CX,AX            ; C = 0101 0101 0101 0101
            mov     DX,BX            ; D = 1010 1010 1010 1010
            add     AX,BX            ; A = 1111 1111 1111 1111
            or      CX,DX            ; C = 1111 1111 1111 1111
            cmp     AX,CX            ; A = C ?
            je      STATUS           ; Continue testing
            jmp     ERROR            ; If A <> C then ERROR

; Test status register

;    1) Test zero flag

STATUS:     xor     AX,AX            ; A = 0
            cmp     AX,1             ; A = 1 ?
            jz      FAULT            ; If ZFL = TRUE then ERROR
            inc     AX               ; A = 1
            cmp     AX,1             ; A = 1 ?
            jnz     FAULT            ; If ZFL = FALSE then ERROR
```

```
;     2) Test carry and overflow flags

              mov     AX,ONES                 ; A = FFFF
              mov     DX,ONES                 ; D = FFFF
              mov     BX,ONES                 ; B = FFFF
              mul     BX                      ; CFL = TRUE and OFL = TRUE
              jnc     FAULT                   ; If CFL = FALSE then ERROR
              jno     FAULT                   ; If OFL = FALSE then ERROR

;     3) Test parity flag

              xor     AX,AX                   ; A = 0000 0000 0000 0000
              or      AX,1                    ; A = 0000 0000 0000 0001
              jp      FAULT                   ; Odd parity means PFL = FALSE

;     4) Test sign flag

              neg     AX                      ; A = FFFF and SFL = TRUE
              jns     FAULT                   ; If SFL = FALSE then ERROR
              jmp     NEXT                    ; Continue testing
FAULT:        jmp     ERROR                   ; Report that CPU test failed

; Test shift and rotates

NEXT:         clc                             ; Clear carry bit
              mov     AX,ZONE                 ; A = 0101 0101 0101 0101
              mov     BX,ONEZ                 ; B = 1010 1010 1010 1010
              sal     AX,1                    ; A = 1010 1010 1010 1010
              cmp     AX,BX                   ; A = B ?
              jne     ERROR                   ; If A <> B then ERROR
              rcl     BX,1                    ; B = 0101 0101 0101 0100
              jnc     ERROR                   ; Carry = 1
              rcl     BX,1                    ; B = 1010 1010 1010 1001
              mov     DH,ZONEH                ; D = 0101 0101 1010 1010
              stc                             ; Carry = 1
              rcr     DL,6                    ; D = 0101 0101 0101 0110
              or      BX,DX                   ; B = 1111 1111 1111 1111
              cmp     BX,CX                   ; B = C ?
              jne     ERROR                   ; If B <> C then ERROR
```

```
; Test arithmetic ability

;    1) Multiplication

            xor     BX,BX               ; Clear B register
            mul     BX                  ; AX * BX = DXAX
            cmp     AX,ZERO             ; Low order result is in A
            jne     ERROR               ; If A <> 0 then ERROR
            cmp     DX,ZERO             ; High order result is in B
            jne     ERROR               ; If D <> 0 then ERROR
            mov     AL,ZONEL            ; A = 01010101
            mov     BL,2                ; B = 2
            mul     BL                  ; Result is in AX
            cmp     AL,ONEZL            ; AL = 10101010 ?
            jne     ERROR

;    2) Division

            mov     BL,2                ; BL = 02 Hex
            xor     AH,AH               ; AH = 00 Hex
            mov     AL,ONEZL            ; AL = 1010 1010
            mov     CL,ZONEL            ; CL = 0101 0101 - Half of AL
            div     BL                  ; Divide accumulator by two
            cmp     AL,CL               ; AL divided by 2 = CL
            jne     ERROR
```

```
; Test the stack

                mov     AX,ZONE                 ; A = 0101 0101 0101 0101
                mov     BX,ONEZ                 ; B = 1010 1010 1010 1010
                mov     CX,ZONE                 ; C = 0101 0101 0101 0101
                mov     DX,ONEZ                 ; D = 1010 1010 1010 1010
                push    AX
                push    BX
                push    CX
                push    DX                      ; Save register values
                push    BP                      ; Equivalent to 286 pusha instr
                push    SI
                push    DI
                xor     AX,AX                   ; A = 0
                xor     BX,BX                   ; B = 0
                xor     CX,CX                   ; C = 0
                xor     DX,DX                   ; D = 0
                pop     DI
                pop     SI
                pop     BP
                pop     DX                      ; Restore register values
                pop     CX                      ; Equivalent to 286 popa instr
                pop     BX
                pop     AX
                cmp     AX,ZONE                 ; A = 0101 0101 0101 0101
                jne     ERROR                   ; If not then ERROR
                cmp     BX,ONEZ                 ; B = 1010 1010 1010 1010
                jne     ERROR                   ; If not then ERROR
                cmp     CX,ZONE                 ; C = 0101 0101 0101 0101
                jne     ERROR                   ; If not then ERROR
                cmp     DX,ONEZ                 ; D = 1010 1010 1010 1010
                jne     ERROR                   ; If not then ERROR
                mov     AL,1                    ; Return a one in accumulator
                jmp     FINISH                  ; Don't execute ERROR code
ERROR:          mov     AL,2                    ; Return a two in accumulator
FINISH:         ret
CPU_TEST        endp
CODE            ends
                end
```

2.5.2.2 STDRAM.ASM

```
$ debug
$ xref

;  /*****************************************************************************
;   *                                                                           *
;   *        MODULE NAME : TEST_RAM                                             *
;   *                                                                           *
;   *****************************************************************************
;   *                                                                           *
;   *   Source Filename  : STDRAM.ASM                                           *
;   *                                                                           *
;   *   Source Compiler  : ASM86                                                *
;   *                                                                           *
;   *   Operating System : DOS 3.10                                             *
;   *                                                                           *
;   *   Description      : Tests the CPU card on-board RAM.                     *
;   *                                                                           *
;   *   Public procedures: MEM_TEST                                             *
;   *                                                                           *
;   *   EPD files        : None                                                 *
;   *                                                                           *
;   *   Include files    : None                                                 *
;   *                                                                           *
;   *                                                                           *
;   *****************************************************************************

$ eject

;   *****************************************************************************
;   *                                                                           *
;   *   HISTORY   Version 1.0 :                                                 *
;   *                                                                           *
;   *             Designed by : P.A. OLANDER    Date :  May 1989              *
;   *             Description : Original                                        *
;   *                                                                           *
;   *                                                                           *
;   *****************************************************************************/
```

```
NAME            TEST_RAM

ZONE            equ    055h         ; 01010101 binary
ONEZ            equ    0AAh         ; 10101010 binary
FAULT           equ      2          ; A reg contains this if an error occurs
BOTTOM          equ      0h         ; Bottom of memory block
TOP             equ  0dfffh         ; Top of memory block
                                    ; Stack resides from 0E000h to FFFFh

; On-board RAM :

RAM_BLOCK       segment public 'RAM_BLOCK'
MEM             db 0ffffh dup (?)
RAM_BLOCK       ends

; BIT results (0 = Untested; 1 = Passed; 2 = Failed; 8 = Absent) :

TEST_RESULTS    struc
CPU_BOARD       db ?
CPU             db ?
ROM             db ?
MASTER_PIC      db ?
SLAVE_PIC       db ?
TMOUT_CIRCUIT   db ?
PIT             db ?
COPROCESSOR     db ?
MPSC            db ?
RAM             db ?
ERAM            db ?
EROM            db ?
SDB_BOARD       db ?
SDB_SELFTEST    db ?
SDB_RAM         db ?
TEST_RESULTS    ends

BIT_RESULTS_SEG segment public 'BIT_RESULTS_SEG'
BIT_RESULT      TEST_RESULTS <>
public          BIT_RESULT
BIT_RESULTS_SEG ends

; The following segment is located absolutely in order
; to interface with protected mode code :

NO_OF_RUNS_SEG  segment
PROCESSOR_ID    dw ?
NUMBER_OF_RUNS  dw ?
public          NUMBER_OF_RUNS
public          PROCESSOR_ID
NO_OF_RUNS_SEG  ends
```

```
; Beginning of RAM test code :

