REAL-TIME INTERACTIVE MULTIPROGRAMMING

by

ANTHONY DOUGLAS HEHER

PRETORIA

1978

SUBMITTED IN PARTIAL FULFILMENT OF THE REQUIREMENTS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY IN THE DEPARTMENT OF ELECTRICAL ENGINEERING, UNIVERSITY OF NATAL.

A B S T R A C T

This thesis describes a new method of constructing a real-time interactive software system for a minicomputer to enable the interactive facilities to be extended and improved in a multi-tasking environment which supports structured programming concepts. A memory management technique called Software Virtual Memory Management, which is implemented entirely in software, is used to extend the concept of hardware virtual memory management. This extension unifies the concepts of memory space allocation and control and of file system management, resulting in a system which is simple and safe for the application oriented user. The memory management structures are also used to provide exceptional protection facilities. A number of users can work interactively, using a high-level structured language in a multi-tasking environ= ment, with very secure access to shared data bases. A system is described which illustrates these concepts. This system is implemented using an interpreter and significant improvements in the performance of interpretive systems are shown to be possible using the structures presented. The system has been implemented on a Varian minicomputer as well as on a microprogrammable micro= processor. The virtual memory technique has been shown to work with a variety of bulk storage devices and should be particularly suitable for use with recent bulk storage developments such as bubble memory and charge coupled devices. A detailed comparison of the performance of the system vis-a-vis that of a FORTRAN based system executing in-line code with swapping has been performed by means of a process control case study. These measurements show that an interpretive system using this new memory management technique can have a performance which is comparable to or better than a compiler oriented system.

INDEX TERMS

Real-time operating system; virtual memory; BASIC; interpreters; protection; interactive systems; structured programming; command lanuages; system documentation.

PREFACE

STATEMENT OF ORIGINALITY

All the work reported in this thesis is the candidate's own original work except where specifically stated to the contrary.

BACKGROUND

During 1974 I was involved in three small process control projects which used a simple real-time BASIC for data acquisition and some simple control functions. The system used, called PROSIC, was an extension of the Varian computer BASIC (GOUWS, 1973). The BASIC implementation had replaced earlier applications which had been coded in assembler, enabling an order of magnitude reduction in program= ming effort to be achieved in the process. Despite this successful use, it became apparent during the course of the projects, that PROSIC (and all other real-time BASIC's available at that time) had a number of limitations. Some of these were overcome in an upgraded version, called PROSIC 2, which was produced in early 1975 (HEHER, 1975, 1976a, 1976b) but serious defects remained which limited the scope of PROSIC.

In 1975 a new medium-scale process control project was commenced (HEHER, 1977b). On examining the requirements for the project, it was clear that a simple real-time BASIC such as PROSIC would not be adequate, primarily because of the lack of multiprogramming facilities. FORTRAN IV was therefore used as an applica= tions programming language for this project, running under the control of the Hewlett Packard Real-time Executive RTE II. In the course of this project considerable experience was gained in the use of a non-interactive compiler-oriented system. The FORTRAN/RTE combination worked satisfactorily, but in various instances it was noted that programming tasks were considerably more difficult to perform in the compiler-oriented system than they would have been in an interactive system. A general purpose real-time operating system like RTE is also relatively difficult for the application oriented user to operate.

The experience gained on this project, together with the experience of using a real-time BASIC on the previous projects, indicated a definite need for an interactive multiprogramming system. The widespread acceptance of structured programming techniques over the last few years also pointed towards the in=

corporation of these concepts in an interactive program development system. Examination of the current literature indicated that this need was being felt elsewhere as well, but that there were no systems available which met all the desired requirements.

The design of a multiprogramming system, which had commenced in 1974, was therefore continued in earnest in 1975. In attempting to design the system it was soon apparent that serious memory management problems existed in the construction of a multiprogrammable system. A variety of techniques for solving the problem were considered and discarded before the concept of 'Software Virtual Memory Management' was evolved early in 1976. This new system was originally called PROSIC 3 but in 1977 the name was changed to VIPER (Virtual Interactive Process Executive for Real-time control) to reflect the totally different structure of the new system.

SCOPE AND CLAIMS

This thesis therefore presents a new method of constructing real-time interactive operating systems for a mini- or microcomputer. The primary claim of this thesis is that to construct such systems fundamental memory management problems must be solved. The concept of software virtual memory management is proposed as a solution which does not require the use of any special purpose hardware, the memory management functions being implemented entirely in software.

The additional claims of this thesis are that:

1. The interactive facilities found in simple monoprogrammed systems can be extended and improved in multitasking systems.

2. Structured programming concepts can be efficiently supported in an interactive multiprogramming environment.

3. An interpretive system can be constructed which has a performance comparable to that of a system executing in-line code with swapping, without requiring an electromechanical storage device for the time-critical tasks.

4. A simple user interface can be provided which facilitates the use of the system by application oriented users.

5. New and improved protection facilities can be provided to permit safe multi-user multi-programming by the application oriented user.

A system incorporating the facilities presented above provides a unique and powerful set of software tools which makes a marked contribution toward the goal of producing more reliable software efficiently and economically. Many of the facilities listed above are not new or original concepts and have been discussed and proposed in various contexts, as referenced in the body of the thesis. It is claimed, however, that they have not or could not be implemented on small mini- or microcomputer systems which use a high-level user oriented language for process control word.

The concepts presented are demonstrated in the experimental operating system VIPER which operates in an interpretive mode. It is claimed that the performance of interpretive systems can be significantly improved using the memory management technique, to the extent where they are competitive with conventional compiler based real-time executives, for a range of applications where interactive systems could not be previously used. The system described in the thesis was developed primarily for experimental process control work, but a further claim of this thesis is that an operating system using software virtual memory management could be extended and its performance improved to an extent where it competes with a wider class of applications.

