

AD-A048 303

ROYAL RADAR ESTABLISHMENT MALVERN (ENGLAND)
A REPORT ON SOFTWARE FUNCTIONAL VARIABILITY.(U)
JAN 77 A D ALLEN, J H CROWTHER

F/G 9/2

UNCLASSIFIED

RRE-TN-789

NL

1 OF 1
AD
A048 303



END
DATE
FILMED
2-78
DDC

UNCLASSIFIED

BR57696

AD A 048303

RRE

TECHNICAL NOTE

No. 789

14 RRE-TN-789

6 A REPORT ON SOFTWARE FUNCTIONAL VARIABILITY.

10 Authors A. D. Allen and J. H. Crowther

*A D Allen is with Computer Analysts and Programmers Ltd

Royal Radar Establishment,
Procurement Executive,
Ministry of Defence,
Malvern, Worcs.

11 Jan 1977



Crown Copyright Reserved

THIS DOCUMENT, UNLESS SUBSEQUENTLY DECLARED TO BE UNLIMITED, IS ISSUED FOR THE INFORMATION OF SUCH PERSONS ONLY AS NEED TO KNOW ITS CONTENTS IN THE COURSE OF THEIR OFFICIAL DUTIES.

COPYRIGHT ©

1977

CONTROLLER
HMSO LONDON

UNCLASSIFIED

311 750

AD INU.
DDC FILE COPY

RRE TECH NOTE No.789 A REPORT ON SOFTWARE FUNCTIONAL VARIABILITY

UNLIMITED

A REPORT ON SOFTWARE FUNCTIONAL VARIABILITY

A D Allen* and J H Crowther

DDC	Diff Section	<input type="checkbox"/>
UNANNOUNCED		<input type="checkbox"/>
JUSTIFICATION		
BY		
DISTRIBUTION/AVAILABILITY CODES		
Dist. AVAIL. and/or SPECIAL		
A		

PREFACE

The work summarized in this report was carried out under MOD(PE) contracts by Computer Analysts and Programmers Limited with assistance from RRE. The contract (K/LT21a/511/CB/LT21a1) to design and implement a pilot system followed an earlier contract (K/LT21a/373/CB/LT21a1) which studied the problem and proposed the methodology known as Functional Variability. The study was undertaken in 1972, lasted for 6 months and involved an average of 4 people. The pilot implementation involved 4 people over a period of 13 months up to May 1974, at which time the system was handed over to RRE.

LIST OF CONTENTS

	<u>Page</u>
1 INTRODUCTION	1
2 TECHNIQUES FOR ACHIEVING FUNCTIONAL VARIABILITY	4
3 A PILOT IMPLEMENTATION OF THE TECHNIQUES	15
4 DISCUSSION	31

Appendix Index of terms

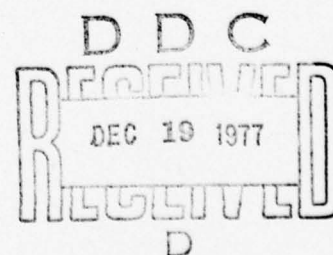
CHAPTER 1INTRODUCTION

This report is concerned with the means by which the software of a computer-based system can be safely modified without disrupting the service that the system provides.

The Need

No specific system is considered but a system used for air traffic control provides an example having the appropriate characteristics. It is vital that the system should operate correctly and continuously because either erroneous operation or non-functioning of the system could result in loss of life. Nevertheless, requirements for making changes to the system will inevitably arise for a number of reasons. The complexity of the software will ensure that programming errors could remain hidden throughout the initial testing phase, but as they are discovered during operational use it may become desirable to correct them. The complexity of the interactions between the computerized part of the system and the organization of men using the system will ensure that the true requirements of the system are unlikely to be completely formalized at the time the system is first designed. Only by using the currently available version of the system can a clearer idea be gained of what is really required.

* A D Allen is with Computer Analysts and Programmers Ltd



The complexity of the external environment within which the system operates will ensure that the environment will change and hence what is required of the system is also likely to change.

The Problem

The requirement to change the system without disrupting the service is specified in operational terms i.e. in terms of the functions the system is required to perform. On the other hand, changes to the system are implemented in engineering terms i.e. in terms of the actual construction of the system. The task is to satisfy the operational requirements for change by making suitable engineering changes. Hardware and software changes may both be needed but it is only the software changes that are considered in detail here. Three stages are recognized, each of which presents its own peculiar problems:

The first stage consists of designing suitable software changes. This involves understanding the original software design and is difficult for the same reasons that software design in general is difficult. For a program to operate efficiently its structure is not necessarily related in any simple manner to the functions it performs for the user: any one section of the program may be involved in many functions and any one function will use many sections of the program. (The term "function" is used here only in the sense relating to the purposes of the system as a whole. Elsewhere the term is used, quite legitimately, to refer to the purpose a section of program serves within the total software). For this reason, even a change which has a simple specification in terms of user functions may appear very complicated from the programmer's view point; changes may be required in many separate parts of the program and each part will have to be understood in detail to ensure that other user functions are not inadvertently affected.

The second stage is to test that the introduction of the proposed program changes has precisely the effect that was called for. This is by no means certain without testing because the program design process is so difficult. A certain amount of testing can be done off-line, but a full test would involve running the complete system in its changed form, fully manned and with live input data. The problem is that testing must not interfere with the performance of the operational system. If a new system is to be tested while the old system continues to be used operationally then evidently the two activities will be in competition for various types of resources - men, terminals, input data, processing power and computer storage. An economical solution will involve the sharing of resources between the two systems but must nevertheless ensure that the system under test cannot affect the operational system.

The third stage is to introduce the change into the operational system. This may well necessitate a disruption of the service, but the disruption may be considered negligible if it is of sufficiently short duration. It should be noted that some hiatus may be inevitable due to the operational problem of adjusting to a changed system. Replacing the old program by the new version can be a quick operation if the new program is available in suitable form on fast backing store. However, a system consists of more than a program; the current state of a system is represented by a data-base which may well contain a model of the state of the outside world. A problem is presented by the need to create a data-base that is suitable for use by the new system and that is also up-to-date and in accord with the state of the environment.

Work towards a Solution

In a CAP report of 1968 d'Agapeyeff and Clark coined the term "Functional Variability" to describe the ability to vary the functions performed by a system without sacrificing its reliability. In 1972 CAP won a Ministry of Defence contract (K/LT21a/373/CB/LT21a1) to study the problem with particular reference to air traffic control. The report resulting from this study described a set of techniques, or methodology, for achieving functional variability and described much of the proposed software structure in considerable detail. The methodology was designed to be consistent with the severe constraints anticipated for a future air traffic control system, although there is no implication that the methodology can only be applied in this field. The methodology proposed is referred to in this report as FV and a system that uses the methodology is called an FV system. In order to develop the ideas by testing them in practice CAP was awarded a further contract (K/LT21a/511/CB/LT21a1) to implement a pilot FV system to run on a Modular-One computer at RRE. The present report covers the work of both these contracts.

The Nature of the Solution

An essential characteristic of the proposed method is that changes are not made directly to the operational system by altering parts of it while it is on-line. The ultimate change of the operational system is made by completely replacing the system by a new version previously prepared and stored on fast backing store. In order first to test the new system, facilities are provided for allowing a test system to be run in parallel with the operational system in such a way that resources can be shared between the two systems and yet the test system cannot disturb the operational system. In order to make this possible there is a non-variable part of the system, called the Nucleus, which provides the necessary control, and the variable part of the system is obliged to conform to fixed regulations concerning its structure. The proposed methodology is concerned only with the problems of testing and introducing a system change; it offers no help with the design of the change and the representation of this design in computer readable form. However, other developments, such as structured programming, are helpful in this respect.

The approach lays great stress on reliability. The type of system being considered is such that high reliability and availability are essential. (If this were not so there would be no need to provide a safe means of testing and introducing changes). Thus, in order to build a practical FV system to do a job such as air traffic control, two goals must be achieved: the system must do the job reliably and it must be possible to modify it safely. The approach taken in deriving the FV method has been to concentrate first on the reliability and second on the safe modification. Consequently the structure finally proposed is suited to the techniques suggested for achieving reliability and can only be understood in these terms. Had safe modification been considered first and reliability second, a different concept might have emerged.

It must be understood that the FV method is not something that can be added to an existing system in order to make it variable. Rather it represents a radical approach to the design of the system. The Nucleus of an FV system must first be constructed to suit the machine in question and then the remainder of the operational system is constructed in a Functionally Variable form known as a "job base". The Nucleus contains only what is essential to support an FV system so that as much as possible of the system resides in the job base where it can be modified by the FV method. Thus functions normally expected of an "operating system" are mostly supplied by the job base, not by the Nucleus.

The Remainder of this Report

The proposed collection of techniques and principles that together constitute the FV methodology are described in Chapter 2. Basically this is an account of the results of the original study, but written in the light of our present understanding, after the experience of implementing the pilot system. The order in which the ideas are introduced is for ease of understanding rather than to give an historical account of the work. As explained above, aspects of reliability are the first concern and variability is considered only when a reliable structure has been established. Chapter 3 describes the implementation of the pilot system. Finally in Chapter 4 there is a discussion of the unsatisfactory features of the pilot FV system and some conclusions are presented.

CHAPTER 2 TECHNIQUES FOR ACHIEVING FUNCTIONAL VARIABILITY

Reliability in the Presence of Faults

A belief, fundamental to this work, is that faults are always liable to occur within the system under consideration. Given this situation, the reliability of the system as a whole depends on its ability to tolerate faults. When a fault occurs it must be localized so that its effects do not spread to affect other parts of the system and something must be done about the work that has failed to be completed correctly. There are two types of fault to consider: those that develop in a part of the system that had previously been perfect; and those that correspond to intrinsic faults in the design or construction of the system. Hardware problems will usually cause faults of the first type while faults of the second type can be expected to correspond to inadequate software.

It may be possible to design the system so that it automatically corrects for the effect of faults of the first type. For example, if data has been incorrectly processed on faulty hardware the processing could be repeated on alternative hardware provided the original data has been preserved and the fault has been anticipated, detected and localized. However, faults of the second type cannot be corrected by a similar technique because an alternative, correct version does not exist. The long-term action required is to design and install an improved version of the system. But this will take a long time, and in the meantime it will be necessary to abandon the work that has been affected by the fault. If the system is to tolerate this type of fault there must be some method of dealing with abandoned work and the workload must be structured such that a fault affects as little as possible of the workload.

Structuring the Workload

Hence the total workload of the system is divided into units called "jobs". As far as possible the jobs are "transaction jobs"; that is, they are of short duration and are self-contained, meaningful units of work. On the one hand, each job should involve only a small amount of processing and should last only a short time from start to finish so that little is lost if the job fails. On the other hand, each job must be self-contained so that its failure does not affect other jobs, and each job must be meaningful and recognizable so that the significance of its failure can be judged. An air traffic controller having a keyboard and various displays connected to the computer system can regard

his interaction with the system as consisting of transaction jobs. He will initiate some jobs himself. For example, he may input new data concerning an aircraft track and receive an acknowledgement or he may type in an enquiry and obtain some information from the system. If a failure of any sort prevents the completion of such a job then no other job need necessarily be affected. The aim would be to localize the fault to the one job so that no other operator is affected nor is any other work on the same console affected. In such cases recovery action can be left to the operator himself. Since he initiated the work he will be in a good position to decide after a failure whether to try again, forget the matter or report it. Other transaction jobs will be initiated by the system. For example, the information shown on a controller's display might be updated at regular intervals, and the work of doing an update could be a job. The system should have built into it the capability of taking corrective action after failure of such a job.

It is inevitable that some of the work of the system cannot be partitioned into units having the characteristics required of transaction jobs. All such work is attributed to a special job, known as the "owner job" of the system, which continues to exist for as long as the system itself. The owner job thus provides the continuity of the system; it is responsible for starting the transaction jobs and dealing with any failures of transaction jobs. Failure of the owner job is equivalent to failure of the whole system. It is possible for the system to recover from such a failure, but not without an interruption to all work in progress.

Thus the methodology provides two lines of defence for dealing with failure situations. First an attempt is made to confine the effects of an error to a single transaction job and the failure of that job is tolerated by the system. If this is not possible then recovery of the whole system is attempted. Only if the first of these methods succeeds in a good proportion of cases can the methodology as a whole be considered successful. Success, therefore, depends on the following factors:

- i Structuring most of the work in terms of transaction jobs. This becomes a basic principle for designers of the applications programs to follow even if it conflicts with the interests of efficiency.
- ii Isolating jobs from each other. This is the next topic discussed below. It is responsible for the basic software structure proposed.
- iii Detecting faults and taking appropriate corrective action. This is discussed later in this chapter. It affects both the structure and the methodology proposed.

Basic Software Structure

In order to maintain the separation of jobs it is necessary to control the flow of information within the system. This implies that the software consists of two layers, one controlling the other. The layer that provides the control is called the "Kernel". It is supposed to be small, independent of the application programs and completely free of programming errors. The other layer contains all the application-dependent software for doing the actual processing required by the jobs of the system and is called the "Job Base". The Job Base must be structured in a standard manner which is known to the Kernel, although the choice

of the particular standard structure is rather arbitrary. In order to ensure that the Job Base has the correct structure its construction is controlled by a standard program called the "Composer" which accepts the designer's specification of the Job Base contents and casts it in the form required by the Kernel.