CODE            segment public 'CODE'
                assume CS:CODE,ES:RAM_BLOCK
                public MEM_TEST
MEM_TEST        proc   far
START:          push   ES
                push   DS
                mov    AX,RAM_BLOCK  ; Set up RAM_BLOCK segment
                mov    ES,AX
                xor    BX,BX         ; Clear B reg
                mov    CX,BOTTOM     ; Load bottom of memory into C reg
AGAIN:          mov    SI,CX         ; C reg contains index to both
                mov    DI,CX         ; source and destination
                mov    BL, MEM[SI]   ; Save original contents
                mov    AL,ZONE       ; A   = 01010101
                mov    MEM[DI], AL   ; RAM = 01010101
                mov    DL, MEM[SI]   ; D   = RAM
                cmp    DL,ZONE       ; D   = 01010101 ?
                jne    ERROR         ; If not, then RAM is faulty
                mov    AL,ONEZ       ; A   = 10101010
                mov    MEM[DI], AL   ; RAM = 10101010
                mov    DL, MEM[SI]   ; D   = RAM
                cmp    DL,ONEZ       ; D   = 10101010 ?
                jne    ERROR         ; If not, then RAM is faulty
                mov    MEM[DI], BL   ; Restore original contents
                inc    CX            ; Increment memory pointer
                cmp    CX,TOP        ; Pointer = Top of RAM ?
                jne    AGAIN         ; Test next location until C = 0
                mov    AL,1          ; Result is one in A reg if test passes
                jmp    RETURN        ; C reg = Top of RAM, so return
ERROR:          mov    AL,FAULT      ; A reg = 2 if test fails
RETURN:         pop    DS            ; Restore status
                pop    ES
                ret
MEM_TEST        endp
CODE            ends
                end
```

2.5.2.3 STDERAM.ASM

```
$ debug
$ xref

;  /*****************************************************************************
;   *                                                                           *
;   *        MODULE NAME : RAM_1611_TEST                                        *
;   *                                                                           *
;   *****************************************************************************
;   *                                                                           *
;   *   Source Filename  : STDERAM.ASM                                          *
;   *                                                                           *
;   *   Source Compiler  : ASM86                                                *
;   *                                                                           *
;   *   Operating System : DOS 3.10                                             *
;   *                                                                           *
;   *   Description      : Procedure ERAM_TEST :                                *
;   *                                                                           *
;   *                      Tests the EPROM/RAM card RAM in a range of           *
;   *                      system specific addresses as specified by the        *
;   *                      option programmed into the system description.       *
;   *                                                                           *
;   *                      Procedure REAL_MODE_CODE :                           *
;   *                                                                           *
;   *                      Provides the link (following the succesful           *
;   *                      completion of POST) between the standardised         *
;   *                      code and the subsystem-specific applications         *
;   *                      code. The address of the start of the                *
;   *                      subsystem-specific applications code is provided     *
;   *                      in the system description.                           *
;   *                                                                           *
;   *   Public procedures: ERAM_TEST                                            *
;   *                      REAL_MODE_CODE                                       *
;   *                                                                           *
;   *   EPD files        : None                                                 *
;   *                                                                           *
;   *   Include files    : None                                                 *
;   *                                                                           *
;   *                                                                           *
;   *****************************************************************************

$ eject
```

```
;   ************************************************************************
;   *                                                                      *
;   *   HISTORY   Version 1.0 :                                            *
;   *                                                                      *
;   *             Designed by : P.A. OLANDER    Date :  May 1989         *
;   *             Description : Original                                   *
;   *                                                                      *
;   *                                                                      *
;   ************************************************************************/

NAME            RAM_1611_TEST

ZONE            equ    055h         ; 01010101 binary
ONEZ            equ    0AAh         ; 10101010 binary
PASSED          equ      1          ; A reg contains this if test passes
FAILED          equ      2          ; A reg contains this if an error occurs
BOTTOM          equ      0h         ; Bottom of segment
TOP             equ  0ffffh         ; Top of segment
RAM_BLOCK_1     equ   4000h         ; Segment address of first block
SEGMENT_LENGTH  equ   1000h         ; Length of each block in selector terms

; In addition to providing the standardised code with useful information
; relating to the system resident on the EPROM/RAM card, the system
; description also provides interfacing subsystems the facility to
; reconfigure certain aspects of the standard computing segment.

SYSTEM_DESC         segment public 'SYSTEM_DESC'
EPROM_ID            db ?            ; For POST to recognise EPROM presence
SYSTEM_NUMBER       db ?            ; Defined as per subsystem
SYSTEM_VERSION      db ?            ; Subsystem applications iteration
EPROM_RAM_OPTION    db ?            ; Defined as per CPU card standard
CSUMS_PRESENT       db ?            ; 1 = Both ROM checksums present
NO_OF_PROCS         db ?            ; 1 = Single processor; 2 = Dual processor
MODE                db ?            ; 5 = Real Mode; 0ah = Protected Mode
PMODE_VER           db ?            ; 1 = Final Version,
RMODE_CODE          dd ?            ; Absolute EPROM location of code to
                                    ; run Real Mode Applications
DESC_FILLER         db 0f2h dup (?) ; Spare locations
EPROM_CHECKSUM      dw ?            ; As calculated by CRC-CCITT formula
public              EPROM_ID
public              SYSTEM_NUMBER
public              SYSTEM_VERSION
public              EPROM_RAM_OPTION
public              CSUMS_PRESENT
public              NO_OF_PROCS
public              MODE
public              PMODE_VER
public              RMODE_CODE
public              EPROM_CHECKSUM
SYSTEM_DESC         ends
```

```
; ROM blocks on the EPROM/RAM card :

EROM_BLOCK_1    segment public 'EROM_BLOCK_1'
ROM_1           db 0ffffh dup (?)
public          ROM_1
EROM_BLOCK_1    ends

EROM_BLOCK_2    segment public 'EROM_BLOCK_2'
ROM_2           db 0ffffh dup (?)
public          ROM_2
EROM_BLOCK_2    ends

EROM_BLOCK_3    segment public 'EROM_BLOCK_3'
ROM_3           db 0ffffh dup (?)
public          ROM_3
EROM_BLOCK_3    ends

EROM_BLOCK_4    segment public 'EROM_BLOCK_4'
ROM_4           db 0ffffh dup (?)
public          ROM_4
EROM_BLOCK_4    ends

EROM_BLOCK_5    segment public 'EROM_BLOCK_5'
ROM_5           db 0ffffh dup (?)
public          ROM_5
EROM_BLOCK_5    ends

EROM_BLOCK_6    segment public 'EROM_BLOCK_6'
ROM_6           db 0ffffh dup (?)
public          ROM_6
EROM_BLOCK_6    ends

EROM_BLOCK_7    segment public 'EROM_BLOCK_7'
ROM_7           db 0ffffh dup (?)
public          ROM_7
EROM_BLOCK_7    ends

EROM_BLOCK_8    segment public 'EROM_BLOCK_8'
ROM_8           db 0ffffh dup (?)
public          ROM_8
EROM_BLOCK_8    ends
```

```
; ROM blocks on the CPU card :

EROM_BLOCK_9      segment public 'EROM_BLOCK_9'
ROM_9             db 0ffffh dup (?)
public            ROM_9
EROM_BLOCK_9      ends

EROM_BLOCK_10     segment public 'EROM_BLOCK_10'
ROM_10            db 0fffch dup (?)
OB_EPROM_CHECKSUM dw ?
OB_CSUM_PRESENT   db ?
public            OB_EPROM_CHECKSUM
public            OB_CSUM_PRESENT
public            ROM_10
EROM_BLOCK_10     ends

; Local variables

DATA              segment public 'DATA'
TEST_SEGMENT      dw ?
E_R_OPTION        db ?
BLOCK_NO          db ?
RESULT            db ?
DATA              ends
```

```
CODE            segment public 'CODE'
                assume CS:CODE,DS:DATA
                public ERAM_TEST

; EPROM/RAM RAM test starts here :

ERAM_TEST       proc   far
START:          push   ES
                push   DS
                mov    AX,DATA                  ; Set up data segment
                mov    DS,AX
                mov    AX,SYSTEM_DESC           ; Set up ES to point
                mov    ES,AX                    ; to SYSTEM_DESCRIPTION
                mov    DS:RESULT,0              ; Initially untested
                mov    AL,ES:EPROM_RAM_OPTION   ; Multiply the EPROM/RAM
                shl    AL,1                     ; option by 2 and then store
                mov    DS:E_R_OPTION,AL         ; value in the data segment
                xor    BX,BX                    ; Clear B reg
                xor    DX,DX                    ; Clear D reg
                mov    BLOCK_NO,1               ; Initialise block number
                mov    TEST_SEGMENT,RAM_BLOCK_1 ; Point to first RAM block