VIPER has been used in an industrial application. From the results of this case study it is claimed that compared to the original FORTRAN implementation, the VIPER implementation required less memory and bulk storage space; was easier to write and generated more readable code; took less time to debug; could be more thoroughly tested; was far safer; and executed faster.

ORGANIZATION

Chapter 1 opens with a review of the problems facing the real-time programmer and of the techniques which have been proposed for the production of cheaper and more reliable software. The properties required of an interactive system are then discussed followed by a brief review of existing real-time interactive operating systems and languages. The chapter concludes with an explanation of the requirement for Software Virtual Memory Management (SVMM).

An overview of the operating system VIPER is presented in Chapter 2. This system has been constructed both to demonstrate the facilities which can be implemented using SVMM and to assist in their development. In Chapter 3, the memory management algorithms themselves are described in more detail together with some comments on alternative structures and the reasons for selecting particular mechanisms in the VIPER implementation.

A detailed description of all the important features supported by SVMM is given in Chapter 4 under the headings of structured programming, interaction, protection and error handling, synchronization and documentation. In Chapter 5 some figures on the performance of the system are given, both in absolute terms and in comparison with VIPER's monoprogrammed predecessor PROSIC. Information on the performance of other interpretive and interactive systems which has been reported in the literature is also presented.

The performance of the SVMM system in comparison with compiler-oriented systems executing in-line code, is made in Chapter 6 by means of a case study. This case study draws upon my experience with the FORTRAN-based process control system mentioned in the opening paragraphs of this preface. The difficulty of performing more precise performance evaluations is also noted.

The concluding chapter discusses the limitations of, and possible extensions to the SVMM system. Some interesting extensions are examined which can be used to improve the performance of the SVMM system and extend its range of application. These extensions relate both to work which is in progress, but which has not been completed, as well as to more fundamental aspects.

DOCUMENTATION OF VIPER

Within this thesis only a brief functional outline is given of the construction and operation of the operating systems VIPER. The primary documentation for this system is the source listing. The source has been written with extensive comments and cross-indexing, so that although it is written in Varian Assembler it is intended to be a readable document even for readers unfamiliar with the Varian code. No flow charts are used in the documentation of VIPER nor were any used in its design. This is in accordance with modern documentation practice.

This ...../(v)

This approach was also adopted with PROSIC and this proved to be an adequate way to diseminate information on the internal structure and operation of the system. The advantage of using the source listing as the primary descriptive document is that up-to-date copies can be easily produced for the interested worker. The excessive bulk of the listing of VIPER (approximately 500 pages), and the cost of duplication, precluded its inclusion as an appendix to this thesis, but, as noted above, copies are readily available if required.

ACKNOWLEDGEMENT

The work reported in this thesis was performed in the laboratories of the National Electrical Engineering Research Institute of the South African Council for Scientific and Industrial Research. The support given to this research project by the Director, Mr J.D.N. van Wyk, and the Head of the Automation Division, Dr G.J. Kühn, is gratefully acknowledged.

The concept of Software Virtual Memory Management on which the operating system VIPER is based emerged from a research programme which initially had quite different goals. The assistance of my supervisor Prof. H.L. Nattrass in pin=pointing the essential targets of the research programme together with astute and helpful criticism of the proposals which were put forward at various stages is noted with thanks.

Many people assisted with the hardware and software required to develop and demonstrate the operating system VIPER. I would, in particular, like to thank the following for their contributions:

- Mr B.D. Ravno and Mr T.E. Burne of Huletts Refineries Limited for invaluable assistance over three years in maintaining and operating the software and hardware in Durban.

- Mr P.S. Hussey for designing and writing many of the FORTRAN programs used in the case study as well as for his assistance in translating and testing some of VIPER programs.

- Mr H. Fromman for enhancing the performance of the cross-assembler and converting this package to run on the CDC CYBER 174.

Mr ...../(vi)

- Mr J.P. Los and Mr P. Grift for maintaining or constructing all the hardware used on the project, including the Varian 620i, MIKROV, CAMAC and numerous peripheral units.

- Mrs N. Thomson for her infinite patience in entering the 22 000 lines of assembler code in VIPER; as well as for her assistance with documenting both the FORTRAN and VIPER case study programs.

- Mr J. van Aardt and other staff of the Digital Systems Section of NIAST for their assistance in constructing the MIKROV.

Finally I wish to thank my wife Jenny for her encouragement and support over the years that it took to complete this thesis.

CONTENTS ...../(vii)

CONTENTS

APPENDICES ...../(x)

APPENDICES

FIGURES AND TABLES

The figures and tables are grouped together at the end of each chapter with which they are primarily concerned.

FIGURES

TABLES

CHAPTER 1

INTRODUCTION

1.1 THE SOFTWARE PROBLEM

The cost of software has been rising rapidly over the past decade and in nearly all applications the software cost now exceeds that of the hardware. Within the next decade it is estimated that the disparity between hardware and software cost will continue to grow to a ratio of 90% for software and 10% for hardware. Two factors contribute to this disparity: the first is the steadily declining cost of the hardware and the second the increasing sophistication which is expected of software. To permit low cost computer hardware to be exploited in new applications there is a pressing need for the software cost to be reduced in every possible way.

There are four components to the total cost of a software project (SMEDEMA, 1977):

1. specification and design;

2. coding;

3. commissioning (testing and debugging);

4. maintenance and upgrades.

To reduce the cost of software, attention must be given to all aspects, but particular attention must be paid to commissioning as this 'can often be the most tiresome, expensive and unpredictable phase of program development' (HOARE, 1975a). Hoare has further noted three principles which are of importance in the production of reliable software:

> 'If a programming language is regarded as a tool to aid the programmer, it should give him the greatest assistance in the most difficult aspects of his art, namely program design, documentation and debugging.