Processing on the Job Base is performed by cooperating sequential processes. A "process" has a permanent existence on a Job Base and runs, on a processor allocated by the Kernel, by obeying code that was determined for it when the Job Base was composed. A process must be limited to obeying only that code specified and to operating only on permitted data. This implies some form of hardware storage protection and probably a processor with a "privileged" instruction set (for such operations as setting the protection registers) which a process must not be allowed to obey. Process code consists of non-privileged instructions together with calls of "Kernel Functions". Kernel Functions are facilities provided by the Kernel for processes to make use of. Individual Kernel Functions are introduced one by one throughout the following account.

Each thread of activity through a process is called a "task". A task starts at the beginning of the process code and ends with a call of the Kernel Function EOT (End of Task). For the implementation it was decided to make the processes single-threaded: that is, each process supports only one task at a time. The possibility of having multi-threaded processes is not considered here because the implications have not been sufficiently explored.

Processes can communicate with each other by calling on the Kernel to transmit messages for them. A "message" is simply a few words of information whose meaning is agreed between the sender and the recipient but is of no significance to the Kernel. The Kernel places the message together with various pieces of control information in a package called an "event control block" or "event" for short. The destination of a message is not simply a particular process but rather it is a particular "port" of the process. A port is a named location at which events are queued for a process. Each port belongs to one and only one process, determined when the Job Base is composed, but a process can have more than one port so that it can maintain a number of separate queues of incoming messages. Ports are used only for incoming messages: outgoing messages are not queued nor do they pass through ports belonging to the transmitting process. The principal message-passing Kernel Functions are SEND and WAIT. A process may use SEND by specifying a message and the name of a port to which the message is to go. The Kernel takes immediate action by queuing an event containing the message at the specified port, and the sending process continues with its work. The message is not removed from the queue and supplied to the receiving process until that process requests the Kernel to do this. A process can obtain a message from one of its ports by calling WAIT and specifying the name of the port. The Kernel will then mark the event at the head of the queue as having been examined and supply the message it contains to the process. If the queue is empty then the process will actually wait until a message arrives at the port. In this way processes can synchronize with each other as well as pass information.

The Kernel must know at all times for what job each process is working. A normal transaction job will involve several processes. The job will start at some instant, will be given an identification number and one process will start working for the job. That process can send messages to other processes which may then also start working for the job. Those processes in turn can set other processes working for the job in question. Different processes working for the same job can send messages backwards and forwards between themselves. Eventually

each process will finish working for the job and when the last process finishes the job itself will end. The Kernel keeps track of the information flow by tagging every message with the number of the job that originated the message. Thus when a process receives a message and starts working on it the Kernel knows which job is being worked on.

It is convenient to distinguish two different types of port according to the purpose for which they are used. "Primary ports" are used for messages that initiate a communication between two processes. Thus messages arrive at a primary port asynchronously with respect to the work in progress at the process whose port it is, and the messages queued at a primary port can relate to a number of different jobs. If the process is working for a transaction job then no messages are accepted from a primary port until the current task has ended. "Secondary ports" enable a process working for a transaction job to receive messages and hence synchronize its activity with that of other processes. The messages arriving at or queued at a secondary port should all relate to the job currently using the process whose port it is. This is arranged by allowing nothing but replies to earlier messages to be transmitted to a secondary port. The Kernel function SEND can only be used to send to a primary port and another Kernel Function REPLY is used for communication to a secondary port. When a process working for a transaction job sends a message to a primary port of another process it can specify one of its own secondary ports as a reply port and the Kernel will insert this port name in a field of the event. The recipient then uses the same event for its reply, the message field being overwritten with the reply message.

Control of Data

Transaction jobs must be isolated from one another and this involves careful control of all the data areas in the system. Various types of data area are recognized and these are now considered in turn. First there are the messages. A process that is working for one transaction job must not be allowed to receive a message relating to another job. Since the Kernel controls all message-passing this rule is easily enforced. Second, there are "task data areas", each of which is used by only one process working for one transaction job. By definition, this data cannot affect other jobs, but a mechanism must be found for enforcing the definition. Third there are "process data areas", each of which is local to one process but may contain information carried over from one task to another. Access to process data must be restricted and controlled because it allows the possibility of different jobs interacting with one another. Fourth, there is "global data", i.e. data that belongs to the Job Base as a whole. Again, if jobs are to be isolated their interaction with the global data must be controlled.

When a process is working for just one transaction job and has no access to process data it is said to be working in "application mode"; otherwise it is working in "system mode". The Kernel knows at all times in which mode a process is working. Some processes operate exclusively in application mode and they are declared to be "application processes" when the Job Base is composed. Such a process has no process data. It starts a task by calling the Kernel Function WAKE and specifying the name of its primary port. (An application process has only one primary port). The Kernel locates the event at the head of the queue at the named port, marks the event as examined, supplies the message it contains to the process and notes that the process is working exclusively for the job recorded in the event. During the running of the task the process is not allowed to examine any other events that may

be queued at its primary port. The process may send messages and receive replies at its secondary ports but the rules for secondary ports ensure that it cannot receive messages relating to other jobs.

Processes that need to operate in system mode are declared to be "system processes" when the Job Base is composed. The Kernel allows these processes the privilege of accessing data relating to more than one job at a time and consequently they have the power to prevent the Kernel from isolating jobs. They must be trusted to use this power responsibly because any misuse can affect the whole system. Usually the interaction between different jobs necessary in a system process is slight. An example is a disc-handler process that needs to examine all the requests for servicing in order to decide upon the servicing order that will minimize head movement. A system process starts by using the Kernel Function SCAN (or a variant of this function) to obtain a message from one of its primary ports. The process can choose to have delivered the message from the first event in the queue, or the first relating to a specified job or the highest priority job. When the message has been delivered the process is not considered to be working for the corresponding job: instead it is working for the owner job. The process has access to process data and can examine any number of messages on any of its primary ports. It can send messages (to primary ports only) and it has the privilege of using the Kernel Function RELAY to redirect an event to a new destination.

System mode working is permitted because it is necessary for certain work - often the sort of work that is done by an "operating system". But, because it frustrates the attempt to isolate jobs from each other, it should be used as little as possible. This rule cannot be enforced by the system and so it becomes part of the methodology to be adopted by programmers. As soon as a system process has finished the work that has to be done in system mode it can revert to application mode by calling the Kernel Function AMODE. This causes the process to lose access to its process data and to all events except the one most recently examined. The job of this current event becomes the job the process works for until the end of the task.

The only data whose control has not yet been discussed is the global data. This data is also known as the "state data" because its values describe the state of that part of the environment that is represented within the system. In an air traffic control system it includes flight plans and track lists. Most jobs will require access to the state data and this represents the major interaction between jobs. This interaction is inevitable. A study of the envisaged air traffic control systems concluded that it was not feasible to divide the data-base into parts such that each part is involved only in a subset of the total system functions. However the data is divided into two parts according to whether it is held in main store or on backing store and different methods of protection are used in each case. Data which needs to be accessed frequently and rapidly is held in main store and is termed the "internal state data". Data not immediately required can be held on backing store and is termed "external state data". A job is said to become "critical" when it first alters the internal state data because from then onwards it can affect other jobs via the internal state data. The aim, therefore, is to perform as much as possible of the job's work before it becomes critical. Thus the methodology to be adopted by programmers states that a job should first operate on a local copy of that part of the internal state data it needs and only when the work is finished and all validity checks have proved successful should it update the global data. The Kernel helps by physically preventing write-access to the

internal state data until a job asks for critical access and thereafter allowing the job only a short time before it must end. Updates of the external state data are permitted at any time during a job's life but they are controlled by using audit-trailing techniques: each time a record is updated the previous state of the record and the identity of the job are written to a logging file.

Fault Detection

The structure described so far can be considered to consist of four layers. The order of the layers, as shown in Fig 2.1, is such that each layer uses the layers below. At the bottom is the computer hardware. Above this is the Kernel, then the Job Base and finally the people who use the system. Each layer has different abilities to detect errors: and so, for high reliability, each layer should try to detect whatever errors it can and should be capable of initiating recovery action. Only for the Kernel are detailed error-detection methods proposed because the other layers are dependent on the particular hardware or the particular application.

The computer hardware is capable of detecting errors originating in the hardware: for example, failure of a parity check may be due to a faulty module of core store. It can also detect errors originating in software: for example, an attempt to address the store outside the range of a protection register. The hardware may itself be fault-tolerant to some extent in that it can utilize redundancy to correct some hardware errors. Other errors are, however, reported to the Kernel for action to be taken.

The Kernel controls the whole system and employs a number of techniques to ensure that all is working correctly. It uses the protection provided by the hardware to isolate processes and give them access to only that data they are entitled to access at any given time. It checks the validity of each call of a Kernel Function. For example, in the case of a call of SEND, it checks that the destination port is primary and that the reply port (if one is given) belongs to the calling process. For each secondary port, the Kernel maintains a count of the number of outstanding events that could subsequently arrive at the port (i.e. events whose reply port is the port considered). An error is detected if a process WAITS at a port where this count is zero. The Kernel also maintains a count of the number of events relating to each job and can fault a process that tries to end its current job while there are outstanding events elsewhere in the system. The Kernel controls the allocation of processors to processes and makes several important time checks. The programmer has to specify various time limits within which he expects certain actions to be completed and if a limit is exceeded the Kernel should detect the fault. For a process, there is a limit to the time taken to complete a task and a limit to the amount of processor time that can be used without making a Kernel call. For a job, there are limits to the total life-time and to the life-time after becoming critical. These checks detect loops and deadlock situations on the Job Base. The Kernel also expects to receive regular messages from the owner job of the Job Base as an assurance that the Job Base is still working. In order to exercise these controls the Kernel must be activated frequently by interrupts from a hardware timing device. The Kernel checks that interrupts are being received and acted upon by having two separate routines that check each other and are stimulated by separately derived interrupts. For complete safety these routines must share no hardware; they must use different processors and different modules of main store. The Kernel can periodically exercise the

the various parts of the hardware to check that they are still working.

The Job Base is the first level at which checks can be made on the validity of the data processing. No special techniques are suggested but the programmer is encouraged to build into the program as many checks as seem appropriate. The objective, for each transaction, should be to discover any inconsistencies in the data before gaining write-access to the internal state data.

Ultimately the decision as to whether the system is satisfactory has to be made at the operational level, by the people using the system. They should not need to use any pre-planned tests since these could be built into the Job Base. Rather they should use their essentially human abilities to assess the system in the situation that presents itself.

Dealing with a Fault

No matter at what level a fault is discovered there exists a mechanism for reporting the fault to the Kernel. The hardware uses an interrupt to activate the Kernel. A process on the Job Base calls the Kernel Function FAILJOB and specifies the job affected by the fault and supplies a message indicating the nature of the fault. A user of the system can type in a command to be translated by the Job Base into a call of FAILJOB.

The action taken by the Kernel after a fault has been notified depends on how much of the system has been affected by the fault. There are four cases to consider depending on whether the affected part is one transaction job, the whole Job Base, the Kernel itself or the hardware. If the fault is limited to one job the Kernel stops all further activity on behalf of that job. Any process working for the job is stopped and forced to end its current task when it is next scheduled to run, or when it calls any Kernel Function, or when its time slice expires. Events relating to the job and still queued at ports are discarded when next examined by the Kernel. Thus the Kernel terminates the job but any recovery action is left to the Job Base. On the Job Base there is a special "failure port" where the Kernel queues an event with a message that says which job has failed and why. The failure handling process that services the failure port can then take whatever further action is required. It may maintain a log of failures for future analysis. If the job originated at an operator's console it may be sensible to inform that operator of the nature of the failure. Or it may be appropriate to restart the job automatically provided the input data has been preserved.

If the fault affects the whole Job Base then the owner job is failed and the recovery mechanism is as follows. First the Job Base is reloaded from backing store where it is held in a permanent form known as a "template". Then the internal state data is reset from the copy preserved on backing store at the most recent checkpoint (see next paragraph). At this stage the Job Base is in a standard self-consistent state with no transaction jobs current. On the Job Base there is a special "activate port" where the Kernel now queues an event with a message giving information about the failure that led to the present recovery action. The process that services the activate port is then responsible for completing the recovery action and restarting normal working.

In order to provide checkpoints for use in recovery, the internal state data must be copied to backing store at frequent intervals during normal working. For a present day air traffic control system it may be appropriate to do this every 10 seconds. To ensure the consistency of the data these checkpoints should only

be taken when all jobs that have started to write to the internal state data have ended. This is another reason why jobs are allowed only a short lifetime after reaching this critical state. When a check-point is taken a record is also made of what job numbers correspond to completed jobs. After recovery of this information the external state data can be reset to a state consistent with the internal state data by means of the audit-trail information. Jobs that were in progress at the time of the checkpoint or started after that time can be restarted without danger of updating the data base twice. No jobs need be lost provided an external record has been kept of all jobs started and of all input data. Having rerun any outstanding jobs the system is ready to accept new jobs. The break in service between the time of the failure and the time that normal working is resumed is of crucial importance. An interval of the order of 15 seconds might be appropriate for an air traffic control system.

If the fault is in the Kernel itself a similar recovery technique is used. The Kernel is replaced by a clean copy obtained from backing store. The code needed for this replacement is itself part of the Kernel and so must be protected from corruption by being kept in main store in duplicate and sum-checked at intervals. The new Kernel then proceeds to recover the Job Base as described above. If the fault is in the hardware and the Kernel is distributed in such a way that it can still operate then the Kernel may be able to reconfigure the hardware and then reload the software. If the Kernel cannot operate on the faulty hardware then manual intervention and a cold restart will be needed.