NEXT_BLOCK:     mov    AX,TEST_SEGMENT          ; Set ES to point to
                mov    ES,AX                    ; RAM block under test
                call   TEST_BLOCK               ; Test the block of RAM
                cmp    RESULT,FAILED            ; If the test has failed
                je     RETURN                   ; then return with result
                mov    AL,BLOCK_NO              ; A = Number of blocks tested
                cmp    AL,E_R_OPTION            ; If test not yet finished then
                jbe    NEXT_BLOCK               ; test the next 64K block
                mov    RESULT,PASSED            ; else test has passed

RETURN:         mov    AL,RESULT
                pop    DS
                pop    ES
                ret

ERAM_TEST       endp
```

```
TEST_BLOCK      proc near
                mov     CX, TOP
                xor     SI, SI
                xor     DI, DI

AGAIN:          mov     BL,ES:[SI]          ; Save original contents
                mov     AL,ZONE             ; A    = 01010101
                mov     ES:[DI], AL         ; ERAM = 01010101
                mov     DL, ES:[SI]         ; D    = ERAM
                cmp     DL,ZONE             ; D    = 01010101 ?
                jne     ERROR               ; If not, then ERAM is faulty
                mov     AL,ONEZ             ; A    = 10101010
                mov     ES:[DI], AL         ; ERAM = 10101010
                mov     DL, ES:[SI]         ; D    = ERAM
                cmp     DL,ONEZ             ; D    = 10101010 ?
                jne     ERROR               ; If not, then ERAM is faulty
                mov     ES:[DI],BL          ; Restore original contents
                inc     SI                  ; Increment memory pointer
                inc     DI                  ; Increment memory pointer
                loop    AGAIN               ; Test next location until C = 0

                inc     BLOCK_NO            ; Look at next 64K block
                add     TEST_SEGMENT,SEGMENT_LENGTH
                jmp     AROUND_ERROR        ; This block has passed

ERROR:          mov     RESULT,FAILED       ; RESULT = 2 if test fails
AROUND_ERROR:   ret
TEST_BLOCK      endp

; Pass control to the subsystem in real mode

public          REAL_MODE_CODE
REAL_MODE_CODE: nop
                mov AX,SYSTEM_DESC
                mov DS,AX
REAL_MODE:      jmp DS:dword ptr RMODE_CODE

; Absolute jump to predefined address (CS = ???? and IP = ????)

CODE            ends
                end
```

2.5.2.4 STDSDB.ASM

```
$ debug
$ xref

;  /*****************************************************************************
;   *                                                                           *
;   *        MODULE NAME : SDB_INTERFACE                                        *
;   *                                                                           *
;   *****************************************************************************
;   *                                                                           *
;   *   Source Filename  : STDSDB.ASM                                           *
;   *                                                                           *
;   *   Source Compiler  : ASM86                                                *
;   *                                                                           *
;   *   Operating System : DOS 3.10                                             *
;   *                                                                           *
;   *   Description      :  Provides a segment to enable the SDB interface      *
;   *                       to be absolutely located.                           *
;   *                                                                           *
;   *   Public procedures: None                                                 *
;   *                                                                           *
;   *   EPD files        : None                                                 *
;   *                                                                           *
;   *   Include files    : None                                                 *
;   *                                                                           *
;   *                                                                           *
;   *****************************************************************************

$ eject
```

```
;   ************************************************************************
;   *                                                                      *
;   *   HISTORY   Version 1.0 :                                            *
;   *                                                                      *
;   *             Designed by : P.A. OLANDER    Date :  May 1989         *
;   *             Description : Original                                   *
;   *                                                                      *
;   *                                                                      *
;   ************************************************************************/

NAME            SDB_INTERFACE

SDB_BUFFER          segment public 'SDB_BUFFER'
SDB_TEST_STATUS     dw ?
SDB_TEST_NUMBER     dw ?
SDB_TEST_PARAMETER_1 dw ?
SDB_TEST_PARAMETER_2 dw ?
SDB_TEST_PARAMETER_3 dw ?
SDB_FILLER          dw 0bfbh dup (?)
SDB_RAM_BUFFER      dw 6fffh dup (?)
public          SDB_TEST_STATUS
public          SDB_TEST_NUMBER
public          SDB_TEST_PARAMETER_1
public          SDB_TEST_PARAMETER_2
public          SDB_TEST_PARAMETER_3
public          SDB_RAM_BUFFER
SDB_BUFFER      ends

                end
```

2.5.3 Entities publicly declared files

2.5.3.1 STDINIT.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDINIT.EPD contains the corresponding external declarations of           *
 * all entities declared public in STDINIT.PLM.                              *
 *                                                                           *
 *****************************************************************************/

INITIALISE_ALL: procedure external;
   end INITIALISE_ALL;

PRINT_RESULTS: procedure external;
   end PRINT_RESULTS;
```

2.5.3.2 STDBIT.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDBIT.EPD contains the corresponding external declarations of            *
 * all entities declared public in STDBIT.PLM.                               *
 *                                                                           *
 *****************************************************************************/

/*****************************************************************************
 *                                                                           *
 *                          Global variables                                 *
 *                                                                           *
 *****************************************************************************/

declare CARD_UNDER_TEST       byte external;
declare ON_BOARD_VAR          word external;
declare OB_EPROM_CHECK        word external;
declare EPROM_CHECK           word external;
declare TEST                  byte external;

/*****************************************************************************
 *                                                                           *
 * The TEST flag is set when tests are executed, so that the processor can   *
 * expect self-generated time-outs. These time-outs are necessary in order   *
 * to test the functionality of certain aspects of the hardware.             *
 *                                                                           *
 *****************************************************************************/
```

```
CRC_RESULT: procedure (REM, INPUT_BYTE) word external;

   declare REM          word;
   declare INPUT_BYTE   byte;

   end CRC_RESULT;

TEST_ROM_BLOCK: procedure (ROM_BLOCK_NO) word external;

   declare ROM_BLOCK_NO byte;

   end TEST_ROM_BLOCK;

OFF_BOARD_ACCESS: procedure (ACTION, OFF_BOARD_VAR_PTR, VALUE_TO_WRITE) external;

 /*****************************************************************************
 *                                                                           *
 * This procedure attempts off-board read/writes at a location passed as a   *
 * parameter. If the CPU card times-out when attempting this access and      *
 * the bus had not been granted to the card, then it attempts to access the  *
 * location again. On time-outs with no bus grant, it will keep trying to    *
 * access the location until the pre-defined maximum number of retries is    *
 * exceeded.                                                                 *
 *                                                                           *
 *****************************************************************************/

   declare ACTION                                word ;
   declare VALUE_TO_WRITE                        word ;
   declare OFF_BOARD_VAR_PTR                   pointer ;
   declare OFF_BOARD_VAR based OFF_BOARD_VAR_PTR    word ;

   end OFF_BOARD_ACCESS;

PRINT_ERROR: procedure external;

 /*****************************************************************************
 *                                                                           *
 * This procedure will output the a message if the EPROM/RAM card does       *
 * does not contain the required information in ROM and will then cause the  *
 * processor to halt.                                                        *
 *                                                                           *
 *****************************************************************************/

   end PRINT_ERROR;
```

```
TEST_CORE: procedure external;

/*****************************************************************************
 *                                                                           *
 * Tests the basic operating kernel on the CPU board in order to provide the *
 * operator with confidence in the integrity of the system. If any of these  *
 * tests fail, it is considered to be a critical failure. The tests for the  *
 * basic kernel are a microprocessor and stack test, an on-board ROM test    *
 * and an on-board RAM test.                                                 *
 *                                                                           *
 *****************************************************************************/

   end TEST_CORE;

ON_BOARD_EPROM_TEST: procedure external;

   /*****************************************************************************
    *                                                                           *
    * This procedure calculates a ROM checksum according to the following       *
    * method ( ref. "COMPUTER NETWORKS", A. TANENBAUM, p. 128 - 132 ) :         *
    *                                                                           *
    * It considers the ROM based code to be one binary number                   *
    * and then by using the CRC-CCITT polynomial X**16 + X**12 + X**5 + 1       *
    * (i.e. 1 0001 0000 0010 0001 binary) as a divisor, calculates the          *
    * remainder. This remainder is then defined to be the checksum.             *
    *                                                                           *
    * So EPROM CODE / POLYNOMIAL = DIVIDEND remainder CHECKSUM                  *
    *                                                                           *
    *****************************************************************************/

   end ON_BOARD_EPROM_TEST;

MPIC_AND_TMOUT_TEST: procedure external;