1. Design. The first, and very difficult, aspect of design is deciding what the program is to do, and formulating this as a clear, precise and acceptable specification. Often just as difficult is deciding how to do it: how to divide a complex task into simpler subtasks ..... A good programming language should give assistance in expressing not only how a program is to run, but what it is intended to accomplish ...

2. Documentation ..... must be regarded as an integral part of the process of design and coding. A good programming language will encourage and assist the programmer to write clear self-documentary code ..... The readability of programs is immeasurably more important than their writability.

3. Debugging ..... even the best-designed and best-documented programs will contain errors and inadequacies, which the computer itself can help to eliminate .....

A necessary condition for the achievement of any of these ob= jectives is the utmost simplicity in the design of the language' (HOARE, 1975a)

It is also recognized (KERNIGHAN, 1977; HOARE, 1975a) that real programs are subject to a steady flow of changes and improvements and that both the language and the operating system should make provision for this dynamic characteristic of software. Maintenance and upgrades together with testing and debugging can constitute 50 to 80% of the cost of a software project and a system which makes specific provision for these tasks can have a significant impact on the total cost of the software.

Although many of the concepts presented in this thesis are of general applicability, the thesis is concerned primarily with soft= ware for real-time applications and for process control systems in particular. KOPETZ (1976) has made some pertinent comments on this class of applications.

'The user group concerned is that of process control and, in particular, the direct control of heavy industrial plant by computer. Many types of industries are involved, such as chemical, petroleum, steel and public utilities (e.g. water, gas and sewage).

A number of user requirements combine to place major constraints on the design of a suitable system. Some of the most significant points are indicated below, though not all of these are applicable to each user:

1. The programming expertise available to a user varies from virtually none to an extensive and expert team.

2. Frequently, the process being controlled, or the control techniques being applied, are secret. In such cases, the user will normally prefer to utilise his own resources to program the most confidential areas.

3. Often, it is not practical to fully define all the functions of the system prior to installation. It is, therefore, necessary for the user to enhance his system as experience and resources permit.

4. It is normal for the system to have to function for 24 hours a day and five or seven days each week. Further, any development work must utilise the process control computer.

5. Because of reliability and maintenance problems, the system must not be dependent upon mechanical devices such as discs and magnetic tapes. These devices are often used, but only for non-critical functions.

6. Man-machine interfaces represent a major proportion of the functions of the system.

7. The market is often conservative, preferring well established techniques to potentially more effective but unproven approaches. Indeed, it is only in recent years that the use of high-level languages have become widely accepted.

8. The cost of a system may vary from around £20K to greater than £300K, but each has the same basic characteristics.'

KOPETZ notes further that no suitable systems were available to meet these requirements and goes on to describe the development of a multitasking BASIC system (the system is described in more detail in section 1.4.3). More extensive survey papers (DIEHL, 1976; GERTLER, 1975; WILLIAMS, 1976) make similar comments on the characteristics of process control systems.

In addition to the points made above there are three additional, related factors which have motivated and influenced the work under= taken in this thesis.

1. Large, complex and costly plants can afford large, complex and costly computer systems, but there are a very large number of smaller plants which can benefit from automation provided it is available at reasonable cost. In other words, decreasing the cost of computer control systems will open up new areas of application rather than merely reducing the cost of present applications.

2. Many applications are in new areas which require extensive experimental work before control strategies can be evolved.

3. The users of the systems are technically well qualified and generally have a good understanding of their plants and how they would like them to perform, even if uncertain of how to attain this performance.

As a result of these factors it is claimed that there is a definite need for improved interactive computing systems which can be used by the process oriented user. The systems should be simple and safe to use and provide flexible multiprogramming facilities to permit new tasks to be written and commissioned concurrently with tasks which are performing on-line control.

In this introductory chapter some factors which can simplify and reduce the cost of writing software are discussed next, followed by an examination of the requirements for a real-time interactive multi= programming system. In the fourth section of the chapter a few existing software systems are briefly reviewed to illustrate the problems encountered in constructing interactive systems. In the fifth and final section the importance of memory management is dis= cussed and a new memory management technique is proposed which can be used to overcome a number of the difficulties reported.

1.2 Techniques for reducing the cost of software

Since the "software crisis" was first identified nearly a decade ago, (NAUR, 1968) there have been a number of developments which have improved the reliability of software and decreased the cost of production. Seven factors which are of relevance to the class of application with which this thesis is concerned are discussed below:

1.2.1 Structured Programming

Undoubtedly the most important advance in recent years has been the development of "Structured Programming" (DAHL, 1972; WILKES, 1976). The methods and discipline associated with this concept have assisted in reducing the cost of all four components listed above. The "top-down design" or "stepwize refinement" (WIRTH, 1971) used, unifies the specification, design and coding phases, while the modularity and structural integrity of segments of code have been widely reported to reduce the number of logical errors which occur, thereby simplifying the commissioning of software. Structured programs are also easier to maintain and upgrade. Although aspects of structured programming are still under development, sufficient evidence has been accummulated to indicate that the concepts should be incorporated in all future languages and operating systems.

1.2.2 Interactive operation

The testing and debugging phase can be further simplified if they are combined with the coding phase by use of an interactive software development system. The interaction is to permit software modules to be tested as, or as soon as possible after, they are written, as well as to allow iterations in the software development cycle with the rapid testing of previously developed modules as additional modules are added. Interactive testing and debugging is particularly important in real-time systems where a complex set of programs co-operate to perform a given task in response to real-time events. If a task need be stopped or taken off-line before 'test' or 'debugging' functions can be included, the commissioning task is made considerably more difficult and time consuming.