Variability

What has been described so far is a system structure and methodology designed to provide high reliability. However it will now be shown that it forms a basis for Functional Variability. The method used for recovery after failure of the Job Base suggests a way of changing the Job Base. The Job Base is simply replaced by loading a different one from backing store. It can be arranged that should the new version fail then the system would revert to the old version. Using the same mechanism for normal recovery, change over to a new system and reversion to the old system represents an economy of design and results in a robust system since the critical reversion process does not depend on an untested mechanism. What is therefore required is a framework in which new variants of the application can safely be developed and checked out prior to replacing the live version.

Hierarchy of Job Bases

In order to allow testing of new job bases the structure is extended to accommodate more than one job base at a time. The work of the operational system continues to be done by just one job base, known as the "live job base", and other job bases are called "test job bases". The organization of job bases is hierarchical with the live job base senior to all the test job bases. The hierarchical organization is reflected in an hierarchical owning relationship between jobs. Each job base has an owner job which owns all the transaction jobs on that job base and each owner job is itself owned by the owner job of a more senior job base. (An exception is the owner of the most senior job base which cannot be owned in this way.) An owner job has control over the jobs it directly owns to the extent that it can create or destroy them. New transaction jobs come into existence by the owner job using the Kernel Function STARTJOB. They end when they themselves call either ENDJOB to end tidily or FAILJOB to end abortively or when their owner aborts them using FAILJOB. In order to add a new job base to the system a job is first started on an existing

job base. The work of loading the job base from a template on disc is done by this job which then calls the Kernel Function INITIATE to change the job into an owner job and to start the job base running by queuing an event at its activate port. Thus the new job becomes the owner of the new job base but continues to be owned by the owner of the original job base so establishing the hierarchy. An owner job can stop all work on a junior job base by applying FAILJOB to the junior owner job. FAILJOB applied to an owner job stops the work of that job and all jobs junior to it but does not affect the rest of the hierarchy. Only in the case of the owner of the live job base does the Kernel initiate an automatic recovery. In other cases an event is formed containing a message indicating the nature of the failure and this is queued at the failure port of the job base immediately senior to the principal failed job base. An alternative to FAILJOB applied to an owner job is QUIESCE which allows transaction jobs already in progress to end before the activity of the job base is stopped. This ensures that the job ends such that its external state data is consistent with the internal state data.

Resource Sharing

It has been assumed that the live job base is a system that cannot be decomposed into subsystems. Consequently the method used to change the system is to replace the entire live job base with a new version and no means is provided for replacing a part of the live job base. However, it is likely that a given modification will affect only parts of the system and that much of the job base will remain unchanged. Thus when the new version is being run as a test job base much of the test job base will be identical to the live job base. Economies of space and processing power can be made by allowing the test job base to use facilities already present on the live job base. Thus arrangements are made for sharing code, processes and data rather than duplicating them on the test job base. It is a fundamental requirement that the same set of external equipment (disc drives, operator's consoles, digitizers etc) should be potentially available to the live and test systems. Thus the total set of external resources must be sharable between the two systems. Shortage of equipment may require that a particular device such as a disc drive should be usable by both systems. Also information input on certain devices may be required by both systems. The provision for these various types of sharing will now be described. The problem is to provide them in such a way that the test system cannot affect the operation of the live system. Provided code is pure (is not altered by being obeyed) and addresses all data through suitably supplied base values, then the code of the live job base can be shared by the test system with safety. The only affect on the live system is a possible slight delay in accessing the storage module containing the code. A mechanism is needed by which one job base can refer to code on another job base. Each job base has a table called the "code name base" each entry of which contains the name of a code section that can be used from that job base and a pointer to the actual code (which may be on another job base). During construction, a job base inherits a copy of the code name base of its immediate senior. Entries are changed if new code is defined to correspond to old names and new entries are added if new code is defined to correspond to new names. Unchanged entries correspond to code available on some senior job base. Each process is associated with the name of a code section and hence, via the code name base, with the code it is to use.

A process can be shared in the sense that it can accept messages that originate on a junior job base in addition to messages from its own job base. The message

passing Kernel Functions are as previously described but port names are interpreted by the Kernel in terms of a "port name base". Each job base has a port name base constructed in the same way as the code name base so that it contains pointers to ports on the job base in question and on job bases senior to it but on no others. A complicated situation is possible in which a process on one job base is using code on a second job base and is working for a job that belongs on yet a third job base. When a Kernel Function involving a port name is called, the Kernel uses the port name base on the job base to which the current job belongs to translate the name into actual port address. Thus each job base has the power, through its port name base, to determine how its transactions thread the system even when they are being processed by a process or code on a senior job base.

The Kernel control of jobs, which was designed to isolate transaction jobs from one another, now protects the live job base from interference by jobs from the test system. Extra protection must, however, be given to the internal state data. When a job requests write-access to the internal state data, the Kernel must determine what job base the job belongs to and give it access to its own internal state data. Processes working in system mode, which always did have to be careful that their process data could not be corrupted by bad data received, now have to be prepared for bad data from a test system. The principle danger of interference is that an application process could be overloaded by receiving so much work from the test job base that work on the live base is seriously delayed. A system process can overcome this problem by servicing the higher priority jobs from its own job base first or even, if necessary, failing the owner of the test system.

There is no harm in allowing the test system to read data that is present on the live job base. Provided suitable processes exist on the live job base the test system can send request messages to them and receive as replies copies of the data it wants. However, what is wanted to save space is that when the test system tries to read its own data base it should actually read the live system's data base if the data read is identical in the two cases: the data does not then need to be stored in the test data base at all. This might be organized via a directory table similar to the port name base but the details have not been worked out.

Each job base has its own "resource table" and all the external devices that the job base may wish to control are included as resources in this table. Associated with each resource table is a process to manage the allocation of the resources. This process works for the owner of the job base and can dynamically allocate resources as required to the jobs it owns. Thus it can allocate resources to a job that is about to become the owner of a junior job base. In this way the junior job base inherits the resources and can then manage them itself. If the senior job base later finds it needs some of these resources itself then it can retrieve them by failing the owner job of the junior. As an alternative to inheriting resources, a job base can, when being introduced, declare some resources to be new to the system, thus permitting new equipment to be added. It is expected that resources internal to the job base, such as records in the data base, will be declared as new resources and handled by the same management process.

The Nucleus

Originally the Kernel was supposed to be small, being the minimum necessary to support the structure of a job base. Since then the single job base has been

replaced by a hierarchy of job bases with the possibility of adding and removing individual job bases. This has added considerably to the amount of code needed to control this structure. However some of the control work could be done by processes and jobs organized in the manner of a job base. Thus a "Nucleus job base" is introduced to do such work and takes a position at the top of the hierarchy, senior to the live job base. The "Nucleus" is the name given to the Kernel and Nucleus job base combined. The operational system that performs the functions required of its users consists of the live job base plus the Nucleus. The live job base is that part of the system which is variable by the technique of job base replacement, whereas the Nucleus is that part of the system that remains unchanged. The greatest scope for varying the system is achieved by maximizing the amount of the system included in the live job base. Thus the design policy is to include in the Nucleus as little as is necessary to enable the techniques of Function Variability to operate: it must support the job base hierarchy and allow safe job base testing and replacement. Within the Nucleus the division into Kernel and Nucleus job base is not critical. (Indeed the term Kernel has been used rather loosely in this Chapter where Nucleus would have been more correct.) Certain facilities must be provided by the Kernel because they are needed to allow the Nucleus job base to run as a job base. (These do include the Kernel Functions described earlier.) Other facilities may be better included in the Kernel for reasons of efficiency, but otherwise as much as possible should be included on the Nucleus job base where all the techniques for checking reliability will be applied and the code will be available for being shared by other job bases if required. A further advantage of having a Nucleus job base is that the live job base is a junior job base and obeys the same hierarchy rules as any test job base.

Method of Use

The way the system would be used to make changes to the operational system is as follows. The normal situation is that the live job base, supported by the Nucleus, is running and performing the functions it is capable of. At the same time there are various requirements for making (probably minor) alterations to those functions. Programmers would design changes to the live job base intended to satisfy the requirements and they would then want to test these modifications under realistic conditions. They would therefore devise a test job base for this purpose and produce a template for it by using the composer. The composer is a program capable of running as a job base itself; so it would be loaded and run in any convenient junior position in the hierarchy. Having composed a template the programmers would load their job base, probably as a junior to the live job base. It would have its own set of resources, some allocated to it by its senior, some new to it, and it could be used freely without interfering with the operation of the live system. A situation is possible where separate programming teams can be working simultaneously with separate test job bases to test different modifications. Programmers would be free to make modifications to their own job bases. They can do this by removing their job base, composing a new template and loading the new job base. A system could be devised that, without affecting the live system, would allow modifications to be made to a test job base on-line. The work of such a modification could be assigned to the owner job of the job base, but an excessive amount of code would be necessary to provide a useful facility of this kind and so the idea is not considered further.

Once a set of modifications had been tested separately they would be incorporated into a single test job base which would represent the new system that is to replace the live job base. After this too had been thoroughly proved the live job base would be QUIESCED, checkpointed and replaced by the new live job base.

The template of the old live job base would be retained so that if the new system failed it would be possible to revert to the old system. Eventually, if the new system proved satisfactory, it could be designated "trusted" so that, in the event of a system failure, fallback would be to the new trusted system rather than to the old out-of-date system.

When one version of the live job base is replaced by another there is a problem concerning the availability of suitable up-to-date and reliable state data. One situation to consider is where the only changes to the live system are additions: i.e. the new version contains all the code and data fields of the old version together with additional code and data. Fall back to the old version is then a matter of loading the old job base and recovering from the last checkpoint those parts of the internal state data that are needed. Cut-over from the old to the new job base is more difficult because the new fields in the state data will be absent in the recovered checkpoint. However, the time of cut-over can be chosen to coincide with a period of light loading of the system so that time is available for bringing the data-base up to date.

The more general situation is where the change to the system is not simply an addition to the old system. It might be required to remove some code or data that is no longer required or to reorganize the data-base. This situation will arise after a number of modifications that involve only additions because the system will continually grow and so become inefficient. The cut-over to the new system can be as before but in the event of failure fallback will be to the new system. Fallback to the old system is not possible because of the absence of a current checkpoint of data that is compatible with the old system. Thus this type of change is irreversible and therefore not as safe as a change involving additions only.

A technique has been proposed that allows a general change to the system to be reversed and also solves the difficulty at cut-over. The technique is to run the old and new systems in parallel for some time before and after the change. The new system is introduced as a test job base, it is fully manned and run until its state data is up-to-date. Both the live and the test systems are then QUIESCED and checkpointed. The new system is introduced as the live job base and the old system in the test job base position and each system recovers its own data from the checkpoints. The new system carries on with the work of the operational system while the old system continues to be manned and maintain its data-base so that reversion to the old system is possible. Provided common parts of the two systems are not duplicated, the spare capacity that the computer needs to maintain two systems is not necessarily very great. Some inputs from men can be fed to both systems but where the men need to respond differently to the two systems then extra manpower may be necessary. Eventually the new system must be accepted as trusted and the maintenance of the old system discontinued. It is worth noting that some changes are inherently irreversible and so the old system can be discarded immediately the change is made. For example if an external organization declares that certain rules built into the system will be changed from a certain date, or if the computer configuration is changed in such a way that the old system cannot operate.

CHAPTER 3

A PILOT IMPEMENTATION OF THE TECHNIQUES

The original report that presented most of the techniques described in Chapter 2 recommended that, before a full-scale functionally variable system could be built for a specific application, three areas of development should first be pursued.:

- i A pilot implementation of a Nucleus and Composer to demonstrate and evaluate the techniques of Functional Variability.
- ii Analysis and evaluation of methods for representing and accessing the Internal State Data.
- iii Application of the proposed techniques to existing systems.

The first of these was undertaken by a joint CAP/RRE team and the work is described in this chapter. The second and third areas of development have not been investigated.

The Computer Hardware

The original study had said little about the type of computer hardware required for FV. Evidently it must offer some sort of data protection and also for reliability in the event of breakdown of components it must be modular with more than one module of each kind. But it seemed that the choice of a particular computer for the pilot implementation was not critical. The CTL Modular-One computer belonging to RRE's Computer Applications Division was chosen, largely because it could be made available for the project. The computer consists of 3 processors, 7 modules of core store (each of 8k 16-bit words), one exchangeable disc drive and a number of other peripherals.

A full FV system must be based on a multiprocessor computer so that if a processor fails other processors can take over the work. Peripheral devices must be attached in such a way that whichever processor fails they can be serviced by another. The Modular-One computer is not capable of providing the symmetrical type of configuration that is needed because each peripheral device can be connected to only one processor. Nevertheless it was decided that the pilot FV system should be implemented to run on a multiprocessor configuration. This was achieved by adopting the configuration shown in Figure 3.1. The FV system runs in the multiprocessor computer consisting of processors A and B, while a separate system runs in processor C to control the peripherals. Communication between the two systems is via the common store. As far as the FV system is concerned peripherals can be regarded as capable of doing direct transfers of blocks of data to or from core store, the transfers being controlled by control data also in core store. Thus the FV system does not handle interrupts from devices except for the two interval timers attached one to each processor. Processors A and B have an identical view of the core store (and hence of the peripherals) except for a small range of addresses which are incremented by 512 in the case of processor B by a specially built modification box; so that each processor has a small region of private work-space.

Use of available Software

The configuration chosen had in any case to be compatible with the requirements of other work being done on the computer during the period of this project. Most users made use of a general purpose operating system called Minos which ran in processor C and used the whole of the core store. Another project was developing a multiprocessor operating system to run in A and B with a system called Minimos running in C to handle peripherals in the same way as in the FV project. Sufficient collaboration between the projects was possible for Minimos to be constructed so as to satisfy most of the front-end processor requirements of the FV project as they were initially envisaged. The development of Minimos is not regarded as part of the present work.