   /*****************************************************************************
    *                                                                           *
    * Tests the Master PIC and time-out circuitry by performing a write to      *
    * an invalid port. This should result in the generation of a time-out       *
    * interrupt.                                                                *
    *                                                                           *
    *****************************************************************************/

   end MPIC_AND_TMOUT_TEST;
```

```
MPIC_AND_PIT_TEST: procedure external;

   /*************************************************************************
    *                                                                       *
    * Tests the Master PIC and timer 0 by enabling timer 0 and then delaying *
    * long enough for timer 0 to cause an interrupt.                        *
    *                                                                       *
    *************************************************************************/

   end MPIC_AND_PIT_TEST;

SPIC_MPIC_AND_PIT_TEST: procedure external;

   /*************************************************************************
    *                                                                       *
    * Tests the Slave PIC and timer 1 by enabling timer 1 and then delaying  *
    * long enough for timer 1 to cause an interrupt.                        *
    *                                                                       *
    *************************************************************************/

   end SPIC_MPIC_AND_PIT_TEST;

COPROCESSOR_TEST: procedure external;

   /*************************************************************************
    *                                                                       *
    * Tests the 80287 Numeric Coprocessor by performing a real number       *
    * calculation with a known result and then checking whether the actual  *
    * result is within an acceptable range when compared to the expected    *
    * result.                                                               *
    *                                                                       *
    *************************************************************************/

   end COPROCESSOR_TEST;
```

```
TEST_ACCESS_TO_1611: procedure external;

   /*************************************************************************
    *                                                                       *
    * This procedure tests accesses to the EPROM/RAM card by reading a      *
    * location on this card. If a time-out is generated (this implies that  *
    * either the EPROM/RAM card is absent, or the CPU card is unable to     *
    * access an off-board location), the routine then attempts to read a    *
    * value from the SDB card (i.e. a Multibus read). If a time-out is       *
    * generated again, then the routine assumes that the CPU card is unable *
    * to access an off-board location, rather than that both the SDB and the *
    * EPROM/RAM cards are absent.                                           *
    *                                                                       *
    *************************************************************************/

   end TEST_ACCESS_TO_1611;

APPLICATIONS_ROM_TEST: procedure external;

   /*************************************************************************
    *                                                                       *
    * This procedure calculates a ROM checksum of the applications code     *
    * resident on the HECS 1611 EPROM/RAM card in exactly the same manner   *
    * as the on-board checksum is calculated.                               *
    *                                                                       *
    * It considers the ROM based code to be one binary number               *
    * and then by using the CRC_CCITT polynomial X**16 + X**12 + X**5 + 1   *
    * (i.e. 1 0001 0000 0010 0001 binary) as a divisor, calculates the      *
    * remainder. This remainder is then defined to be the checksum.         *
    *                                                                       *
    * So EPROM CODE / POLYNOMIAL = DIVIDEND remainder CHECKSUM              *
    *                                                                       *
    *************************************************************************/

   end APPLICATIONS_ROM_TEST;

SDB_SELFTEST: procedure external;

   /*************************************************************************
    *                                                                       *
    * During POST, this procedure checks the self test result reported by   *
    * the SDB card. If called during off-line diagnostics mode, the         *
    * procedure will issue a reset to the SDB card and then delay for 100 ms *
    * before checking the result of the SDB self test.                      *
    *                                                                       *
    *************************************************************************/

   end SDB_SELFTEST;
```

```
SDB_RAM_TEST: procedure external;

   /*************************************************************************
    *                                                                       *
    * Tests the SDB dual port user RAM by performing an alternate write and *
    * read of 0101 0101 0101 0101 and 1010 1010 1010 1010 binary.           *
    *                                                                       *
    * If a time-out interrupt occurs or what is read is not what was written *
    * then the SDB RAM is reported as having failed.                        *
    *                                                                       *
    *************************************************************************/

   end SDB_RAM_TEST;

LOCAL_BUS_TEST: procedure external;

   /*************************************************************************
    *                                                                       *
    * This tests the functionality of the local bus. The only way to        *
    * determine via which bus off-board accesses are being performed at run *
    * time in the standard computing segment, is by performing a speed test *
    * over both busses (Multibus runs much slower than Local bus).          *
    *                                                                       *
    * The routine operates as follows:                                      *
    *                                                                       *
    * The timer is enabled and for 10 ms a single location is read from the *
    * EPROM/RAM card (i.e. a Local bus read). For each Local bus read, a    *
    * counter is incremented. Following this, a single location is read for *
    * 10 ms from the SDB card (a Multibus read). For each Multibus          *
    * read, a second counter is incremented. The two counter values are then *
    * compared and the test passes if                                       *
    *                                                                       *
    * LOCAL_BUS_COUNTER - MULTIBUS_COUNTER    >    SPEED_THRESHOLD          *
    *                                                                       *
    * where SPEED_THRESHOLD is a predefined value, determined by the 1442   *
    * clock speed and the 1611 memory device access times.                  *
    *                                                                       *
    *************************************************************************/

   end LOCAL_BUS_TEST;
```

2.5.3.3 STDINTS.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDINTS.EPD contains the corresponding external declarations of           *
 * all entities declared public in STDINTS.PLM.                              *
 *                                                                           *
 *****************************************************************************/

/*****************************************************************************
 *                                                                           *
 *                        Global variables                                   *
 *                                                                           *
 *****************************************************************************/


declare TMOUT_INTERRUPT   byte external;
declare TMOUT_STATUS      byte external;
declare BUS_GRANT         byte external;
declare CLOCK_1_INTERRUPT byte external;
declare CLOCK_2_INTERRUPT byte external;

/*****************************************************************************
 *                                                                           *
 *                      External interrupt procedures                        *
 *                                                                           *
 *****************************************************************************/

/* Intel reserved interrupt handlers */

DIVIDE_ERROR:             procedure interrupt 0 external;
  end DIVIDE_ERROR;

SINGLE_STEP:              procedure interrupt 1 external;
  end SINGLE_STEP;

NMI:                      procedure interrupt 2 external;
  end NMI;

BREAKPOINT:               procedure interrupt 3 external;
  end BREAKPOINT;

INTO_OVERFLOW:            procedure interrupt 4 external;
  end INTO_OVERFLOW;

BOUND_RANGE:              procedure interrupt 5 external;
  end BOUND_RANGE;
```

```
INVALID_OPCODE:           procedure interrupt 6 external;
  end INVALID_OPCODE;

PROC_EXT_NOT_AVAILABLE:   procedure interrupt 7 external;
  end PROC_EXT_NOT_AVAILABLE;

DOUBLE_EXCEPTION:         procedure interrupt 8 external;
  end DOUBLE_EXCEPTION;

PROC_EXT_SEGMENT_OVERRUN: procedure interrupt 9 external;
  end PROC_EXT_SEGMENT_OVERRUN;

INVALID_TASK:             procedure interrupt 10 external;
  end INVALID_TASK;

SEGMENT_NOT_PRESENT:      procedure interrupt 11 external;
  end SEGMENT_NOT_PRESENT;

STACK_OVERRUN:            procedure interrupt 12 external;
  end STACK_OVERRUN;

GENERAL_PROTECTION:       procedure interrupt 13 external;
  end GENERAL_PROTECTION;

NCP_INTERRUPT:            procedure interrupt 16 external;
  end NCP_INTERRUPT;


/* Default interrupt handler */

ILLEGAL_INT:              procedure interrupt 33  external;
   end ILLEGAL_INT;
```

```
/* Master 8259A PIC interrupt handlers */

TMOUT:                  procedure interrupt 96  external;
   end TMOUT;

MASTER_CLOCK:           procedure interrupt 97  external;
   end MASTER_CLOCK;

MASTER_INTERRUPT_2:     procedure interrupt 98  external;
   end MASTER_INTERRUPT_2;

MASTER_INTERRUPT_3:     procedure interrupt 99  external;
   end MASTER_INTERRUPT_3;

MASTER_INTERRUPT_4:     procedure interrupt 100  external;
   end MASTER_INTERRUPT_4;

MASTER_INTERRUPT_7:     procedure interrupt 101  external;
   end MASTER_INTERRUPT_7;

/* Slave 8259A PIC interrupt handlers */

SLAVE_INTERRUPT_0:      procedure interrupt 128 external;
   end SLAVE_INTERRUPT_0;

SLAVE_INTERRUPT_1:      procedure interrupt 129 external;
   end SLAVE_INTERRUPT_1;

SLAVE_INTERRUPT_2:      procedure interrupt 132 external;
   end SLAVE_INTERRUPT_2;

TMAP_INTERRUPT:         procedure interrupt 130 external;
   end TMAP_INTERRUPT;

SLAVE_CLOCK:            procedure interrupt 131 external;
   end SLAVE_CLOCK;

SLAVE_INTERRUPT_5:      procedure interrupt 133 external;
   end SLAVE_INTERRUPT_5;

SLAVE_INTERRUPT_6:      procedure interrupt 134 external;
   end SLAVE_INTERRUPT_6;

SLAVE_INTERRUPT_7:      procedure interrupt 135 external;
   end SLAVE_INTERRUPT_7;
```