WILKES (1976) made some pertinent comments in this connection: "There has, to my mind, been too little interest in devising efficient methods for locating the errors that do get introduced. Most debugging procedures in current use are crude and depend on examination by the programmer of a static picture of his program when it has stopped. Methods of obtaining a trace of what was happening during the running of a program have been successfully used in the past and I suggest that the time has come to re-examine these methods with the object of developing them into serious tools that can be used by the software engineer.".

1.2.3 User programming

The function of software is to perform a service for some user. If the user is able to perform the programming task himself, the program is far more likely to meet his specific requirements. This need for the programming to be undertaken by those who understand the problem has been emphasised by DRIESTROWSKI, 1975; GORDON CLARK, 1975; DIEHL, 1976; ZEH, 1976 and others. To enable the application oriented user to perform the task himself, however, excellent software tools must be available so as to "improve software reliability by reducing the opportunity for error" (GRIEM, 1975). The user does not wish to, and should have no need to learn the intricacies of a real-time operating system. There are four essential requirements to enable a user to perform the real-time programming task himself:

1. The system should be simple and safe to use and should inspire confidence in the user.
2. The user's previous experience should be built upon and extended without attempting to force him to adjust to radically new concepts. Many process engineers; for example, are familiar with FORTRAN and BASIC and any new system should draw upon this experience wherever possible.

3. The system should guide the user gently and naturally into the use of new programming techniques such as structured programming and should give him every possible assistance in preparing and maintaining good documentation.

4. Good error reporting and recovery facilities should be provided and adequate protection mechanism must be implemented to protect the user against his own errors and against his errors affecting any other users.

1.2.4 Documentation

Documentation is an important aspect of any software system, as was noted in section 1.1. In an interactive experimental environ= ment, where the programming task is evolving on-line, documentation is even more important, and commensurately more difficult to maintain. The language and operating system should provide every assistance to the programme in maintaining clear, readable documentation. An important point is that documentation is related not only to the description of a particular piece of code or program module. Of equal or even greater importance is the documentation of the overall structure of the system and the relationships amongst the various code and data modules out of which a task is constructed. As these relationships can vary dynamically, it is desirable for this aspect of documentation to be automated, so that the information represents the actual state of the system rather than an assumed state as may occur with manually produced documentation.

1.2.5 Synchronization

An essential requirement of any multiprogramming system is the provision of synchronization functions to control access to shared resources. A wide variety of techniques have been developed for synchronization (BRINCH HANSEN, 1973; DIJKSTRA, 1968; HOARE, 1974, WETTSTEIN, 1977) many of which are designed primarily for the more complex synchronization problems which occur in the construction of real-time operating systems. Only the simpler functions are needed

for ...../1.8

for the user-oriented system under consideration. Suitable functions are available and can be readily implemented, as discussed in section 4.4

1.2.6 Protection and reliability

The ideal program is one which is known with absolute certainty to be correct. This can be established for certain classes of software by using formal proofs of correctness, but as BRINCH HANSEN (1973) has pointed out "a proof is merely another formal statement of the same size as the program it refers to, and as such it is also subject to human errors. This means that some other form of program verification is still needed".

The next best thing to absolute correctness is immediate detection of errors when they occur. This can be done at compile time or at run time. (In the case of an interpreter using as incremental compiler, compile time implies any time before execution.) In either case reliance is placed in a certain amount of redundancy in programs which makes it possible to check automatically whether operations are consistent with their types of variables and whether they preserve certain relations among those variables. Error detection at compile time is possible only by restricting the language construction e.g. by using a "structured" language; error detection at run time is possible only by executing redundant statements e.g. subscript bounds on array variables. In interactive systems, which frequently use an incremental compiler, greater reliance may need to be placed on run time checks, but compile time checking should still be used wherever possible.

This still leaves a class of errors that is caught neither at compile time nor at run time. This implies that a secure and reliable system must protect both the data and physical resources of each com= putation against unintended interference by other computations.

A further class of errors are those arising from time depen= dences. These are in fact the most difficult to trace and fix as they are frequently non reproducible. The synchronization functions mentioned in the previous section are an important safeguard in this respect. Although they cannot prevent all errors, if correctly used

they ...../1.9

they can ensure that the results of each computation is independent of the speed with which the computation is carried out. In other words the result of a computation is unaffected by concurrent processes which may be running simultaneously.

All four types of verification and protection, namely compile time checking, run time checking, data and resource protection and time dependence error protection, should be implemented in a secure system.

1.2.7 High-level languages

The use of a high-level language has been more or less taken for granted in the discussion up to now as no user-oriented system should ever descend to the level of Assembler coding. High-level languages are in fact now being increasingly used even for system programming functions (SMEDEMA, 1977) and are also reportedly in= vading the small program microprocessor domain (CLAGGETT, 1977; MAPLES, 1977). While certain system programming (and microprocessor) applications will continue to be programmed in Assembler code, purely due to the lack of a suitable high-level language on a particular machine, there is a no justification for the typical process control application to do so. A high-level language should be used in all but the most exceptional circumstances, such as low-level functions with very fast response time requirements; but even these functions should be controlled from high-level routines.

1.3 PROPERTIES REQUIRED OF A REAL-TIME INTERACTIVE MULTIPROGRAMMING SYSTEM

The facilities required in interactive computing systems have been studied in some detail by a number of workers (ARDEN, 1975a; CHU, 1976; GOULD, 1975; HILDEN, 1976; PALME, 1975). Chu in particular presents a list of desirable properties of an interactive program development system:

"1. The interactive language is symbol-executable, expression-executable, and statement-executable as each symbol is being entered; the degree of interactiveness can be made under the user's command.

2. The declaration statement is permitted to be entered at any point of the source program for the user's convenience in making program entry and program composition. In order to obtain program clarity, a "declaration collector" could be included in the interpreter in much the same way that a BASIC interpreter allows the resequencing of its lines.