The FV system is implemented to run directly on the bare machine rather than depending on any other operating system. Nevertheless it was decided to make maximum use of existing software in a supporting role. In particular the Minos operating system was used extensively for program preparation, module testing and postmortem analysis. Thus there are two modes of operation of the machine: either (i) Minos is loaded to run in processor C or (ii) Minimos is loaded to run in processor C together with the FV system in A and B. This technique would be acceptable for the development phase of a full FV system but once operational a full FV system would have to be self-sufficient because of the requirement for continuous operation. In contrast, the pilot FV system is required to run only for a sufficient period to demonstrate the principles, after which the machine can be released. Thus it was also considered permissible to write the composer as a normal program and to run this under Minos when a new job base is to be composed. In a full FV system the composer would itself have to be in the form of a job base so that it could be loaded and run without disturbing the system.

Core Store

A simple approach is adopted towards the use of core store. The core store available to the FV system consists of 5 physical modules each of 8k words. These are set up to provide a continuous range of addresses from 0 to 40k and the boundaries between the physical modules are disregarded. If one module were to fail it would be desirable to reconfigure the remaining modules so that the range of addresses was still continuous (from 0 to 32k). Unfortunately it is not possible to do this by software control because the range of addresses of a module is determined by the physical connection to it. Consequently a physical reconfiguration would be necessary. The alternative approach of constructing the software so that it could operate in an address space having an arbitrary 8k gap was rejected as being too complicated.

Processors

A processor in the Modular-One computer operates in one of two states: normal state or special state. In normal state access to core store is restricted to 3 segments controlled by 3 base/limit registers X, Y and Z. Segment boundaries can only occur at addresses that are multiples of 256 words. The X segment contains the code being executed and may contain read-only data: it is not possible to write into the X segment. Y and Z each provide read/write data segments and differ only in the address modes available: the mode determines what modifier registers can be used and whether addressing is direct or indirect.

In special state the whole of the core store can be accessed. Some address modes still operate as displacements within one of the three segments but absolute addressing is also possible. The code being executed is now in the first 8k absolute rather than being in the X segment. Privileged operations permitted only in special state are the resetting of the X, Y and Z registers and the control of the interface with peripherals.

A transition from normal state to special state is brought about by an interrupt which is of one of three types: (i) an interrupt from a peripheral device, (ii) a violation interrupt resulting from such causes as addressing beyond the limits of a segment, addressing a non-operational core module or failure of the processor's power supply or (iii) a software interrupt caused by obeying an instruction known as an SVC (Supervisor call). In each case the interrupted processor's mode is switched to special state and the contents of the 4 registers that define the working context (not X, Y, Z) are exchanged with the

contents of 4 words of store known as dedicated locations. Each interrupt has its own dedicated locations and so can have its own service routine.

The Programming Language

The programming language chosen for the implementation was Coral 66, a high-level language suitable for real-time work and for which a compiler was available for the Modular-One computer. Machine code inserts are permitted to perform operations that cannot be done directly in Coral and by representing these as macros the readability of the Coral can be maintained. The benefits of a high-level language were considered so great that it was decided to use standard Coral wherever possible and to restrict code inserts to purposes for which they are essential. Only normal state code may be inserted into a Coral program so the special state code required had to be written as a separate machine code program.

A program can be divided into a number of modules for separate compilation and consequently a link-editor is needed to form the compiled modules into a complete program. Again it was decided to use the standard link-editor that was available even though it restricted the way in which the compiled code could be used. The result of using the standard compiler and the standard link-editor is that an object program is produced that requires one X segment, one Y segment and possibly one Z segment. Once loaded the program is expected to run with fixed settings of the X, Y and Z segments. The X segment contains all the code and constants such as character strings and is limited to a maximum size of 8k words. The Y segment contains declared variables and compiler generated workspace. Because of the addressing modes provided by the computer, the compiler also generates dope vectors for accessing data structures and places these in the first 256 words of the Y segment, a region known as Y0. The Z segment is not used unless variables are explicitly declared to be in Z.

The Kernel

The Kernel of the FV system consists of two programs: the SS Kernel operating in special state and the NS Kernel operating in normal state. Operation in special state is so privileged that it cannot be permitted outside the Kernel and consequently the SS Kernel must trap every possible entry into special state. This is easily achieved since entry is always via the dedicated locations and these can direct control to the appropriate special state interrupt service routine. As little processing as possible is done in special state and then control is passed to the NS Kernel for further processing. (See Figure 3.2) After the NS Kernel has done its work the SS Kernel is re-entered to set up the correct context for the next activity of the processor. The NS Kernel has only one entry point and one exit point, so the SS Kernel can be regarded as providing a set of routes for interrupts which fan-in to the NS Kernel and then fan-out to the next activity. The interrupts possible are those from the interval timers (the only peripherals attached), the violation interrupts from the processors and the software interrupts (SVC's). The latter are used to implement calls on Kernel Functions by job base processes. The main function of the SS Kernel is thus to establish the correct processing context on entering and leaving the NS Kernel. Another function is to serialize the use of the NS Kernel to one processor at a time using a lock mechanism. It was decided to make the NS Kernel single-thread in this way for simplicity: it is then not necessary to write re-entrant code or to impose semaphore mechanisms on the use of system variables. On the other hand, the SS Kernel is re-entrant per processor; code is shared but each processor has its own dedicated locations

for interrupts and a small amount of private workspace.

The NS Kernel is written as a normal Coral program. It runs with the X segment set to cover its code and the Y segment set to cover its internally declared variables. Since the Kernel's purpose is to control the processing on job bases it evidently needs to access the job bases and this is accomplished by setting the Kernel's Z segment to cover all the space available for job bases (see Figure 3.3). The Kernel's Z segment also contains a data area known as the Kernel Parameter Block which is used for communication with the SS Kernel. It happens that the principal control blocks and tables that the Kernel maintains need to be examined by several processes on the Nucleus job base. These too are placed in the Kernel's Z segment and the Nucleus processes run with the same setting of Z. The addressing mode used to access the Z segment limits its size to 32k words. The occupancy of core store is as shown in Figure 3.3, with the SS Kernel occupying the lowest addresses.

Job Base Code

Important considerations were how the X, Y and Z segments should be used when a job base process is running and how the code of a job base process should be written. These problems are intimately related. It was decided that one job base should correspond to one program and hence, because of the decision to use the standard compiler and link-editor, all the code of the job base is put together in one X segment. Therefore, when a process runs, the X segment is set to cover the total code of one job base (which is limited to a maximum size of 8k words). The code of each process is represented by a main procedure which can call other procedures, provided they are part of the same program. Thus different processes represented by different main procedures on the same job base can share common subroutines. However it is not possible for a procedure to call a procedure on another job base. Obeying such a procedure call would involve changing the X segment, which would have to be done by the Kernel. But a procedure call is a standard feature of Coral and as such does not invoke the Kernel. A process may use the same code as a process on another job base but the code shared must be the whole code of the process not just a subroutine. One consequence of setting the bounds of the X segment wider than the code of one process is that no code can be regarded as private to a given process. Corrupted or bad code can gain access to any other code on the same job base. Provided special state code is excluded (by trapping all SVC's in the Kernel) then no harm can be done by a process that starts obeying code other than its own. The code itself cannot be overwritten because of the X segment read-only capability and the data of other processes is also protected as described next.

Process and Task Data

It is essential that the process data and task data of each process should be protected from influence by any other process. This is achieved by allocating to each process a separate Y segment. The Y segment of a normal Coral program contains all the data of the whole program and so would be too large for each process to have its own complete Y segment. Data is accessed by its offset from the start of the segment and so it is not possible to narrow the bounds of the Y segment to cover just the data of one process. The solution to this problem involved prohibiting the use of normal procedures and insisting that all process code be written in terms of procedures which are declared as recursive. The compiler does not allocate space for data declared within a recursive

procedure but it addresses the data using a register called W as a modifier within the Y segment and it assumes that W will have been set up at run time as a suitable stack pointer. The compiler implants code to increment the stack point, W, when a recursive procedure is called and decrement it on return, so the Kernel need only set W for the main recursive procedure of each process (which of course is never "called"). Although dope vectors for arrays and tables declared in recursive procedures are not put in Y0 the compiler still uses this region at the start of Y for constants used in addressing some objects: for example a Coral string is stored in X but it is accessed via a word in Y0. Not all processes need access to Y0 but any that do must have the constants from the compiled Y0 reproduced in the correct locations of their own Y segments. Only a small number of words will be involved so the waste of space is not serious. On entry to the main recursive procedure of a process, W should be set to point to the first free location following the Y0 constants.

There are restrictions associated with the use of recursive procedures in Modular-One Coral which make the writing of job base code rather difficult. Recursive procedure declarations cannot be nested so that process or task data declared in a main recursive procedure have to be passed as parameters if they are to be used by a subroutine. A maximum of 256 words of data space can be created by any one recursive procedure so that if more space is needed it is necessary to have extra recursive procedures. Arrays which in normal Coral can have one or two dimensions can only be one-dimensional in a recursive procedure.

A serious deficiency of the FV implementation is that no distinction is made between process data and task data. Both are located in a process' Y segment and are always accessible to a process even though which is accessible should depend on whether the process is in system or application mode. Also, because a process retains the same Y segment throughout its life, all data behaves as process data in that it is preserved from one task to the next. No method was devised for enabling the Kernel to distinguish between process data and task data. It is not possible, for instance, to make use of another segment by using Z in the same way as Y because all data declared in Z will be aggregated into one segment even when declared in recursive procedures.

Kernel Function Calls

It happens that the first 9 words of Y are not used by the Coral program and these are very conveniently available for the communication of data between a process and the Kernel. A Kernel Function call, which normally has parameters, is written as a macro (with parameters) and this expands to code which plants the parameter data in the first words of Y before making an SVC call. The same space is also used for information passed from the Kernel to the process. If a process wishes to keep such information it must copy it elsewhere before making another Kernel call.

Event Control Blocks

The messages used for interprocess communication are another type of data that must be well protected. The method of achieving this protection is to make the Kernel copy the message out of the regime of the transmitting process and later to copy it into the regime of the receiving process. In transit the message may need to be queued and for this purpose it is stored in an event

control block (ECB) accessible only to the Kernel. ECB's are located on each job base so that the number available can be chosen to suit the job base. They are all of equal length to avoid problems of store fragmentation. Free ones are chained together, with the head of the chain located in the Kernel data area (see Fig 3.3). When a fresh ECB is required one is removed from the free chain of the job base to which the relevant job belongs and it is queued at the port for which it is destined.

The length of message required will depend on the application: sometimes the arrival of a message will itself provide the required stimulus without the message containing any information; at the other extreme it may be required to transmit a large block of data. Two lengths of message are supported in the present implementation: a simple message is just one word while a compound message is nine words long. The size of an ECB was chosen to hold the control information such as chaining pointers, job number, reception and reply ports together with a one word message. A compound message is two ECB's with the control information in one and the message in the other. Longer messages are not possible and a different means by which processes can communicate larger amounts of data is described later.

In the method described above each message is copied twice by the Kernel. This could have been rendered unnecessary if the sending and receiving processes could have been given direct access to the message area of an ECB. The Kernel could possibly have controlled the access by setting a process' Z segment to cover the message area. Access to only one message at a time would be provided and a process wishing to handle more messages would have to make copies for itself. Unfortunately segment boundaries can only be placed at addresses that are multiples of 256 so that the smallest segment possible occupies 256 words. This method was therefore rejected because it would have wasted far too much space in view of the large number of short messages expected to be present in the system. A possibility that was not seen at the time would have been to implement this method for long messages in addition to the method actually implemented for short messages.

Transaction Communication Areas

Nevertheless it was recognized that processes would sometimes need to communicate large amounts of data. One possibility would be to send a sequence of messages each containing a small packet of the data. Because this would be so inefficient a method was proposed that introduced data areas of a new type, known as "transaction communication areas". The idea was that a transaction communication area (TCA) should be freely accessible by all processes working for a given transaction job. In order to maintain the independence of jobs required in an FV system it would be necessary to allocate a TCA to only one job at a time and to prevent access by processes working for other jobs. Provided this is done then this new type of data area can be introduced without conflicting with the aims of the FV design. Certainly, processes having simultaneous access to one TCA have the problem of coordinating their use of the area but this problem can be resolved by the processes themselves, using synchronization messages.

As implemented, TCA's each of 256 words, are located on each job base, the number of them being a job base parameter. They are regarded as resources and are allocated to jobs on their job base by the resource manager process. Other processes access them via their Z segment which is set permanently to cover all

the TCA's of the job base. Two deficiencies of the implementation are that a process has physical access to TCA's that have not been allocated to it and a TCA cannot be shared by processes on different job bases even though they are working for the same job. A better system would have been for the resource manager to have informed the Kernel of its allocations and for the Kernel to have dynamically set the Z segment of each process according to what job it was working for.