2.5.3.4 STDPIC.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDPIC.EPD contains the corresponding external declarations of            *
 * all entities declared public in STDPIC.PLM.                               *
 *                                                                           *
 *****************************************************************************/

$ include (PICLITS.INC)

INITIALISE_8259A_PIC: procedure external;

    /*************************************************************************
    *                                                                        *
    * Initialises the Programmable Interrupt Controller as follows:         *
    *                                                                        *
    * 1) Master PIC                                                         *
    *                                                                        *
    *    i) to trigger on edge transitions                                   *
    *   ii) with an interrupt base of 96 and interval of 8                   *
    *  iii) input line IR6 is linked to the 8274 MPSC and input line IR7 is  *
    *       linked to the Slave PIC and both these devices supply their own  *
    *       vectors when providing interrupts                                *
    *   iv) buffered and not special fully nested                            *
    *    v) to provide an automatic End Of Interrupt                         *
    *   vi) to operate in 8086 mode                                          *
    *                                                                        *
    * 2) Slave PIC                                                          *
    *                                                                        *
    *    i) to trigger on edge transitions                                   *
    *   ii) with an interrupt base of 128 and interval of 8                  *
    *  iii) cascaded to input line IR7 of the Master PIC                     *
    *   iv) buffered and not special fully nested                            *
    *    v) to expect a normal End Of Interrupt                              *
    *   vi) to operate in 8086 mode                                          *
    *                                                                        *
    *************************************************************************/

   end INITIALISE_8259A_PIC;
```

```
UNMASK_PIC: procedure (INTERRUPT_NO) external;

/*****************************************************************************
 *                                                                           *
 * This procedure takes the PIC's current mask status and unmasks additional *
 * interrupts.                                                               *
 *                                                                           *
 *****************************************************************************/

   declare INTERRUPT_NO             byte;

   end UNMASK_PIC;

MASK_PIC: procedure (INTERRUPT_NO) external;

/*****************************************************************************
 *                                                                           *
 * This procedure takes the PIC's current mask status and masks additional   *
 * interrupts.                                                               *
 *                                                                           *
 *****************************************************************************/

   declare INTERRUPT_NO             byte;

   end MASK_PIC;

ISSUE_EOI: procedure (INTERRUPT_NO) external;

/*****************************************************************************
 *                                                                           *
 * This procedure issues an end of interrupt signal to the PIC for the       *
 * appropriate interrupt.                                                    *
 *                                                                           *
 *****************************************************************************/

   declare INTERRUPT_NO byte;

   end ISSUE_EOI;
```

2.5.3.5 STDPIT.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDPIT.EPD contains the corresponding external declarations of            *
 * all entities declared public in STDPIT.PLM.                               *
 *                                                                           *
 *****************************************************************************/

$ include (PITLITS.INC)

INITIALISE_8254_PIT: procedure external;
   end INITIALISE_8254_PIT;
```

2.5.3.6 STDSTAT.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDSTAT.EPD contains the corresponding external declarations of           *
 * all entities declared public in STDSTAT.PLM.                              *
 *                                                                           *
 *****************************************************************************/

$ include (STATLITS.INC)

OUTPUT_TO_LATCH: procedure (TEST_NO) external;

   /* Write to status latch */

   declare TEST_NO byte;

   end OUTPUT_TO_LATCH;

INPUT_FROM_LATCH: procedure byte external;

   /* Read status latch */

   end INPUT_FROM_LATCH;
```

2.5.3.7 STDDIAG.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDDIAG.EPD contains the corresponding external declarations of           *
 * all entities declared public in STDDIAG.PLM.                              *
 *                                                                           *
 *****************************************************************************/

DIAGNOSTICS: procedure external;
   end DIAGNOSTICS;
```

2.5.3.8 STDIO.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDIO.EPD contains the corresponding external declarations of             *
 * all entities declared public in STDIO.PLM.                                *
 *                                                                           *
 *****************************************************************************/

$ include (STDIO.INC)

/*****************************************************************************/
/*                                                                           */
/*     INITIALISE IO ADDRESSES : sets up the port addresses of the 8274      */
/*         for two ports at a time. The interrupt mode and addresses         */
/*         are set up at the same time.                                      */
/*                                                                           */
/*****************************************************************************/

INITIALISE_IO_ADDRESSES : procedure( PORT_NO,
                                     BASE_ADDRESS,
                                     INTERRUPT_NO  )    external;


   declare  PORT_NO             byte;
   declare  BASE_ADDRESS        word;
   declare  INTERRUPT_NO        byte;

   end INITIALISE_IO_ADDRESSES;
```

```
/*************************************************************************/
/*                                                                       */
/*     INITIALISE_PORT : Writes the control characters required to set   */
/*         up a 8274 MPSC from the table set up in the port data area    */
/*         BUFFER(I).                                                    */
/*                                                                       */
/*     INPUT : A pointer to the correct buffer area.                     */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*     Any output in progress at the time of initialisation is aborted   */
/*     and the buffers restored to the free buffer list.                 */
/*                                                                       */
/*************************************************************************/

INITIALISE_PORT : procedure( PORT_NO,
                             BAUD_RATE,
                             NO_OF_DATA_BITS,
                             NO_OF_STOP_BITS,
                             PARITY_TYPE,
                             MODE ) external;

   declare PORT_NO         byte;
   declare BAUD_RATE       byte;
   declare NO_OF_DATA_BITS byte;
   declare NO_OF_STOP_BITS byte;
   declare PARITY_TYPE     byte;
   declare MODE            byte;

   end INITIALISE_PORT;

/*************************************************************************/
/*                                                                       */
/*     MPSC_IO_TEST : Called to test the operation of the 8274           */
/*                    multiple protocol serial controller.               */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : A test result.                                           */
/*                 1 ==> Test passed                                     */
/*                 2 ==> Test failed                                     */
/*                                                                       */
/*************************************************************************/

MPSC_IO_TEST : procedure ( PORT_NO ) byte external;

   declare PORT_NO  byte;

   end MPSC_IO_TEST;
```

```
/*************************************************************************/
/*                                                                       */
/*     ENABLE_PORT : Called to enable the Tx and Rx of the port.         */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

ENABLE_PORT : procedure ( PORT_NO ) external;

   declare PORT_NO  byte;

   end ENABLE_PORT;

/*************************************************************************/
/*                                                                       */
/*     DISABLE_PORT : Called to disable the Tx and Rx of the port.       */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

DISABLE_PORT : procedure ( PORT_NO ) external;

   declare PORT_NO  byte;

   end DISABLE_PORT;

/*************************************************************************/
/*                                                                       */
/*     CLEAR_BUFFER : Called to clear an input buffer of unread          */
/*         characters.  If the routine LAST_READ_CHARACTER is used, the  */
/*         buffer is never emptied and must be done so explicitly        */
/*         by the user.                                                  */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

CLEAR_BUFFER : procedure ( PORT_NO ) external;

   declare PORT_NO  byte;

   end CLEAR_BUFFER;
```

```
/*************************************************************************/
/*                                                                       */
/*     TRANSMIT_CHARACTER : Called as a direct result of a Tx buffer     */
/*         empty interrupt, this routine send the next character to the  */
/*         8274, and if this emptys the transmit buffer, the next        */
/*         buffer is linked to the port data area.  If no characters     */
/*         are waiting to be sent, the Tx pending flag in the 8274 is    */
/*         reset.                                                        */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

TRANSMIT_CHARACTER : procedure( PORT_NO ) external;

   declare PORT_NO     byte;

   end TRANSMIT_CHARACTER;

/*************************************************************************/
/*                                                                       */
/*     RECEIVE_CHARACTER : Called as a direct result of a Rx buffer      */
/*         available interrupt, this routine transfers the received      */
/*         character to the INPUT_BUFFER in PORT_DATA( PORT_NO ).        */
/*         If the character is a <CR>, the INPUT_FLAG is set to 2.       */
/*            else INPUT_FLAG is set to 1.                               */
/*         If the input character is a <DEL> or a <BACK_SPACE> then      */
/*         a backspace, space, backspace is transmitted and the char     */
/*         deleted from the buffer.                                      */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