3. The syntax allows left-to-right, nonbacking-up, symbol-by-symbol syntax checking and execution.

4. The values of the user's data structures should be inspect=able at any point during the program execution without affecting the source program.

5. The precedence relation of the operators allow left-to-right statement execution and top-to-bottom program execution.

6. There should be a language construct which permits a "pro=grammatical pause" so that the user may examine and modify the values in his data structures.

7. There should be language constructs for program entry, program editing, program execution, program debugging, and program documentation. There should be uniformity in the syntax of these language constructs in order that the interactive language becomes easier to learn". (CHU, 1976)

Some of the properties are only directly applicable to the particular single user direct execution system described in his paper, but the concepts are extendable to more general interactive systems.

The facilities required in real-time languages and operating systems have also been examined by a number of authors (BARNES, 1975; BIANCHI, 1976; BRISTOL, 1975; ELZER, 1972; ELZER, 1977; HAASE, 1972; KOPETZ, 1976; KYLSTRA, 1977). From these papers and from the author's experience with various process control systems and applications (HEHER, 1975, 1976a, 1976b, 1977a, 1977b) a definitive list of the attributes required for a real-time inter=active multiprogramming system can be specified.

The system must:

1. support structured programming concepts with independent named procedures and subroutines, together with multi-tasking facilities;

2. provide controlled access to shared data bases (synchroni= zation and protection);

3. be simple and safe to use;

4. provide flexible interactive operations which facilitate the on-line writing, testing debugging, maintenance and documentation of real-time tasks.

1.4 REVIEW OF EXISTING INTERACTIVE OPERATING SYSTEMS

In this section a number of existing interactive systems are reviewed and their successes and shortcomings discussed.

1.4.1 Real-time BASIC and derivatives

Real-time versions of BASIC are the simplest form of interactive operating systems and they have been widely and successfully used in a variety of applications. Their primary advantage is that they permit a high-level language to be used without recourse to an expensive bulk storage device and a complex real-time operating system. The systems are simple to operate and program and have been used to a large extent directly by the users, but three fundamental restrictions have limited the more widespread use of BASIC systems.

1. BASIC is essentially a monoprogrammed system supporting one single monolithic task. No provision is made within the language nor in many implementations for multiple independent tasks.

2. The language has very poor structure which together with the limited variable naming conventions results in large programs

being ...../1.12

being unwieldy and difficult to read or change.

3. Even where multiple programs can be used using techniques such as overlays, the shared data facilities are limited and unsafe.

A further disadvantage of BASIC is the execution time penalty which results from the interpretive mode of operation. On the other hand, if a compiler is used the interactive facilities are sacrificed to a greater or lesser extent.

1.4.2 Compiler-oriented disc-based real-time operating systems

In more complex applications where BASIC cannot be used, the next "step-up" in computing power is to use a real-time operating system which supports an on-line compiler for a high-level language. Owing to the size of the compilers and the associated loader, editor and library, these systems must be disc-based and generally use some form of foreground/background memory partition with swapping of programs to and from disc storage. An example of such a system is the Hewlett-Packard RTE-II operating systems which supports FORTRAN, ALGOL and BASIC. (This system is described in more detail in Chapter 6 where it appears in a Case Study.)

These executives which support on-line compilation are frequently called interactive in that a program can be edited com= piled and link-loaded in a few minutes without disturbing other tasks in the system. This type of interaction is considerably different conceptually from that offered by BASIC however, and requires a far greater level of experience and training to utilize effectively. Some other disadvantages of these systems are mentioned below. Be= fore listing these, it should be noted that these operating systems are generally very successful in their intended function and represent a major advance in the state of minicomputer real-time software. They are powerful 'general purpose' systems which will continue to be used for a variety of applications which require the speed and generality of multi-language systems.

The ...../1.13

The disadvantages of using this type of executive for inter= active process control software development are as follows:

1. The complexity of the systems makes them difficult to operate and easy to 'crash' (some commercial systems are known to be particularly unstable and susceptible to operator error).

2. True interactive program development is not possible and real-time programs can be extremely difficult to debug because of the difficulty of providing suitable high-level debugging facilities. The only facilities available are generally memory dumps and limited utilities for monitoring the operating of programs at the assembler code level.

3. Error handling and reporting is rudimentary and is usually in machine level terms e.g. memory protect at location xxx.

4. Tasks and data areas are afforded little protection and can be turned on or off or overwritten by other users whether authorized or not.

The primary purpose of these real-time executives is in fact to provide the support necessary for writing more special purpose user-oriented software rather than for users to use the system directly. The software system VIPER described in this thesis, could for example, be developed, and run, under the control of a real-time executive as well as in a stand alone mode. To this extent user-oriented interactive software systems like VIPER and general purpose real-time executives may be considered complementary rather than competitive.

1.4.3 Multi-user and multiprogrammable BASICs

In recognition of the gap that exists between compiler-oriented real-time executives and simple real-time BASIC, various attempts have been made to extend the facilities of BASIC into a multi= programmed mode. As it is difficult to generalize about these systems, four particular systems will be briefly reviewed. The first two retain the interpretive mode of operation while the second two use a combined compiled/interpreted mode.

1. HP real-time multi-user BASIC (HEWLETT-PACKARD, 1976)

   This system runs under the HP RTE II or III executive and supports up to four users each of whom has his own copy of the entire BASIC subsystem. If sufficient memory is avail= able, a user may be memory resident, but in typical instal= lations the users will share a memory portion with other tasks. In this situation the entire BASIC program and the BASIC subsystem are swapped to and from the disc with an overhead of 100 to 250 ms per swap. The users have limited shared data facilities and each user can only have one main program which is partitioned into subtasks by line numbers. All tasks have a global (common) symbol table. A flexible subroutine calling mechanism is provided, but subroutines can only be coded in ASSEMBLER or FORTRAN. (The BASIC GOSUB function is not a subroutine call in the accepted definition of a subroutine). In summary, although the system has a limited multi-user capability, it is not a multiprogrammable system.