Internal State Data

Processes also require access to the internal state data (ISD). Write-access can only be granted for a short time at the end of a job's life and the Kernel must be informed so that it knows the seriousness of a failure during this critical time and can check that the job soon ends. Read-access can be granted freely without violating the FV principles but some transactions will require exclusive use of part of the ISD: having read some records a job may require that they are not changed by another job until it has updated them. Therefore the component parts of the ISD are allocated as resources by the resource manager process. Ideally the resource manager should be able to ask the Kernel to give processes the physical capability to access the resources allocated to the job for which they are working. Again the problem is that segment boundaries cannot be set finely enough. The solution adopted is to allow only the resource manager to access the ISD and for it to copy into a TCA any data that another process wants to read. The resource manager also writes into the ISD on behalf of other processes, informing the Kernel before doing so. This method has the disadvantages of requiring extra copying of data and relying on the resource manager to inform the Kernel correctly when jobs go critical. The resource manager needs to access the resource table, the ISD and all the TCA's and it was found convenient to group these together and set the resource manager's Z segment to cover them all. Thus the resource manager has its Z segment set differently from other processes.

Structure of a Job Base

Figure 3.4 shows the component parts of a job base and summarizes the use made of the X, Y and Z segments. The parts occupy successive areas of store, although some space may be wasted because segments have to start on page boundaries (addresses that are multiples of 256). Once established in core store, a job base remains at a fixed location and so addresses of components are stored as absolute offsets from the start of the Kernel's Z. The first component is a control block of 37 words containing important parameters of the job base and pointers to the start of each major component on the job base. The Kernel accesses objects on the job base via these pointers.

Task Control Blocks and Dispatching Strategy

Each process is represented by a task control block (TCB) of 30 words which contains all the static and dynamic information that the Kernel needs in order to control the running of the process. The static information includes the location of its code (which can be on a senior job base) and its ports, the identity of the process and its owner job, the type of process, its priority, time limits for the length of a task and the time spent running between Kernel calls, and initial register settings and access rights. The dynamic information includes the values of the registers (preserved when the Kernel is entered), current status, access rights, times when current time limits expire and the identity of the job currently using the process. A process can be in one of three states: either it is currently using a processor (active) or it is waiting for an event at a port (blocked) or it is waiting for a processor

(ready). The TCB's of ready processes are chained together on dispatcher queues of which there is one for each priority level. (The priority is a parameter of the job base.) The dispatching strategy of the Kernel is to allocate a free processor to the first process on the dispatcher queues taken in order of priority. Unless a time limit is exceeded, which is a fault condition, a process runs until it makes a Kernel call. If the process is ready to run again when the Kernel Function is completed then it is placed on the end of a dispatcher queue, or reactivated, depending on the nature of the Kernel Function.

Port Control Blocks

Each port is represented by a port control block (PCB) of 11 words, which contains the information that the Kernel requires concerning the port. The PCB points to the head and tail of the queue of messages currently waiting at the port. The PCB also contains such information as which message, if any, is currently being serviced by the process owning the port and whether the process is currently blocked, waiting for a message to arrive at the port.

Port Name Base

The port name base contains a list of the PCB addresses of ports that can be referred to by jobs on this job base. The PCB's may be on this job base or on a more senior job base. The port name base is used essentially as described in Chapter 2. However, the names of ports are not actually stored as character strings. Names in the source code are in fact macros that are replaced by integers that represent offsets in a port name base. Each time a port is referred to by a running process the Kernel applies the offset to the appropriate port name base (i.e. that belonging to the job base of the current job) to find the PCB address.

Code Name Base

The code name base contains information about each unit of code that could be the code of a process on this job base. The code can be on this job base or on a more senior one. The information includes the X segment setting and the start address within the segment needed to obey the code, how much data space the code requires (Y segment) and how much preset data in Y0 is required. The code name base is not used during normal running (as the port name base is) but is needed when loading a junior job base which may use code already in core store on a senior job base.

Nucleus Job Base

The Nucleus job base is essentially similar in structure and operation to other job bases but it differs in a number of respects. Several Nucleus processes require privileged capabilities for accessing core store beyond the domain of a normal process. This is provided by giving Nucleus processes a Z segment which is identical to the Kernel's Z segment i.e. one that covers all the job bases (including the Nucleus itself) and some of the control tables in the Kernel area (see Figure 3.3). For simplicity this extended setting of the Z segment is given to Nucleus processes whether or not they need to make use of it. The only exception is the resource manager process: so that it can access its resource table in the same way as a resource manager of any other job base its Z segment is set to cover its resource table as though it were on a normal job base. There are no transaction communication areas on the Nucleus job base, nor is there any internal state data. Thus the only resources belonging to the

Nucleus are external resources such as peripherals and files.

The Nucleus job base comprises ten processes which are described briefly below. Further information about some of them is given later under headings covering particular topics. The first six processes provide services for other job bases, the request for a service being made by sending a message to the principal port of the appropriate process. The remaining four processes perform functions required only by the Nucleus and should not receive messages from other job bases.

- i SOFTWARE CLOCK The message received is a request by the sender to be stimulated at a given clock time. The SOFTWARE CLOCK does this by sending a reply when the requested time is no longer ahead of the system time. (The system time is maintained by the routines dealing with interrupts from the hardware timers.)
- ii ASSURANCE This process must be used periodically by the owner job of every job base. When a request is received each process and each job owned by the owner job is checked for having exhausted any time limit allocated to it and the time limit placed on the owner job is extended.
- iii PREPARE This process is used to introduce a new job base into the current hierarchy of job bases. The process organizes the whole operation, from allocating core for the new job base to causing the new job base processes to run. It may be used by any job base that wants to introduce a new job base as a junior to itself.
- iv DISC HANDLER This process can be used by any other process to transfer a block of data from a Minos file into its own environment (either its Y segment or its Z segment) or vice versa. It is the only way in which processes can use the disc.
- v CHECKPOINT This process is used by an owner job to cause its internal state data to be copied onto a checkpoint file. It is also used in a recovery situation to reset the internal state data from the latest checkpoint.
- vi ROUTER This process deals with input and output on the Nucleus teletypes. Any job base can send a message for out-putting. Inputs are assembled into messages and dealt with according to the initial digit: if it is zero the STARTJOB function is used to start a Nucleus job and send the message to the VALIDATE process; otherwise the digit determines which job base the message is intended for and Kernel Function ROUTE is used to start a job on that job base and send the message to a special port on that job base known as the "router port". The process sends to the SOFTWARE CLOCK to be woken up after a short interval so that it can poll the teletypes for input.

- vii VALIDATE This process receives messages from the ROUTER process corresponding to operator requests for Nucleus facilities. The process checks the legality of the request and initiates the requested action. The only facilities provided in this way are the ability to load a live job base (when none exists already) or to terminate a job base.
- viii ACTIVATE This is the first process to run when the Nucleus job base is loaded into the system. It starts the other processes running and in the case of automatic recovery it initiates the re-loading of a live job base. It sends to the SOFTWARE CLOCK to be woken up at regular intervals in order to send to the ASSURANCE process to assure the Nucleus job base.
- ix FAIL HANDLER This process services the failure port at which the Kernel queues messages notifying failure of jobs owned by the owner of the Nucleus. It uses the ROUTER process to inform operators of failures.
- x RESOURCE MANAGER The only function of this process is to allocate allocatable resources to the live job base when the latter is being loaded. This action is in response to a message from the PREPARE process. (The only resources possessed by the Nucleus are the line printer and two teletypes: these are all marked as allocatable.)

Introduction and Removal of Job Bases

It is the Nucleus PREPARE process that loads a job base into the running FV system, but before it can do this the job base must exist as a template on backing store. The generation of templates is the function of the Composer and, as already explained, this is a program that runs under the Minos operating system rather than within the FV system. The Composer generates templates from job base specifications supplied by programmers and stores them in a template library on disc. The PREPARE process extracts the required template from the library, loads it into store and converts it into a working job base within the hierarchy. The Composer goes as far as possible with the construction of the job base, given that it does not know where it will be located either in the hierarchy or in core store. Thus the structure of the template is essentially a core image of the final job base and the work to be done by the PREPARE process is minimized.

The programmer writes modules of regular Coral in order to express his specification of the job base including its code. These modules are linked together with standard composer modules to yield a program which when run generates the required template. The advantage of combining the particular specification with the general composer functions is that it facilitates cross-referencing between the two. For example, a job base process is specified by a procedure call in which the parameters define properties of the job base process such as the names of its ports, the code it uses, the mode of the process and the time limits to be set on its operations; whereas the procedure body is a standard part of the composer and actually sets up a task control

block on the template. The only references that need to be made to objects on other job bases are to ports or code that will be referenced (in the port name base or code name base) on the job base under which this one will be loaded. In fact port names and code names are macros that are defined to be integers representing offsets in the appropriate name base. Hence all that need be known about the job base's senior is the macro definitions of its port names and code names. The job base specifier has to be careful to supply this list of definitions and to add to it definitions of new port names and code names according to simple rules such that the final effect will be as though the names had actually been stored in the name bases. Several templates might yield a job base that would be a valid senior to the one being composed, so they are listed in "possible senior job bases" on the template (see Fig 3.4).

A new job base can only be introduced into the FV system if an existing job base is willing to support it as a junior to itself and sends a request to the PREPARE process. The work is done on behalf of a job, newly started on the existing job base, which eventually becomes the owner of the new job base. The request to the PREPARE process supplies no information other than the identifier of the job base template. The allocation of core store to job bases is managed by the PREPARE process which maintains the table telling the Kernel where existing job bases are. The template requested is loaded into the first sufficiently large contiguous region found and thereafter remains at the same location in core for the whole life of the job base. Address references within the template which had been relative to the start of the template are modified to be relative to the start of the Kernel's Z segment. The code name base and port name base of the senior job base are used to complete the name bases of the new job base. Messages are sent to the senior's resource manager to acquire any resources that need to be allocated down. Finally the Kernel Function INITIATE is called to finish the job of establishing the job base, by performing those actions that must be synchronized with other Kernel activities.

Because the PREPARE process is a part of the Nucleus job base it is obvious that the Nucleus job base itself cannot be loaded in this way. Instead, a further off-line program (running under Minos) operates on the template of the Nucleus job base to perform those actions normally carried out by PREPARE and INITIATE. This is possible in the case of the Nucleus because it is known in advance what its position in the hierarchy will be and what the running state of the whole system will be. Thus the Nucleus template is a true core image of a fully initialized Nucleus job base and it is a straightforward job for the Kernel to load it and let it run.

The reverse action of removing a job base is supervised by the ASSURANCE process. A job base cannot be removed until all its elements, such as event control blocks, have been detached from the various system queues. It is the Kernel that detaches the job base elements (after a call of the Kernel Function QUIESCE or FAILJOB applied to an owner job) but it does this only as the elements are discovered one by one during the normal work of the Kernel. The ASSURANCE process periodically examines all the elements of the job base and when it finds that all have been detached from the rest of the system it informs the Kernel by calling TERMINATE. The Kernel ends the owner job of the job base and marks the job base as terminated so that the PREPARE process can de-allocate the space when next it runs.

Resource Management

The resources belonging to a job base are represented by entries in the resource table and are managed by the resource manager process of that job base. In

principle each job base is free to choose what sort of entities it wishes to regard as resources and how these should be managed. However in order for the management to be effective it is necessary for the manager to control the physical access to the resources and in practice therefore the amount of freedom is limited by the overall structure of the FV system. The management of various entities is already taken care of by the system, so they cannot be treated as resources. Thus processor time is handled by the Kernel; processor allocation being determined by a simple dispatcher strategy together with parameters such as priority and maximum run time set up when a job base is composed. Core allocation for new job bases is managed by the PREPARE process according to a policy of first come first served. The accessibility of process code and process ports is determined during composition and there is no mechanism for managing it at run time. By the time a job base starts to operate the segment settings for each process have been determined and cannot be changed, so the resource manager cannot pass any physical access capabilities to other processes.

Hence resource management is only achieved by cooperation between various parts of the system and therefore it is strongly recommended that the resource manager uses the standard code provided. This code would be provided on the Nucleus except that the Nucleus code would then exceed 8k words which is the limit to the amount of code possible on one job base. Instead the resource manager code is provided as a separate module which can be incorporated in the live job base while a cut-down version of the code is provided on the Nucleus. The resource manager provided deals with 4 categories of resource which are described below. It operates in response to messages received at its principal port and allocates resources to jobs (never to processes) by making appropriate entries in the resource table. Requests received may be to gain exclusive access or shared access or to release resources. They may refer to all resources, all resources within a category, a particular resource or any resource from a category. When a junior job base is loaded the resource manager will receive requests from the PREPARE process to allocate resources needed on the new job base. When a job fails the Kernel will send a message to the resource manager requesting release of all resources being used by that job.

The 4 categories of resource are as follows:

- i Peripherals This category does not include the disc drive but includes the other devices connected to processor C: these are 2 teletypes, 2 paper tape readers, 1 punch, 1 line printer, and 2 visual displays. The 2 teletypes and the line printer belong initially to the Nucleus but can be allocated to the live job base, and the other devices can be introduced as "new" resources on a job base. Minimos running in processor C actually controls the physical devices but provides for each a logical device which is controlled by a "device control block". A device control block is an area anywhere in core store pointed to by an entry in a unique "interface control block" belonging to the FV Kernel. A process on a job base can gain control of a device by calling the Kernel Function ENGAGE and specifying 2 areas within its own process data area to be used as a device control block and a data transfer buffer. Before entering in the interface control block a pointer to the device control block (and thus connecting the process to

Miniminos) the Kernel checks that the device is not already engaged and also that the device is listed as present in the resource table of the job base to which the current job belongs. A process is disconnected from a device by calling the Kernel function DISENGAGE.

ii Files

There is only one disc drive in the system and this is needed for the work of the Nucleus (for check-pointing and for the library of job base templates). The control of the disc by Miniminos is governed exclusively by the DISC HANDLER process on the Nucleus and the disc drive is not treated as a resource. The DISC HANDLER allows other processes to access Minos files: in response to messages received files can be opened and closed and block transfers can be made to and from a process' data area. Files are identified by name. File names are treated as resources by the resource manager but the system design is deficient in that there is no cross check between the DISC HANDLER and the resource manager on the use of files.

iii ISD resources

The internal state data is considered to be a two-dimensional array or table with rows and columns being the individual resources. The resource manager has absolute control of these resources since ordinary processes have no means of direct access to the area. Read requests are satisfied by copying the data into a TCA belonging to or allocated to the current job. Before writing to the internal state data the resource manager calls the Kernel to set the current job's status to critical.

iv TCA's

Transaction communication areas constitute the fourth category of resource. Their properties have already been described. Although handled as resources there is no physical protection to prevent them being used without the resource manager's knowledge.