RECEIVE_CHARACTER : procedure( PORT_NO ) external;

   declare PORT_NO      byte;

   end RECEIVE_CHARACTER;
```

```
/*************************************************************************/
/*                                                                       */
/*     GET_MODE: Returns the mode of a serial port.                      */
/*         0 = TERMINAL MODE                                             */
/*         1 = DATA MODE                                                 */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*                                                                       */
/*     OUTPUT : The function returns a byte.                             */
/*                                                                       */
/*************************************************************************/

GET_MODE: procedure( PORT_NO ) byte external;

   declare PORT_NO  byte;

   end GET_MODE;

/*************************************************************************/
/*                                                                       */
/*     SET_MODE: Called to set the mode of a serial port.                */
/*                                                                       */
/*     INPUT : A pointer to the correct port data area.                  */
/*             The byte value of the new mode.                           */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

SET_MODE: procedure( PORT_NO, NEW_MODE ) external;

   declare PORT_NO  byte;
   declare NEW_MODE byte;

   end SET_MODE;
```

```
/*************************************************************************/
/*                                                                       */
/*     WRITE : Transfers data from a user character string to a write    */
/*         buffer, and queues the write buffer for output.               */
/*         If a free buffer is not available the routine waits for a     */
/*         buffer to become free before continuing.                      */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*             A pointer to the user data string (DATA_ADDRESS)          */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/

WRITE : procedure ( PORT_NO, DATA_ADDRESS ) external;

   declare PORT_NO  byte;
   declare DATA_ADDRESS pointer;

   end WRITE;


/*************************************************************************/
/*                                                                       */
/*     WRITE_POLL : Writes a character string pointed to by DATA_ADDRESS */
/*         to the port pointed to by PORT_NO.  The routine waits for all */
/*         characters to be transmitted before returning. Tx empty       */
/*         interrupts are disabled for the duration of the routine.      */
/*         The port is polled continually waiting for each character     */
/*         to be sent before the next one is output.                     */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*             A pointer to the user data string (DATA_ADDRESS)          */
/*                                                                       */
/*     OUTPUT : none.                                                    */
/*                                                                       */
/*************************************************************************/


WRITE_POLL : procedure ( PORT_NO, DATA_ADDRESS ) external;

   declare PORT_NO  byte;
   declare DATA_ADDRESS pointer;

   end WRITE_POLL;
```

```
/*************************************************************************/
/*                                                                       */
/*     INPUT_CHARACTER : Clears the input buffer, then waits for a       */
/*         character to be received.  It is a function call.             */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*                                                                       */
/*     OUTPUT : the function call returns a character.                   */
/*                                                                       */
/*************************************************************************/

INPUT_CHARACTER: procedure ( PORT_NO ) byte external;

   declare PORT_NO  byte;

   end INPUT_CHARACTER;

/*************************************************************************/
/*                                                                       */
/*     GET_CHARACTER : Returns the next character in the input buffer.   */
/*         If no characters are waiting then the buffer is cleared and   */
/*         the program waits for an input string terminated by a <CR>.   */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*                                                                       */
/*     OUTPUT : the function call returns a character.                   */
/*                                                                       */
/*************************************************************************/

GET_CHARACTER: procedure ( PORT_NO ) byte external;

   declare PORT_NO   byte;

   end GET_CHARACTER;
```

```
/*************************************************************************/
/*                                                                       */
/*     LAST_CHARACTER_READ: Returns the content of the input buffer      */
/*         of the specified port.  It is a function call.                */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*                                                                       */
/*     OUTPUT : the function call returns a character.                   */
/*                                                                       */
/*************************************************************************/

LAST_CHARACTER_READ: procedure ( PORT_NO ) byte external;

   declare PORT_NO  byte;

   end LAST_CHARACTER_READ;

/*************************************************************************/
/*                                                                       */
/*     GET_BYTE : Clears the input buffer, waits for a <CR> and then     */
/*         returns the value of the 1st numeric string in the buffer.    */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*                                                                       */
/*     OUTPUT : the function call returns a numeric byte value.          */
/*                                                                       */
/*************************************************************************/

GET_BYTE: procedure( PORT_NO, MIN_VALUE, MAX_VALUE ) byte external;

   declare PORT_NO        byte;

   declare MIN_VALUE      byte;
   declare MAX_VALUE      byte;

   end GET_BYTE;
```

```
/*************************************************************************/
/*                                                                       */
/*     GET_WORD : Clears the input buffer, waits for a <CR> and then     */
/*         returns the value of the 1st numeric string in the buffer.    */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*                                                                       */
/*     OUTPUT : the function call returns a numeric word value.          */
/*                                                                       */
/*************************************************************************/

GET_WORD: procedure( PORT_NO, MIN_VALUE, MAX_VALUE ) word external;

   declare PORT_NO         byte;
   declare MIN_VALUE       word;
   declare MAX_VALUE       word;

   end GET_WORD;

/*************************************************************************/
/*                                                                       */
/*     BUFFER_READY: Returns 00H if the input buffer is empty.           */
/*         Returns 01H if data is available in the input buffer.         */
/*         Returns 02H if a <CR> has been received in the input buffer.  */
/*                                                                       */
/*     INPUT : A pointer to the port data area (PORT_NO).                */
/*                                                                       */
/*     OUTPUT : the function call returns a byte value.                  */
/*                                                                       */
/*************************************************************************/

BUFFER_READY: procedure( PORT_NO ) byte external;

   declare PORT_NO     byte;

   end BUFFER_READY;
```

```
/*************************************************************************/
/*                                                                       */
/*                  MPSC interrupt handlers                              */
/*                                                                       */
/*************************************************************************/

CHB_TX_EMPTY:   procedure interrupt  80 external;
   end CHB_TX_EMPTY;

/*************************************************************************/

CHA_TX_EMPTY:   procedure interrupt 84 external;
   end CHA_TX_EMPTY;

/*************************************************************************/

CHA_RX_AVAIL:   procedure interrupt 86 external;
    end CHA_RX_AVAIL;


CHB_RX_AVAIL:   procedure interrupt 82 external;
    end CHB_RX_AVAIL;

/*************************************************************************/

CHA_RX_ERROR: procedure interrupt 87 external;
    end CHA_RX_ERROR;

/*************************************************************************/

CHB_RX_ERROR: procedure interrupt 83 external;
    end CHB_RX_ERROR;
```

2.5.3.9 STDCONVT.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDCONVT.EPD contains the corresponding external declarations of          *
 * all entities declared public in STDCONVT.PLM.                             *
 *                                                                           *
 *****************************************************************************/

C010_BIN_WORD_TO_ASCII_OCTAL: Procedure (BINARY_NUMBER,ASCII_ARRAY_PTR) external;
   declare BINARY_NUMBER word;
   declare ASCII_ARRAY_PTR pointer;
   end C010_BIN_WORD_TO_ASCII_OCTAL;

C020_BIN_BYTE_TO_ASCII_OCTAL: Procedure (BINARY_NUMBER,ASCII_ARRAY_PTR) external;
   declare BINARY_NUMBER byte;
   declare ASCII_ARRAY_PTR pointer;
   end C020_BIN_BYTE_TO_ASCII_OCTAL;

C030_BIN_WORD_TO_ASCII_DEC: Procedure (BINARY_NUMBER,ASCII_ARRAY_PTR) external;
   declare BINARY_NUMBER word;
   declare ASCII_ARRAY_PTR pointer;
   end C030_BIN_WORD_TO_ASCII_DEC;

C040_BIN_BYTE_TO_ASCII_DEC: Procedure (BINARY_NUMBER,ASCII_ARRAY_PTR) external;
   declare BINARY_NUMBER byte;
   declare ASCII_ARRAY_PTR pointer;
   end C040_BIN_BYTE_TO_ASCII_DEC;

C050_BIN_BYTE_TO_ASCII_BIN: Procedure (BINARY_NUMBER,ASCII_ARRAY_PTR) external;
   declare BINARY_NUMBER byte;
   declare ASCII_ARRAY_PTR pointer;
   end C050_BIN_BYTE_TO_ASCII_BIN;

C060_WORD_TO_BCD_VAL: Procedure (BINARY_NUMBER) word external;
   declare (BINARY_NUMBER) byte;
   end C060_WORD_TO_BCD_VAL;

C070_CONVERT_WORD_TO_HEX: Procedure ( INPUT_WORD, ASCII_ADDRESS ) external;
   declare INPUT_WORD word;
   declare ASCII_ADDRESS pointer;
   end C070_CONVERT_WORD_TO_HEX;
```