2. NOVA Multex-BASIC (PERSEUS, 1976)

   This system uses a single re-entrant copy of an interpreter to execute a set of independent tasks which are located in user specified memory partitions. A maximum of 32 tasks are per= mitted each of which is a single monolithic BASIC program. Only ASSEMBLER subroutines can be called from BASIC programs. The size of a memory partition can be changed with user commands only, the system performing no memory management outside of a memory partition. A single global common area, which is not protected in any way, is used for all tasks. A notable feature of the system is the ability to provide some degree of protection by prohibiting a partition from using specified commands. A major disadvantage is the necessity to have a physical I/O device connected permanently to a partition if that partition

performs ...../1.15

performs any input or output. Only the system console can be switched from one partition to another with operator commands.

3. KENT K90 BASIC (KENT, 1974; KOPETZ, 1976)

This BASIC system operates in two disjoint modes. The one is a development mode where normal BASIC type interactive operations are permitted and the other is a multiprogrammed mode. Only compiled programs can exist in the multiprogrammed mode and no interactive operations are permitted on these procedures. The development mode is similar to a time sharing BASIC in that up to three terminals can be active simultaneously, but no communication is possible between a user at a terminal and any other task in the system. Access to the plant data-base is also restricted in the development mode in that no output operations are permitted.

In the multiprogrammed mode programs are compiled either into resident memory areas or into user specified partitions. Programs resident in one partition are swapped to and from bulk storage devices as required by the scheduler. Only a limited number of resident programs can be added or deleted without performing a system regeneration. Hardware memory mapping devides are used to provide the necessary access to partitions. (The system operates only on PDP-11 computers.) No memory resident shared data facilities are provided and task to task communication beyond a single word must be per= formed via shared files which are resident on a bulk storage device.

A notable feature of the K90 system is the comprehensive treatment of error handling. A variety of modes are possible ranging from full system control and reporting of errors to full user control and reporting. A major drawback of the system however is the complete separation of the program development and multi-programming modes, each of which uses its own set of keyboard commands and program directives. This lack of uniformity of presentation is a serious handicap to process-oriented users.

4. SWEPSPEED (WILKINS, 1976)

SWEPSPEED is a multiprogrammed BASIC system similar in many respects to the KENT K90 BASIC. All procedures must be compiled before execution but a limited set of interactive facilities are available for use on executing programs. (The symbol table is retained in the compiled version permitting symbolic examination of variable values when in a special mode.) The various procedures within the multiprogrammed system are identified by number only and no named subroutines are permitted either.

It is a single-user system with only one console being supported where program development can be performed. All commands to the command job which controls the system, all editing and listing and all error messages are transmitted through this terminal.A single global data area is provided for access to shared data. A certain degree of protection is provided for this data area in that programs below a certain priority can only read and not modify global variables, while other priority levels can read and write to globals, but cannot create them. This restriction is necessary because globals can only be deleted with difficulty once created, requiring either a system generation or a temporary shut down of the system to enable the 'system manager' to clean up memory. Deleting statements and certain other operations also result in wasted memory which can only be recovered with difficulty.

A notable feature of the system is the ability to back-list (decompile) a program from its compiled code. (This is another reason for retaining the symbol table.) The advantages of only a single copy of a program without the need for a separate copy of the source are therefore retained together with the advantages of high-speed execution.

1.5 ...../1.17

1.5 SOFTWARE VIRTUAL MEMORY MANAGEMENT

Comparing the requirements stipulated in section 1.3 with the pro= perties of the systems described in section 1.4, it can be seen that no existing systems are satisfactory in all aspects. Their major shortcomings are:

1. The lack of independent named procedures and subroutines which is essential for a structured programming approach.
2. The poor shared data facilities and a lack of protection for any facilities that are provided.
3. Restricted interactive facilities, in that none of the systems listed, nor any system known to the author, permit the interactive operations to be used on executing tasks.

These shortcomings can all be traced to a single problem: memory management. The implementation of interactive facilities requires that the code defining a task and its associated data areas, be expanded and contracted as the interaction proceeds. In a multi= programmed system the difficulty occurs in attempting to allow multiple tasks or procedures to simultaneously undergo this dynamic change in size and structure. The addition of a multi-user capability further complicates the memory management task, as does the requirement for flexible access to shared data areas.

Hardware virtual memory mapping devices were considered as a possible solution to this memory management problem, but were rejected because of the desire to maintain processor independence. Suitable mapping systems are in any event only available on medium to large scale machines, whereas the system described in this thesis is designed for use on mini- or microcomputer systems. A memory management technique was required which would permit the operating system to be as transportable as BASIC.

These considerations led to the development of a new memory management technique. This management system is implemented entirely in software, but has many of the characteristics of a system using

hardware ...../1.18

hardware virtual memory management. It is for this reason that the technique used has been called 'Software Virtual Memory Management' (SVMM).

The term 'virtual memory' has two connotations in the context of this thesis: the first is related to the usual concept of addressing a logical space which is larger than the physical space; the second is related to the security of, and access to, both tasks and data structures which are operated upon as if they were located in a file system. Both executable (and executing) tasks and data structures are afforded protection in a hierarchy of security levels. The user therefore creates, modifies and executes tasks as if he were working on a set of files which may in fact be memory-resident; and conversely, he operates within a task as if all tasks and data structures were memory-resident, when in fact they may be resident on some external device. This file-system analogy is an extension of the usual concept of virtual memory in that it is associated with the reverse mapping of memory onto a mass-storage device, as opposed to the mapping of mass-storage onto memory, which is the property of the extended logical space. The importance of this reciprocity is that the properties of the memory management system can be utilized to construct an operating system with the attributes required of a real-time interactive multiprogramming system.