Thus although the resource manager handles 4 categories of resource it has physical control of only one, the ISD. In other cases the granting or denial of access can be ignored and the successful operation of the resource manager depends on the cooperation and integrity of the processes using resources. In practice the Kernel and the DISC HANDLER play management roles in controlling access to devices and files respectively.

Timing Assurance

Assurance that the system is still running correctly is based on having two mutually checking routines activated by separate timing device interrupts. The hardware provides two interval timers, one on each processor. Consequently both processors are needed for the cross-checking function. When only one processor is in operation the interrupts received from the one timer are used to activate both routines: the system can operate in this way but there is no independent check that the system is still running.

The two routines are called the Kernel Assurance Routine (KAR) and the Timing Assurance Routine (TAR). The KAR runs at intervals of 1 second. It checks a

flag set by the TAR to verify that the latter has run during the interval. It also inspects some of the critical Kernel code and data, checks that each processor has been used during the interval, checks the time limit of the owner job of the Nucleus and updates the system time. The TAR runs at intervals of 0.2 second. Every sixth entry it checks that the KAR has run since the last check.

Each job base is responsible for periodically sending a message to the ASSURANCE process which will then check whether time limits for objects on that job base have been exceeded. Assurance that each job base is doing this depends on the fact that job bases belong to a hierarchy. When an owner job sends to the ASSURANCE process the latter extends the time limit of the owner job by an amount determined when the job base was composed. Except in the case of the Nucleus, the owner job is itself owned by the owner job of a more senior job base and hence its time limit will be examined when the senior job base sends to the ASSURANCE process. As stated earlier, the time limit of the owner of the Nucleus is checked by the KAR.

Failure and Recovery

Failure and recovery are treated much as proposed in Chapter 2 but a few extra details should be given. After a job has failed, the Kernel places a message at the failure port of the appropriate job base and further action is a matter for the process servicing that port. If, however, the job owned any resources it will be necessary to inform the resource manager that the job has failed. In order to save time and effort therefore the Kernel automatically sends a "release all resources" message to the appropriate resource manager.

Automatic recovery of the system is instigated by a failure of the owner of the live job base, the owner of the Nucleus or the Kernel. The recovery logic treats these cases in the same way and reloads the Kernel, Nucleus and live job base. Although some of this may be unnecessary it is simpler to have just one logic, and little time is wasted in reloading the Kernel or Nucleus. The "Fallback" routine, although part of the Kernel, is probably uncorrupted since it is checked frequently by the KAR. This is used to read from disc a program called "Startover". Startover reads in a clean copy of the Kernel and activates it as though a power-on interrupt had been received. The Kernel reads in a clean copy of the Nucleus job base and starts it with a message at its activate port. The Nucleus interprets this message as a job base identifier and duly loads the appropriate job base as the live job base. It is the responsibility of the live job base to recover its checkpointed data and to restart normal working. Information that has to be carried over from the failed system to the recovered system has to be carefully preserved throughout this sequence of operations: this includes the reason for failure and the identifier of the job base to be used for recovery.

The automatic recovery logic is complicated by the need to prevent interference by processors other than the one instigating the recovery. The strategy adopted is that the processor executing the Fallback routine in the NS Kernel releases the Kernel lock but holds the other processor on an inner lock at the start of the Fallback routine. This lock is released only when the Startover program is about to be activated. Startover in turn is protected by another lock which the processor executing the code releases only when the new Kernel is about to be activated. This solution is unsatisfactory in that the processor carrying out the recovery has to wait for the other processor to enter the Kernel before the Startover program can be loaded. There is no guarantee that

the other processor has not gone out of control. An interprocessor interrupt, which was not available, would be required to improve the position.

Live and Test Job Bases

The pilot implementation provides the Nucleus and Composer to support a functionally variable system. Although there was no requirement to produce a specific live job base, thereby completing a system, it was necessary to write application job bases in order to demonstrate that the Nucleus and Composer do provide the facilities intended. The five demonstrations or tests that were carried out are described below. The job base construction and testing necessary for these demonstrations constitutes the only experience of using an FV system.

System test 1 The primary aim of this test was to exercise the basic Kernel facilities using a primitive job base as a test driver. The test demonstrated the process management and intercommunications mechanisms, the dispatching strategy, job management and failure localization, and basic job base composition.

System test 2 The purpose was to exercise the basic Nucleus job base and to test the ability of the Kernel to support a hierarchy of job bases. Facilities demonstrated were timing assurance mechanisms, detection and handling of failures, true multiprocessing, the ability of the Nucleus to load, activate, support and remove an application job base, the ability of a junior base to use objects (code and processes) of a senior, and generalized job base composition.

System test 3 The purpose was to test the management of resources and the recovery and variability (cutover) mechanisms. The facilities demonstrated were the allocation and sharing of resources (particularly the elements making up the Internal State Data of an application), checkpointing and reinstatement of the ISD, the identification of errors necessitating a system restart, the Cold Start sequence (an operator controlled reload of the system), the Hot Start sequence (automatic fallback and recovery) and the ability to terminate the live application job base and replace it by another version and continue from the point in service reached.

System test 4 The purpose was to test the capabilities of the complete Nucleus. The facilities demonstrated (additional to those covered in the previous system tests) were the ability of the Nucleus to support parallel operation of a live and test application, the ability of the test base to share various objects on the live base and the Nucleus while not prejudicing the live service, failure of the test base without disturbing the live job base, termination of a hierarchy of job bases, simultaneous checkpointing of live and test job bases, and cutover (termination of the live base - and hence the test base - and continuation of the service from the point reached by the live base and re-instatement of the test base).

System test 5 This test involved what was known during the development as the "RRE application". This is a job base that performs in a manner that has some similarities to a system providing information for air traffic control. The system maintains in a data base the current positions and velocities of "aircraft" and updates the positions at intervals. Labelled positions of aircraft are displayed on a VDU. Teletypes are used to input data and to control the system. Using this as a live job base the full test involves (i) introducing a test job base that performs in a similar but different manner to the live and shares facilities with the live, (ii) cutting

over to the new system and falling back to the old system as required and (iii) removing the old system so that the new system carries on as the live system and falls back to itself after a failure.

Size of the Run-time System

The software of the run-time system proved to occupy more core store than had been anticipated. The sizes of the major components are indicated on Figure 3.3. It can be seen that the Nucleus (i.e. the Kernel plus the Nucleus job base) occupies 24k words of the 40k available to the FV processors, so that only 16k remains for the live and test job bases. Thus the amount of free core store is so small that there is no real possibility of making use of a hierarchy of test job bases. If equal space is allowed for one test job base and the live job base then each can occupy 8k. This is too small for any practical system: it should be remembered that the Nucleus is not an operating system and the live job base must provide its own input/output routines for example. Functional Variability applies to only a minor part of the operational system since the major part of the system, the Nucleus, is invariant.

CHAPTER 4

DISCUSSION

The objectives of the implementation have, in broad terms, been achieved. A Nucleus and Composer have been implemented which are capable of supporting a Functionally Variable system as described in Chapter 2. Most of the proposals made in Chapter 2 have been incorporated in the implementation. The Nucleus runs on a bare machine (rather than depending on a general purpose operating system) and makes use of a multiprocessor configuration.

However, this achievement must be qualified by a discussion of the rather large number of unsatisfactory features of the implemented system. It is convenient to divide these into four classes. First there are the accepted limitations corresponding to deliberate simplification of the objectives. Second there are the undesirable features that resulted from the decision to use the available hardware and Coral compiler. Third there are the defects due to poor design of the software. Fourth there are the problems inherent in the proposals made in Chapter 2. The divisions between the classes are not always exact but they form a useful basis for discussion.

I Accepted Limitations

The aim of the pilot implementation was to demonstrate the fundamental principles of FV using available hardware and a limited amount of effort. Consequently decisions were made that would reduce the amount of work needed to implement a system. Inevitably the pilot implementation is more limited than a full FV system, but the limitations listed here are considered to be acceptable because they do not interfere with the study of FV principles.

- 1 No attempt was made to provide a complete system for a specific application: only the Nucleus and Composer were produced, together with sufficient applications programs to demonstrate their facilities. Thus many of the features that a live job base would be expected to provide were not implemented. These include the handling of external state data and the recovery action to be taken after failure of transaction jobs.

- 2 No attempt was made to provide a high level of reliability by using hardware redundancy. Clearly the available hardware would not have been

sufficient to do this: for example there is only one disc drive and both processors depend on the same store module zero.

The Kernel does not test or reconfigure the hardware. If a permanent fault develops in a hardware module it is likely to lead to repeated failure and recovery of the system and human intervention is necessary first to locate the faulty unit and then to reconfigure the system. The only hardware effects the system will automatically tolerate are a power-off on one processor (which will cause failure and recovery to the last checkpoint) and failure or removal of a core store module used only by test job bases (which will cause failure of the test job bases only).

3 The FV system does not handle external interrupts (except those from the timers). The interrupts from peripherals are dealt with by the Minimos front-end processor which is outside the scope of FV. The reasons for this have already been discussed (see page 16).

4 The normal state Kernel is single thread. This decision should have little effect on the efficiency of operation of the system since the time spent in the Kernel is small.

5 The Composer is not designed to run as a job base. Consequently composition has to be done first under Minos before running any demonstration that involves introducing a job base into the FV system.

6 It was decided to use the standard Coral compiler and link-editor. This gave rise to no difficulties in writing the Kernel although some parts could have been made more efficient by using more code inserts. However the writing of the job base code is made awkward by the restrictions associated with the exclusive use of recursive procedures (see page 20). Also it is not possible for different job bases to share code at the procedure level. (They can only share the code of a whole process.)

II Undesirable features due to the hardware and language

The decisions to use the available Modular One hardware configuration and the Coral compiler resulted in certain accepted limitations of the system, as discussed above. However, these same decisions caused the system to be unsatisfactory in a number of ways that were not anticipated and accepted from the start. These are discussed here. In order to overcome these deficiencies it would seem necessary to choose more suitable hardware and/or a more suitable language compiler.

1 No distinction is made between process data and task data. This is a serious deficiency in that it upsets the principle of preventing interactions between different transaction jobs. It results from the decision to use Coral and has already been discussed (see page 20).

2 There is no continuity in the record of time across a failure and recovery of the system. This is because there is no real hardware clock that continues to measure the passage of time. The two interval timers require operational software to keep them running. As implemented, the system time maintained by the Kernel is reset to zero during a system recovery, which is not at all satisfactory. The problem could be overcome by using a hardware clock that continues to measure real time throughout a system failure.

3 It is possible to write a poor resource manager process that updates the internal state data without obtaining critical access rights from the Kernel. This is because the resource manager has full access to the internal state data and does not need to apply to the Kernel for access capabilities. Hardware with a more sophisticated capability mechanism would be required in order to solve this problem. For the pilot implementation it is recommended that the problem is avoided by using the trusted code provided for the resource manager, although this denies the possibility of regarding the resource manager as functionally variable.

III Defects due to software design

The limitations listed here are defects in the way the software was designed. They could all have been overcome by more careful design of the software, without changing the hardware, the Coral compiler or the link-editor.

1 Device control blocks, which are accessible to processes that use peripheral devices, contain control information whose corruption could have serious consequences for the integrity of the system. The inclusion within one control block of elements that are used directly by a process and elements that should only be used by the Kernel and Minimos is a deficiency of the design of the interface between Minimos and the FV system. A process thus has access to a pointer giving the absolute address of the data area to be used for peripheral transfers. If it accidentally or maliciously changes this pointer some other area (even on a different job base) can be overwritten. This deficiency could have been overcome by separating those data fields in the device control blocks that need to be accessed by a process from those that do not.

2 The misuse of transaction communication areas allows jobs to interact in a manner contrary to the principles of FV. These areas are intended to be used for communication between different processes working for the same job. The resource manager can be used to reserve a TCA for a particular job but each process has access to all the TCA's on its job base and the system provides no checks that the reservations are being honoured. A solution to this problem has already been proposed (see page 22).

3 Transaction communication areas can be accessed only by processes on their own job base and consequently a process on a more senior job base cannot be used to process data in a TCA. Indeed if this is attempted the affects can be confusing. Suppose a process working for a job on a more junior job base sends a message to the resource manager to obtain some internal state data. The junior's port name base will be used to find the destination of the message which will therefore be the junior's resource manager. As expected the resource manager will access the junior's internal state data, copy some of it into a TCA and reply with the address of the TCA relative to the start of the Z segment. However if the senior process tries to access the TCA it will apply the address as an offset from its own Z segment and will access something quite different. The confusion arises by thinking of a TCA as an extra-long message; messages can be sent to senior job bases. Rather than implementing TCA's it would have been better to have implemented the extra-long message as discussed on page 21.

4 New jobs on the live job base can be prevented from starting by the presence of too many jobs on test job bases. This is because there is a

fixed number of slots in the job table to hold information about current jobs. A better scheme would have been for each job base to have its own job table.