2.5.3.10 STDCPU.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDCPU.EPD contains the corresponding external declarations of            *
 * all entities declared public in STDCPU.ASM.                               *
 *                                                                           *
 *****************************************************************************/

CPU_TEST:  procedure byte external;

   /* Test the functionality of the CPU */

   end CPU_TEST;
```

2.5.3.11 STDRAM.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDRAM.EPD contains the corresponding external declarations of            *
 * all entities declared public in STDRAM.ASM.                               *
 *                                                                           *
 *****************************************************************************/

/*****************************************************************************
 *                                                                           *
 *                  Global variables                                         *
 *                                                                           *
 *****************************************************************************/

declare BIT_RESULT structure
                  (CPU_BOARD          byte,
                   CPU                byte,
                   ROM                byte,
                   MASTER_PIC         byte,
                   SLAVE_PIC          byte,
                   TMOUT_CIRCUIT      byte,
                   PIT                byte,
                   COPROCESSOR        byte,
                   MPSC               byte,
                   RAM                byte,
                   ACCESS_TO_1611     byte,
                   ERAM               byte,
                   EROM               byte,
                   SDB_BOARD          byte,
                   SDB_SELFTEST       byte,
                   SDB_RAM            byte,
                   LOCAL_BUS          byte     ) external;
```

```
declare PROCESSOR_ID        word external; /* for dual processor environment */
declare NUMBER_OF_RUNS      word external; /* for "cold" or "warm" start     */

MEM_TEST:  procedure byte external;

   /* Test the functionality of the RAM */

   end MEM_TEST;
```

2.5.3.12 STDERAM.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDERAM.EPD contains the corresponding external declarations of           *
 * all entities declared public in STDERAM.ASM.                              *
 *                                                                           *
 *****************************************************************************/

/*****************************************************************************
 *                                                                           *
 *                          Global variables                                 *
 *                                                                           *
 *****************************************************************************/

declare OB_EPROM_CHECKSUM           word external;
declare OB_CSUM_PRESENT             byte external;
declare EPROM_CHECKSUM              word external;
declare EPROM_ID                    byte external;
declare SYSTEM_VERSION              byte external;
declare SYSTEM_NUMBER               byte external;
declare EPROM_RAM_OPTION            byte external;
declare CSUMS_PRESENT               byte external;
declare NO_OF_PROCS                 byte external;

declare ROM_1  (SEGMENT_MAX)        byte external;
declare ROM_2  (SEGMENT_MAX)        byte external;
declare ROM_3  (SEGMENT_MAX)        byte external;
declare ROM_4  (SEGMENT_MAX)        byte external;
declare ROM_5  (SEGMENT_MAX)        byte external;
declare ROM_6  (SEGMENT_MAX)        byte external;
declare ROM_7  (SEGMENT_MAX)        byte external;
declare ROM_8  (SEGMENT_MAX_LESS_2) byte external;

/* Two bytes at the top of the EPROM/RAM card are reserved for the checksum. */

declare ROM_9  (SEGMENT_MAX)        byte external;
declare ROM_10 (SEGMENT_MAX_LESS_2) byte external;

/* Two bytes at the top of the CPU card are reseved for the checksum.       */
```

```
ERAM_TEST: procedure byte external;

   /* Test the functionality of the RAM on the EPROM/RAM 1611 card */

   end ERAM_TEST;
```

2.5.3.13 STDSDB.EPD

```
/*****************************************************************************
 *                                                                           *
 * STDSDB.EPD contains the corresponding external declarations of            *
 * all entities declared public in STDSDB.ASM.                               *
 *                                                                           *
 *****************************************************************************/

$ include (SDBLITS.INC)

declare SDB_RAM_BUFFER(SDB_RAM_BUFFER_MAX) word external ;
declare SDB_TEST_STATUS                    word external ;
declare SDB_TEST_NUMBER                    word external ;
declare SDB_TEST_PARAMETER_1               word external ;
declare SDB_TEST_PARAMETER_2               word external ;
declare SDB_TEST_PARAMETER_3               word external ;
```

2.5.4 Include files

2.5.4.1 PLMPAR.INC

```
$ debug
$ large
$ rom
```

2.5.4.2 LITS.INC

```
/*****************************************************************************
 *                                                                           *
 *                        General literals                                   *
 *                                                                           *
 *****************************************************************************/

declare ZERO_ONES           literally '55h'   ;
declare ONE_ZEROS           literally '0aah'  ;
declare WORD_OF_ZERO_ONES   literally '5555h' ;
declare WORD_OF_ONE_ZEROS   literally '0aaaah';
declare DUMMY_VALUE         literally '0ffffh';

declare READ                literally '0'     ;
declare WRITE_IT            literally '1'     ;

declare MAP                 literally '05h'   ;
declare INVALID_PORT        literally '0h'    ;
declare RESET               literally '0'     ;

declare UNTESTED            literally '0'     ;
declare PASSED              literally '1'     ;
declare FAILED              literally '2'     ;
declare NOT_PRESENT         literally '3'     ;
declare PRESENT             literally '1'     ;

declare FALSE               literally '0'     ;
declare TRUE                literally '1'     ;

declare BUSY                literally '1'     ;
declare ABSENT              literally '8'     ;

declare SDB_RAM_BUFFER_MIN  literally '0h'    ;
declare SDB_RAM_BUFFER_MAX  literally '06ffeh';
declare CTRL_287            literally '033eh' ;
declare NCP_MASK            literally '02h'   ;
declare CRC_POLYNOMIAL      literally '1021h' ;  /* 0001 0000 0010 0001 */
declare SEGMENT_MAX         literally '0fffdh';
declare SEGMENT_MAX_LESS_2  literally '0fffbh';
```

```
declare ROM_BLOCK_MAX       literally '7'     ;
declare SPEED_THRESHOLD     literally '10'    ;
declare BUS_NOT_GRANTED     literally '1'     ;  /* Bus grant is active low */
declare MAX_RETRIES         literally '2'     ;
declare VDU                 literally '1'     ;
declare KB                  literally '1'     ;
declare SECOND_RUN          literally '5a5ah' ;

  /* Maximum retries allowed for the processor to be granted the bus */

declare CLEARSCREEN (*) byte data

   /* Clear the VDU */

   (ESC, '[1J', ESC, '[0;0H', EOM);
```

2.5.4.3 PICLITS.INC

```
/*****************************************************************************
 *                                                                           *
 *   8259A Programmable Interrupt Controller literals                        *
 *                                                                           *
 *****************************************************************************/

declare PIC_MASTER_8259A_ADR1 literally '0c0h'     ;
declare PIC_MASTER_8259A_ADR2 literally '0c2h'     ;

declare PIC_SLAVE_8259A_ADR1  literally '0c4h'     ;
declare PIC_SLAVE_8259A_ADR2  literally '0c6h'     ;

declare MASTER_ICW1_8259A     literally '011h';
/* Trigger on edge transitions for interrupts; Interval of 8;
   Cascade mode; ICW4 required                                          */

declare MASTER_ICW2_8259A     literally '060h';
/* Interrupt base of 96                                                 */

declare MASTER_ICW3_8259A     literally '060h';
/* IR6 linked to serial controller and IR5 cascaded to 8259A slave :    */
/* both supply their own vectors when providing interrupts              */

declare MASTER_ICW4_8259A     literally '00fh';
/* Not special fully nested mode; bufferred mode (master);              */
/* auto EOI; 8086 mode                                                  */

$ eject
```

```
declare SLAVE_ICW1_8259A      literally '011h';
/* Edge triggered for interrupts; Interval of 8;
   Cascade mode; ICW4 required                                          */

declare SLAVE_ICW2_8259A      literally '080h';
/* Interrupt base of 128                                                */

declare SLAVE_ICW3_8259A      literally '005h';
/* Locates the slave PIC (as being for IR5) on the master PIC input lines    */

declare SLAVE_ICW4_8259A      literally '009h';
/* Not special fully nested mode; bufferred mode (slave);               */
/* normal EOI; 8086 mode                                                */

declare LEVEL_TRIGGERED_MODE  literally '00001000b';
/* Mask to allow for level triggered interrupts                         */
declare AUTO_EOI_MASK         literally '00000010b';
/* Mask to check for PIC initialised to AUTO EOI                        */

declare EOI_MASK              literally '060h'     ;