CHAPTER 2

AN OVERVIEW OF VIPER

In this chapter the operation of VIPER (Virtual Interactive Process Executive for Real-time control) is briefly described to provide a background for the detailed discussion of the construction of SVMM and other facilities in chapters 3 and 4. The overview deals with seven topics:

1. Interpretive operation.
2. Multiprogramming.
3. Interactive operations.
4. Protection.
5. Shared data.
6. Bulk storage devices.
7. Limitations.

VIPER was constructed both to demonstrate the facilities which can be im= plemented using Software Virtual Memory Management (SVMM) and to assist in their development. The level of development was such as to enable VIPER to be used in an industrial application to permit a realistic assessment of its performance to be made, as discussed in chapters 5 and 6. Some of the specific limitations and omissions that resulted from this approach are listed in section 7 of this chapter, while some of the more fundamental limitations of Software Virtual Memory Management (SVMM) are discussed in chapter 7.

VIPER is an interpretive system which evolved from an earlier monoprogrammed real-time BASIC called PROSIC (HEHER, 1975, 1976a, 1976b). PROSIC in turn was a development from the original VARIAN BASIC (GOUWS, 1973). VIPER is coded in VARIAN Assembler and like BASIC is a stand-alone system containing all its own operating system functions. Further information on the hardware systems and software techniques used in the development of VIPER are given in Appendix A3.

2.1 INTERPRETIVE OPERATION

The interpretive mode of operation of the original BASIC has not been changed significantly in VIPER. The language processing modules and the operating system functions are all resident in memory, and it is only the remaining memory which is manipulated in the SVMM system. Figure 2.1 shows this basic division of memory as well as the approximate size of the partitions.

The basic mode of operation of the system is shown in Figure 2.2. Between the interpretive execution of each statement a single flag is tested to determine whether any system work is pending. The various categories of work which may need to be performed are listed in Figure 2.3. This procedures ensures that no asynchronous events are handled during the interpretive phase and the evaluator is therfore not re-entrant. (This would in any case have been difficult to achieve on the VARIAN 620i.) The response time to asynchronous events is therefore limited by the time to execute a statement interpretively, which may be as much as 10 to 20 ms. This was acceptable for the range of work envisaged for VIPER.

In the evaluator section of the interpreter shown in Figures 2.4 and 2.5, two modes of operation are possible, depending on whether the internal meta-codes are stored in infix or Polish forms. Examples of these two types of internal representations are given in Figures 2.8. The infix form was enherited from the original BASIC. In this form, precedence is only determined as a statement is executed, requiring an operator stack as well as an operand stack. The Polish mode of operation is mentioned here even although it has been only briefly tested, as this is the way in which the inter= preter should be operating. This aspect is commented on in more detail in sections 5.1 and 7.2.

A program in VIPER consists of a three-part module, as shown in figure 2.6. The symbol table consists of a list of descriptors containing both the ASCII characters of all identifiers and their values. The ASCII representation is required for the backlisting (decompilation) and interactive operations. The structure of the descriptors on this table is closely related to the memory management functions and this aspect is therefore described in chapter 3 and Figures 3.2, 3.3 and 3.4 illustrate the descriptors used in VIPER. The statement pool consists of elements as shown in Figure 2.7, while the structure of individual code words is shown in Figure 2.8. The major difference between VIPER and its forerunners is that all operand references (variable addresses) are values relative to the start of the symbol table. An absolute address is therefore computed from the relative operand address and the current position of a segment.

As a result of using these relative pointers, the address field is comparatively small and can be packed into one 16 bit word together with an operator code. Used together with the Polish form, this structure results in a compact representation, as shown in an example in Figure 2.8. HELPS (1974) and BROWN (1977) have commented on the advantages of this compaction property of interpretive systems which can be used to achieve significant savings in memory space.

2.2 MULTIPROGRAMMING

VIPER permits independent, named segments of code and data to be executed and manipulated concurrently. Each of the code segments is a self-contained procedure as shown in Figure 2.6, which is similar in many respects to a stand-alone BASIC program. The data segments are used for shared data as well as for input/output buffering and other system activities. These segments all exist in an area of memory which is reserved for SVMM operations, the remainder of the memory being used for the fixed, resident operating system nucleus. The resident code is VARIAN machine code, while the code segments which are manipulated in the SVMM space can consist only of the special high level language meta-codes which are executed which are executed interpretively. Figure 2.1 shows this basic division of memory as well as the approximate size of the partitions.

The procedures (= code segments) are created and manipulated interactively from an input device. More than one keyboard can be active at once as VIPER has a multi-user, multi-terminal capability, as well as multiprogramming facilities. Other tasks in the system can also run concurrently while program development is proceeding. At any given time an input device is associated with a particular procedure and all commands and statements are executed within the scope of that procedure. The association of a device and procedure can be changed with simple commands.

Table 2.1 illustrates some of these interactive operations, while a complete description of all commands and their syntax is given in Appendices A1 and A2.

All statements have the same syntax, irrespective of whether they are executed as commands or as program statements. In other words the command and programming languages are synonymous. This duality not only simplifies the user interface but also results in the protec= tion and data manipulation facilities being applied equally to the command and programming languages. Statements are differentiated from commands by the presence or absence of a line number.

As each line is entered it is incrementally compiled into the internal meta-code format. If it is a command it is executed immediately,whereas if it is a statement it is stored in the appropriate position in the segment. As the line of code is being entered, the segment with which it is going to be associated may be memory resident or it may have been swapped out onto a bulk storage device. In the latter case, the segment will be swapped back into memory under control of SVMM for the compilation and storing operations. Immediately after compilation the segment may be swapped out again if the space is required for other tasks, or it may remain resident. When the segment is swapped back in, it can be positioned at any location in memory where there is space i.e. it does not have to return to the same location. If there is sufficient memory available, all segments may be memory resident all the time even with two or more users working simultaneously. In addition to being swapped, segments can also be dynamically relocated (moved) in memory to make

space ...../2.5

space for additions to a segment or to make space for a new segment. The movement of segments to and from a bulk storage device is in= visible to the user and results in perceptible delays in keybord response only when the segment size approaches the size of availablable memory.