5 The multiprocessor aspects of the implementation do not work reliably. If power to one of the FV processors is switched off the system should continue to operate provided one processor is operational. However if this is done it sometimes causes a failure of the system from which it does not recover. Presumably this is due to some error in the logic for processor synchronization. Sometimes automatic recovery after a failure of the FV system is not possible because of lack of synchronization with the Minimos system. This could have been overcome by providing an emergency channel on the interface through which to notify Minimos of a breakdown of the FV system.

6 In the case of single processor operation, a process that monopolizes the processor by looping in its code will cause the current time limit of the Nucleus' owner job to expire and hence an automatic recovery of the whole system. This is because the assurance process does not run and hence it is not discovered that the looping process has exceeded its allotted processor time. The routine activated by one of the timer interrupts checks the time limit of the Nucleus' owner job and this eventually expires. The problem could be overcome by implementing some form of time slicing so that timer interrupts cause the looping process to stop and hence allow the assurance process to run.

7 Checks that certain time limits have not been exceeded are made too infrequently. This is because most checking is left to the assurance process and the checks involve limits differing by one or two orders of magnitude. The time for which a process is allowed to use a processor without making a Kernel call may be say 100 ms and it is therefore not a useful safeguard to make such a check say every 5 s. However to run the assurance process more often would entail unacceptable overheads. A better safeguard and a more efficient check would be provided if the Kernel were to check processor time consumed by the current process.

IV Problems inherent in the FV proposals

During the design, implementation and use of the pilot system a number of problems were encountered which can be identified with inadequacies in the FV proposals as described in Chapter 2. Some of these concern matters where the proposals provide insufficient guidance. Other problems are such that their solution would seem to require a change to the proposals of Chapter 2, although what changes are needed has not been determined.

1 Resource management

The original proposals provide insufficient guidance on the handling of resources. They do not specify what entities are to be treated as resources. The four categories of resource in the implementation were chosen rather arbitrarily from a large number of possibilities. Other possible resources that might have been considered include processor time, storage, code, process ports, event control blocks and slots in the job table. The need to manage resources was recognized and the resource manager processes were

suggested with the intention of fulfilling this need. However, in order to manage a resource effectively it is necessary to have control of the physical access to it, and a mechanism for providing this control was not discussed. Physical access should surely be granted by the Kernel. This is not done in the implementation however and consequently the control of resources is ineffectual. The resource manager can be bypassed or ignored by a process trying to access a resource from any of the categories devices, files or TCA's and the resource manager only succeeds in retaining control of the fourth category, internal state data elements, by not allowing other processes to have direct access to such resources.

2 The test job base interface

An operational system designed according to the proposals of Chapter 2 may fail to be functionally variable because the live job base does not provide the facilities needed to support a test job base. Since the Nucleus is intended to provide what is needed to support an FV system it would be expected that any live job base that could run under the Nucleus would be variable. However, variability is achieved by first testing a new version running as a junior to the live job base and this requires support from the live job base, which is only available if the required support was anticipated when the live job base was constructed. First the live job base must cooperate in order for a test job base to be loaded as a junior. Second, if a test job base is to be allowed to use processes on the live job base then the latter must be constructed in such a way that it can cope with unexpected and possibly misleading messages from a junior job base, whereas its ability to do so may never have been tested. Third, since no general method has been suggested by which a test job base can access data on the live job base such access can only be obtained with the cooperation of the live job base - it could supply data in reply to messages requesting such a service. The problem is that if the live job base does not already provide the necessary support then the technique of functional variability is not available to make the changes that will provide it.

3 The role of the Composer

A new job base that has been tested in a junior position cannot be introduced as a replacement for the current live job base without first being changed. This is because in the test position it will have used facilities provided by its senior, the live job base, which will not be available when it becomes the live job base. Since there is no formal specification of what facilities of the senior are used it is difficult to see how an automatic program such as the Composer could modify the tested job base so that it could provide those facilities within itself.

A new job base may be unsuitable as regards the FV methodology for reliability even though it has been shown in tests to perform well. For example (i) it may not checkpoint its internal state data with sufficient frequency (ii) it may have unsuitable values set for the various time limits that the Kernel checks are not exceeded or (iii) it may specify that fallback shall be to the old system with a checkpoint from the new system even though the change is in fact irreversible. Some checking of suitability could be done by the Composer but the mechanism and criteria to be adopted is not clear. A determination of whether a change is reversible

or not depends on detailed knowledge of the structure of the data-base and the processing of the data. It is difficult to see how a fixed program such as the Composer can make such assessments without imposing unacceptable constraints on the flexibility of a job base.

4 The backup system

After an irreversible change has been made the proposals suggested that the old system could be maintained in the test job base position so that its data-base would be kept up to date and it would be available as a backup should the new system fail. However, this provides a very insecure backup system since there is no automatic recovery of a test job base after a failure and so the old job base may be failed as a result of some error in the new live system.

5 Programming problems

The FV system structure and methodology impose constraints upon the design of the live job base. Difficulties were experienced in designing job bases to run on the pilot implementation because the proposals gave too little advice on how to work within the constraints. The necessity for processes to communicate only via messages led to the writing of obscure code for the generation and interpretation of messages. The complexity of the rules concerning the validity of message passing functions according to circumstances resulted in the rules being frequently broken. When parallel activities are used within one transaction job it is necessary to synchronize the activities before ending the job and this seemed difficult to achieve without causing a bottleneck. A live job base contains functions that in a conventional system would be regarded as operating system functions and also application program functions but with no sharp demarcation between the two. Applications programmers are used to working in an environment where operating system functions are provided as supervisor calls, but in an FV system supervisor calls are used for Kernel functions and operating system functions provided on the job base can only be provided in terms of messages between processes.

6 Error diagnosis

The proposals of Chapter 2 make no provisions for diagnostic aids and in fact the implementation provided only very basic ones. Errors result in job failures which are notified by the Kernel to the appropriate failure handling process. Thus the test job base itself is expected to handle failures of its own transaction jobs, using the live job base failure logic if so desire, and complete failure of the test system has to be handled by the live system. The Nucleus job base failure handling is rather primitive, at best printing out the immediate reason for the failure which is rarely much help in tracing the primary error. The Kernel of the pilot implementation maintains a file in which each Kernel action is monitored by recording a reference number: diagnosis consists of halting the whole system and tediously analysing this file and a core dump. Monitoring on a selective basis is clearly required.

Conclusions

Some of the difficulties encountered in the implementation were due to the choice of hardware and the choice of language for writing job bases. Although a detailed study of what is required has not been made it is

possible to draw some conclusions from the experience of the present work. To cope with hardware failures peripheral devices should be capable of being serviced equally by any of the main processors and it would be helpful if the address range covered by a store module could be changed under software control. The required storage protection demands more than the 3 segments provided on the Modular-One, although exactly how many is not clear. Fine control is needed of the range covered by a segment: the unit of 256 words on the Modular-One is much too large and control at the single word level would be highly advantageous. A suitable language in which to program job bases would have to be specially designed and have a special compiler. Ideally there would be an object type in the language corresponding to each type of object recognized by the FV methodology and the compiler would check, as far as is possible before run time, that the objects are only used in the permitted way. Object types such as PORT could be implemented in Algol 68 by MODE PORT or in Coral by using macros (although in this case there is no mode checking). However, object types such as PROCESSDATA have to be built into the language so that they can be mapped correctly onto the hardware protection segments.

The main value of the implementation work has been the light it has thrown on the concepts implicit in the methodology proposed. The difficulties of maintaining, modifying or using the pilot system make it unattractive as a tool for further investigations, and it is not currently being maintained. If the facilities provided by the Nucleus for the job base programmer were to be further experimented with it would probably be better to simulate them on a large general purpose computer using a more powerful language (e.g. Algol 68) and using the diagnostic aids available on such a computer.

However, we do not think that an improved implementation of the existing design is what is now required. Future work should concentrate on improving the FV methodology in such a way as to overcome the problems revealed by the present implementation.

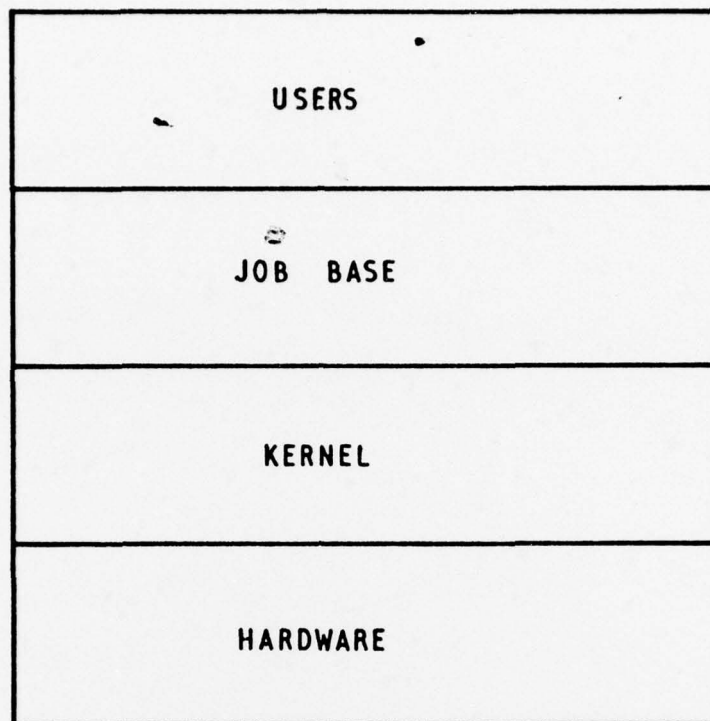


FIG. 2-1
LAYERS IN THE BASIC STRUCTURE.

RR/R 3/20691

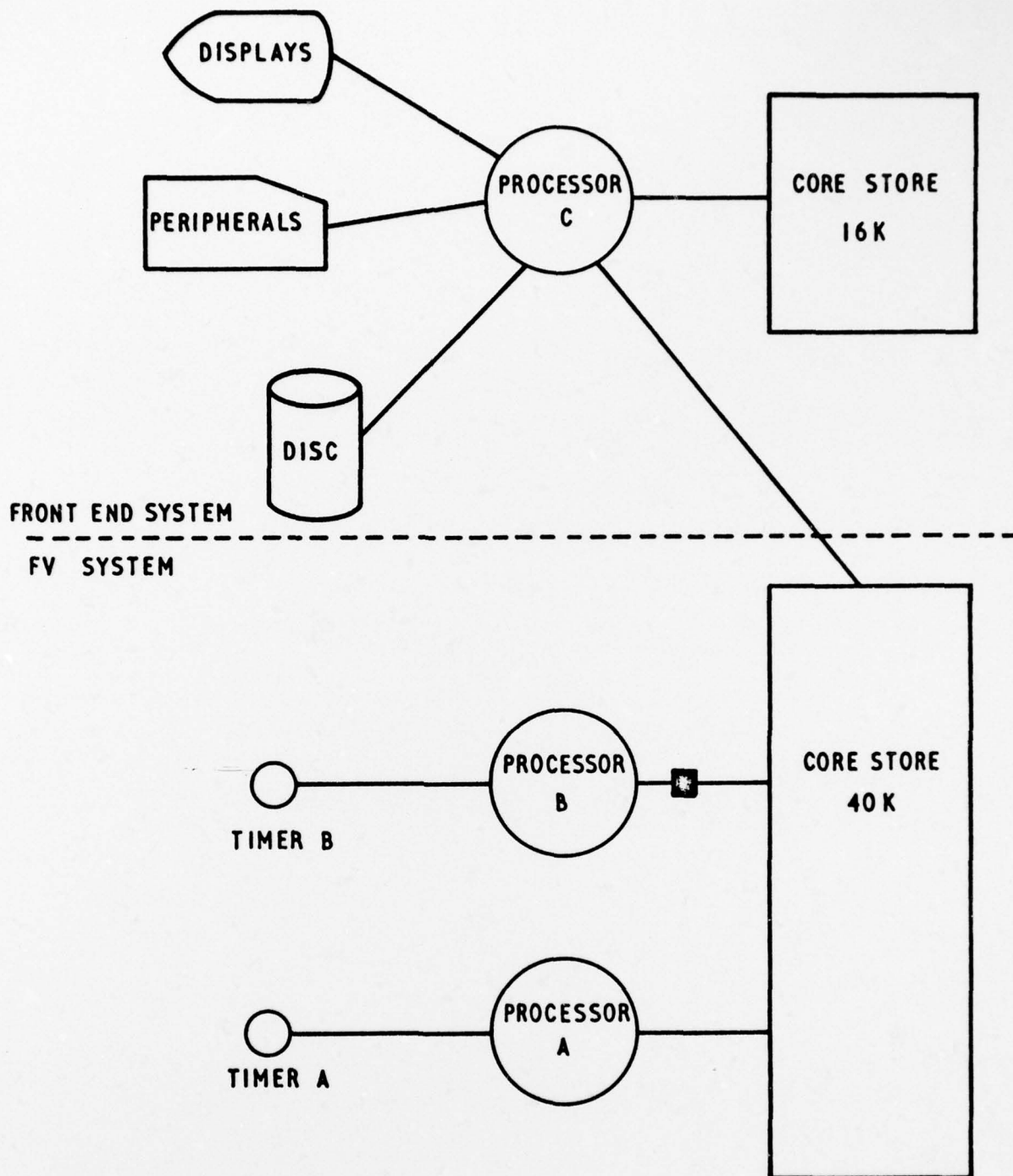


FIG. 3-1
CONFIGURATION OF THE COMPUTER.