$ eject

declare TMOUT_INTERRUPT_NO    literally '0'        ;
declare CLOCK_1_INTERRUPT_NO  literally '1'        ;
declare DRAM_INTERRUPT_NO     literally '2'        ;
declare SDB_INTERRUPT_NO      literally '3'        ;
declare GRAPHICS_INTERRUPT_NO literally '4'        ;
declare SLAVE_INTERRUPT_NO    literally '5'        ;
declare MPSC_INTERRUPT_NO     literally '6'        ;
declare DEFAULT_INTERRUPT_NO  literally '7'        ;
declare API_INTERRUPT_NO      literally '8'        ;
declare SCMB_INTERRUPT_NO     literally '9'        ;
declare INTERRUPT_NO_10       literally '10'       ;
declare TMAP_INTERRUPT_NO     literally '11'       ;
declare CLOCK_2_INTERRUPT_NO  literally '12'       ;
declare INTERRUPT_NO_13       literally '13'       ;
declare INTERRUPT_NO_14       literally '14'       ;
declare INTERRUPT_NO_15       literally '15'       ;

$ eject
```

```
declare MASK_ALL_INTS          literally '11111111b';

/* Master PIC */

declare UNMASK_TMOUT           literally '11111110b';   /* Time-out on INT 0  */
declare UNMASK_CLOCK_1         literally '11111101b';   /* Clock 1 on INT 1   */
declare UNMASK_DRAM            literally '11111011b';   /* DRAM on INT 2      */
declare UNMASK_SDB             literally '11110111b';   /* SDB on INT 3       */
declare UNMASK_GRAPHICS        literally '11101111b';   /* GIM on INT 4       */
declare UNMASK_SLAVE           literally '11011111b';   /* Slave PIC on INT 5 */
declare UNMASK_MPSC            literally '10111111b';   /* MPSC on INT 6      */

declare UNMASK_ALL_MASTER      literally '10000000b';

/* Slave PIC */

declare UNMASK_API             literally '11111110b';   /* API on INT 4       */
declare UNMASK_SCMB            literally '11111101b';   /* SCMB on INT 5      */
declare UNMASK_SCSI            literally '11111011b';   /* SCSI on INT 6      */
declare UNMASK_SLAVE_INT3      literally '11110111b';   /* Slave 1442 ID      */
declare UNMASK_CLOCK_2         literally '11101111b';   /* Clock 2 on INT 4   */

declare UNMASK_ALL_SLAVE       literally '11101100b';

declare MASK_TMOUT             literally '00000001b';
declare MASK_CLOCK_1           literally '00000010b';
declare MASK_DRAM              literally '00000100b';
declare MASK_SDB               literally '00001000b';
declare MASK_GRAPHICS          literally '00010000b';
declare MASK_SLAVE             literally '00100000b';
declare MASK_MPSC              literally '01000000b';

declare MASK_API               literally '00000001b';
declare MASK_SCMB              literally '00000010b';

declare MASK_CLOCK_2           literally '00010000b';
```

2.5.4.4 PITLITS.INC

```
/*****************************************************************************
 *                                                                           *
 *          8254 Programmable Interval Timer Initialisation literals         *
 *                                                                           *
 *****************************************************************************/

declare CTR_CTRL_ADR        literally '0d6h'  ;
declare CTR0_8254_ADR       literally '0d0h'  ;
declare CTR1_8254_ADR       literally '0d2h'  ;
declare CTR2_8254_ADR       literally '0d4h'  ;

declare CTR0_8254_CTRL      literally '034h';
/* Select counter 0; R/W LSB then MSB; Mode 2: Rate generator : this mode   */
/* functions like a divide by N counter, typically used to generate a       */
/* Real Time clock interrupt; 16-bit counter                                */

declare CTR1_8254_CTRL      literally '074h';
/* Select counter 1; R/W LSB then MSB; Mode 2: Rate generator ;             */
/* 16-bit counter                                                           */

declare CTR2_8254_CTRL      literally '0b6h';
/* Select counter 2; R/W LSB then MSB; Mode 3; 16-bit counter               */

declare CTR0_8254_VAL_LSB   literally '000h';
declare CTR0_8254_VAL_MSB   literally '030h'; /* 3000h = 12288 decimal      */
/* Interrupt = 10 ms = Counter value / PIT clock freq = 12288 / 1.2288 MHz  */

declare CTR1_8254_VAL_LSB   literally '000h';
declare CTR1_8254_VAL_MSB   literally '030h';
/* Interrupt = 10 ms = Counter value / PIT clock freq = 12288 / 1.2288 MHz  */

declare CTR2_8254_VAL_LSB   literally '008h';
declare CTR2_8254_VAL_MSB   literally '000h';
/* Counter = 8 = PIT clock freq / (Baud rate * 16) = 1.2288 MHz/(9600 * 16) */
```

2.5.4.5 IOLITS.INC

```
declare BEL                   literally ' 07h';
declare ESC                   literally ' 1bh';
declare BACK_SPACE            literally ' 08h';
declare DEL                   literally ' 7Fh';
declare CR                    literally ' 0Dh';
declare LF                    literally ' 0Ah';
declare EOM                   literally '0FFh';
declare TERMINAL              literally ' 00h';
declare DATA_MODE             literally ' 01h';

declare EOI_8274              literally ' 38h';
declare ERROR_RESET_CODE      literally '030h';

/*****************************************************************************/

declare REC_ERR_MESS(*) byte data
        ( ESC, '[18;25H', 'CH.A REC.ERR [F,O,P]:         ', EOM );

declare RUBOUT_MESSAGE(*) byte data (BACK_SPACE, ' ', BACK_SPACE, EOM );

declare NO_OF_PORTS               literally '4';
declare NO_OF_BUFFERS             literally '10';

declare PORT_A                    literally '0';
declare PORT_B                    literally '1';
declare PORT_C                    literally '2';
declare PORT_D                    literally '3';

declare NO_PARITY                 literally '00000000b';
declare EVEN_PARITY               literally '00000011b';
declare ODD_PARITY                literally '00000010b';

declare ONE_STOP_BIT              literally '00000100b';
declare ONE_AND_A_HALF_STOP_BITS  literally '00001000b';
declare TWO_STOP_BITS             literally '00001100b';

declare FIVE_DATA_BITS            literally '00000000b';
declare SIX_DATA_BITS             literally '10000000b';
declare SEVEN_DATA_BITS           literally '01000000b';
declare EIGHT_DATA_BITS           literally '11000000b';
```

2.5.4.6 STATLITS.INC

```
/*****************************************************************************
 *                                                                           *
 *            POST firmware predefined status latch literals                 *
 *                                                                           *
 * The values listed below are those which are output to the 8-bit diagnostic *
 * latch on the CPU card in the event of a failure occuring during the       *
 * Power-On Self Tests or off-line diagnostics of the standard computing     *
 * segment.                                                                  *
 *                                                                           *
 *****************************************************************************/

declare STATUS_LATCH_ADR          literally '0e0h'  ;

declare RESET_LATCH               literally '00h'   ;  /* CPU card failures */
declare CPU_TEST_NO               literally '01h'   ;
declare ROM_TEST_NO               literally '02h'   ;
declare RAM_TEST_NO               literally '03h'   ;
declare INITIALISATION            literally '04h'   ;
declare MASTER_PIC_TEST_NO        literally '05h'   ;
declare SLAVE_PIC_TEST_NO         literally '06h'   ;
declare TMOUT_TEST_NO             literally '08h'   ;
declare PIT_TEST_NO               literally '09h'   ;
declare MPSC_TEST_NO              literally '0Ah'   ;
declare COPROCESSOR_TEST_NO       literally '0Bh'   ;
declare OFF_BOARD_ACCESS_TEST_NO  literally '0Ch'   ;

declare EPROM_RAM_ABSENT          literally '10h'   ;  /* RAM card failures */
declare EROM_TEST_NO              literally '11h'   ;
declare DELAYING_SLAVE_PROCS      literally '12h'   ;
declare ERAM_TEST_NO              literally '13h'   ;

declare SDB_SELFTEST_TEST_NO      literally '20h'   ;  /* SDB card failures */
declare SDB_RAM_TEST_NO           literally '21h'   ;

declare LOCAL_BUS_TEST_NO         literally '30h'   ;  /* Local bus failure */
declare MULTIBUS_TEST_NO          literally '31h'   ;  /* Multibus failure  */

/* General status latch outputs */

declare BUS_GRANTED               literally '0f0h'  ;
declare NO_BUS_GRANTED            literally '0f1h'  ;
```