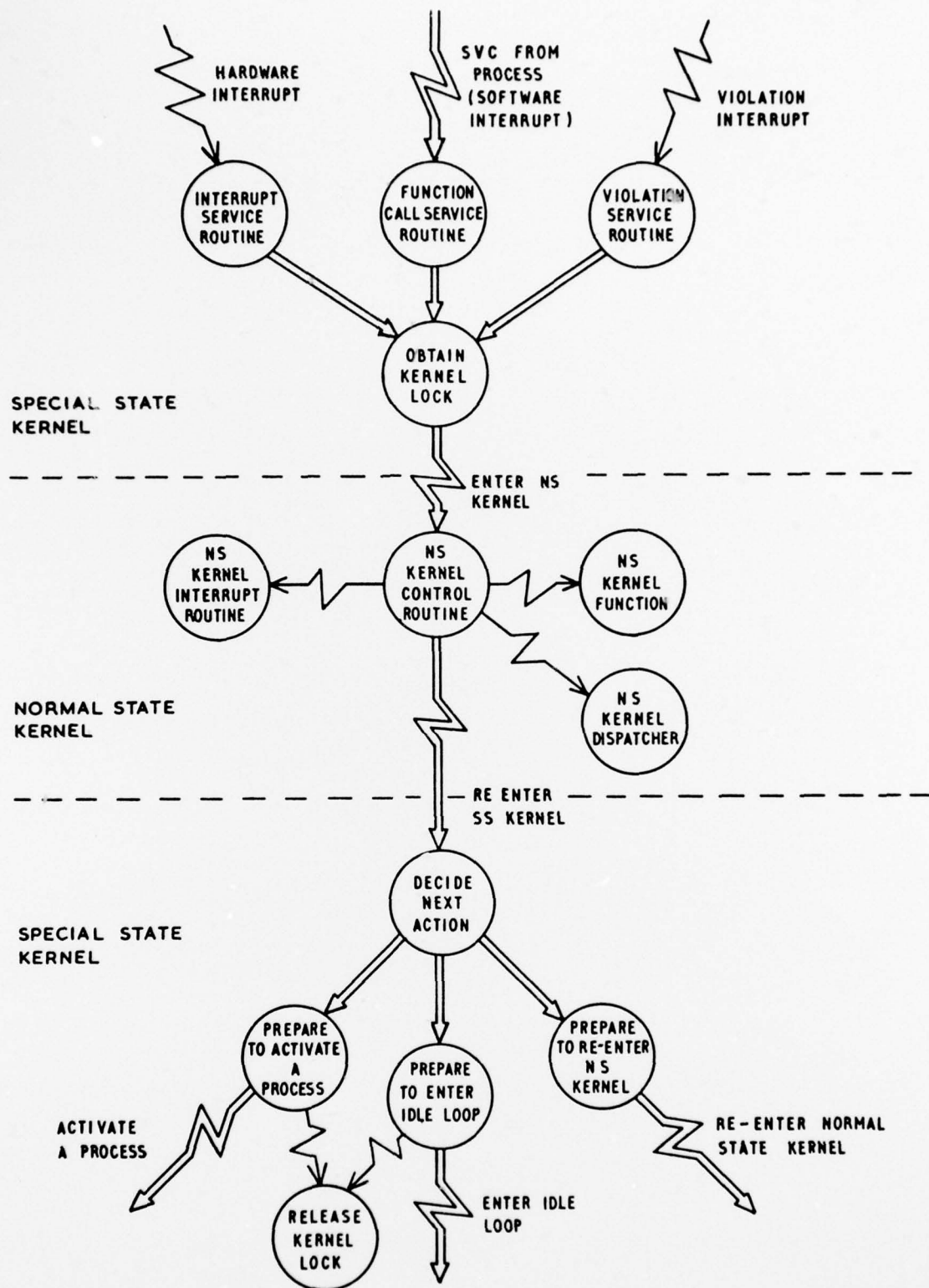
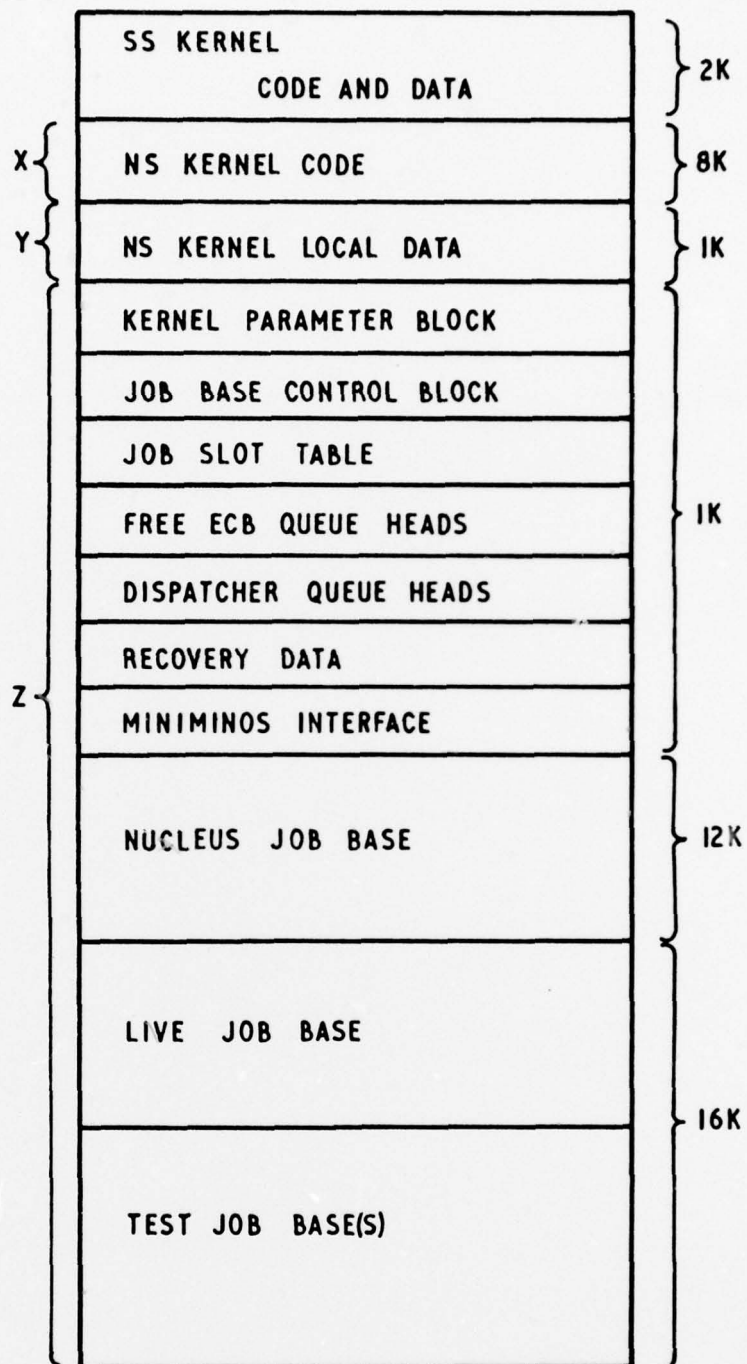


FIG. 3-2

RELATION BETWEEN NS AND SS KERNEL.

NS KERNEL SEGMENTSWORDS OF COREFIG. 3-3

LAYOUT IN CORE STORE, SHOWING THE KERNEL'S
USE OF SEGMENTS AND THE CORE UTILISATION.

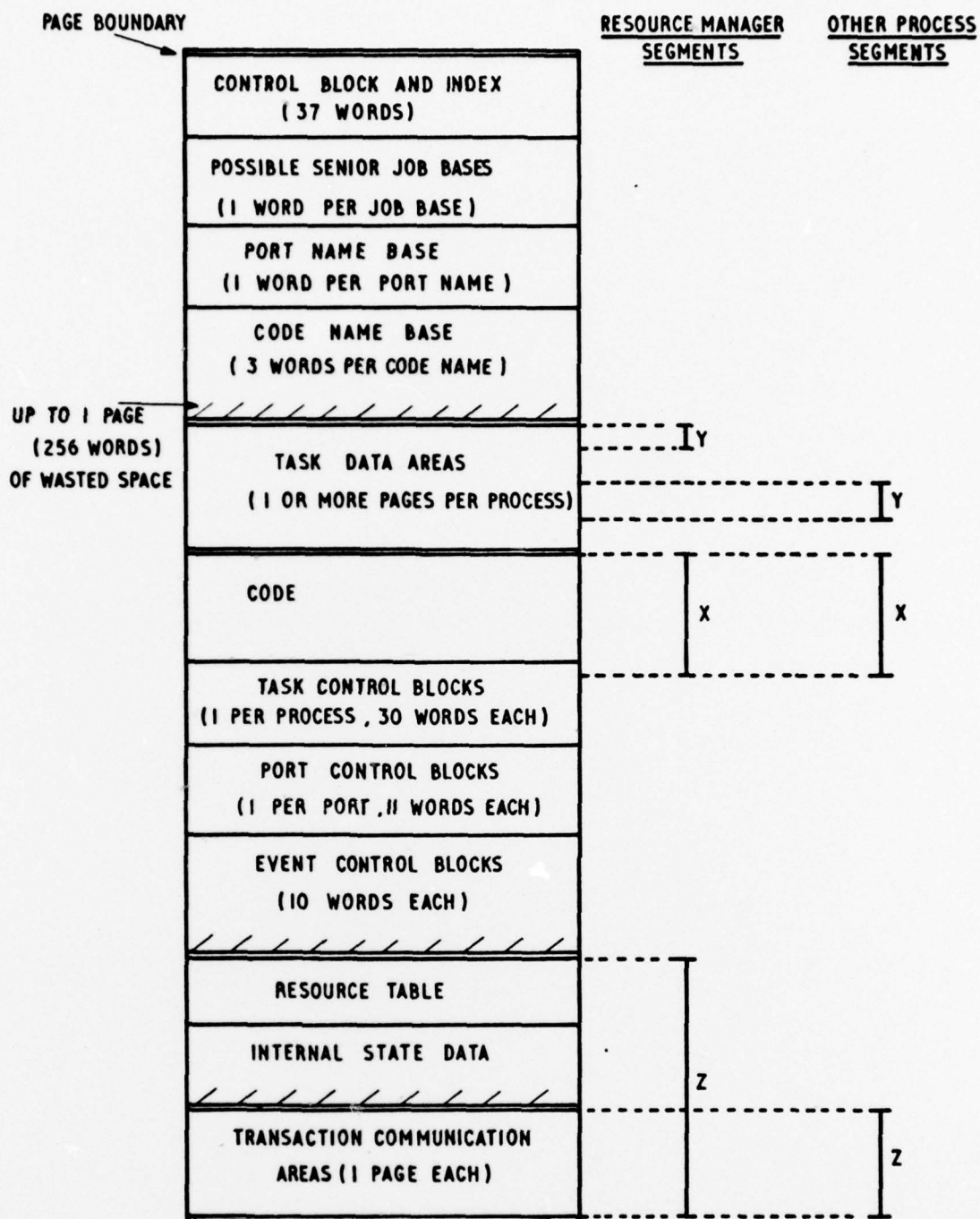


FIG. 3-4

COMPONENTS OF A JOB BASE, SHOWING LAYOUT AND
PROCESS' USE OF SEGMENTS.


APPENDIX Index of terms

Activate port	10
AMODE	8
Application mode	7
Application processes	7
Code name base	12
Composer	6
Critical	8
DISENGAGE	28
ENDJOB	11
ENGAGE	27
EOT	6
Event	6
External state data	8
FAILJOB	11
Failure port	10
Global data	7
INITIATE	12
Internal state data	8
Job Base	5
Jobs	4
Kernel	5
Kernel Assurance Routine	28
Kernel Functions	6
Live job base	11
Message	6
Nucleus	14
Nucleus job base	14
Owner job	5
Port	6
Port name base	13
Primary port	7
Process	6
Process data areas	7
QUIESCE	12
RELAY	8
REPLY	7
Resource table	13
ROUTE	24
Router port	24
SCAN	8
Secondary ports	7
SEND	6
STARTJOB	11
System mode	7
System processes	8
Task	6
Task data areas	7
Template	10
TERMINATE	26
Test job bases	11
Transaction communication areas	21
Transaction jobs	4
WAIT	6
WAKE	7

DOCUMENT CONTROL SHEET

Overall security classification of sheet UNCLASSIFIED

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference RSRE TN 789	3. Agency Reference	4. Report Security Classification Unclassified	
5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location MOD PE Royal Signals and Radar Establishment, St Andrews Road, Malvern, Worcestershire			
5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location -			
7. Title A report on software functional variability				
7a. Title in Foreign Language (in the case of translations) -				
7b. Presented at (for conference papers) Title, place and date of conference -				
8. Author 1 Surname, initials ALLEN A D	9(a) Author 2 CROWTHER J H	9(b) Authors 3,4... -	10. Date January 1977	pp. ref.
11. Contract Number	12. Period	13. Project	14. Other Reference	
15. Distribution statement 				
Descriptors (or keywords) Software, Testing, Air Traffic Control, Reliability Programming Functional variability continue on separate piece of paper				
Abstract Proposals are made for a software structure that provides a high degree of fault tolerance and is appropriate for a large, complex, real-time system such as an air traffic control system. Further proposals describe how the structure can be extended to allow software development work to be carried out in parallel with operational use of the system. A pilot system has been implemented on a Modular-One multiprocessor computer in order to test some of the concepts. An assessment of the work indicates that it has been only partly successful and some problems of a fundamental nature remain to be investigated.				