2.3 INTERACTIVE OPERATIONS

One of the most important properties of VIPER and SVMM is that inter= active operations, including the execution of commands and the addition of statements, can continue while a procedure is executing. Operations of this type were illustrated in Table 2.1. This facility is an invaluable aid in the debugging of process control software, where a number of tasks are executing concurrently. Typical tasks of this type execute cyclically, obtaining data from a plant data base, calculating a control algorithm and then outputting a command to an actuator. As the control algorithm is invariably time dependent, stopping the task from executing in order to examine the values of a variable (as would be necessary with all but one real-time BASIC which is known to the author) destroys the time dependent characteristics of the data. A FORTRAN-based system is in an even worse position as the task must not only be stopped but edited, compiled, link-loaded and executed afresh before the required data can be monitored (assuming that this can be done). Besides being extremely cumbersome, by the time this re-loading is complete, the condition which it was desired to monitor will quite likely have been destroyed, requiring that the task be re-edited, compiled and line-loaded once more to re= move the write statements ... ! (or suffer voluminous printout for the next few hours while waiting for the event to repeat itself). The alternative to the above procedure is to place all the variables of interest in a common area and monitor their value from another program. The difficulty with the approach is that the allocation of common areas must be carefully performed when the control programs are first planned and usually cannot be expanded at will. By the very nature of program bugs and typical real-processes, it is also very difficult to foresee all the possible states in which a task may execute and hence equally difficult to decide which task variables must be allocated to common areas.

These ...../2.6

These problems are compounded by the fact that control algo= rithms frequently have special coding to cater for transient or unusual plant conditions which may occur relatively infrequently. Off-line testing and simulation can be used for testing these conditions in some cases (and should be used wherever possible) but on-line real-time testing is still an essential requirement in most process control systems.

The provision of interactive debugging operations on executing real-time tasks is therefore not merely a convenient feature, but a powerful tool for the testing and debugging of real-time software. As noted in section 1.1, this commissioning phase can be "the most tiresome, expensive and unpredictable phase" (HOARE, 1975) and any tool which can simplify and shorten this phase can make an important contribution towards the goal of producing more economical and reliable software.

2.4 PROTECTION

The basic philosophy underlying the protection functions in VIPER was to extend the concept of protection to executing tasks and their associated data structures. Protection facilities are provided in most operating systems but usually only to bulk-storage (disc) resident files. Executing tasks and shared data elements are frequently afforded no protection whatsoever.

A specific goal of VIPER was therefore to provide file-system-like protection measures (and additional facilities) to executing tasks as well as to the shared data structures. It should be possible for a user to grant a range of access rights ranging from virtually unrestricted access to completely restricted access to all accept holders of the appropriate password. Reasonable protection facilities should be (and are) applied at all times without specific user action but a user can be expected to expend a modicum of effort to obtain the highest degree of security.

The actual protection facilities implemented in VIPER and additional facilities which could be implemented if required, are described in section 4.3.

2.5 SHARED DATA

Shared data areas are an important resource in a real-time environ= ment. They are used to pass information on the process state from one procedure to another and hence require protection from inadver= tent or illegal modification if the system is to be secure. Simple read or "read/write" access attributes, together with password protection on who may change the access state, are adequate in many instances. Additional facilities are required for synchronization purposes however, and to this end a semaphore has been included as an integral part of the data structures used in VIPER. This can be used either directly with independent LOCK-FREE commands or in as a structured-pair in the form REGION-ENDREGION.

Shared data segments in VIPER are referenced and defined in a manner analogous to that of named COMMON in FORTRAN IV, with the significant difference that the segments can be created and deleted dynamically like files, protected like files and moved to and from input/output devices. Table 2.2 illustrates some of the commands and statements available for manipulating these shared data elements. A more complete description is given in Appendices A1 and A2 as well as in sections 4.3 and 4.4.

2.6 BULK STORAGE DEVICES

VIPER was originally developed with the intention of operating it primarily in a memory resident mode, with only infrequent access to bulk storage devices being required. If a computer with 32 K words of memory is used the assumption is valid for a wide class of applications. Due to hardware delivery problems, however, only a 16 K machine was available for nearly all development work on VIPER, including the entering and initial debugging of all the 25 programs written for the Case Study. Working in this restricted space where only one or two of the programs could fit into memory at once, forced more attention to be paid to the use of bulk storage devices at higher swapping rates.

Table 2.3 lists the devices which have been used in VIPER and their characteristics. A typical configuration consists of the

use ...../2.8

use of a cassette unit for program storage and transportation together with either the cartridge disc or CAMAC Bulk Memory for the temporary storage of programs which are swapped out. (The cassette unit was used for program storage as there was a unit available for use on each of the two computers used in testing VIPER, whereas there was only one disc unit. The bulk memory is volatile and there= fore cannot be used for storage.)

The management of these bulk storage devices is described in section 3.3.

2.7 LIMITATIONS

In its present from VIPER is an experimental operating system con= structed to develop and demonstrate the concepts discussed in this thesis. Due to the lack of suitable hardware and software tools which would have permitted a more sophisticated implementation, the development of VIPER has been halted at a point where it is adequate to perform the operations required for the case study described in chapter 6. Certain limitations and omissions are mentioned in the text where applicable while some of the more fundamental ones are listed below.

1. VIPER is coded in VARIAN Assembler code as no high-level language was available on the VARIAN computers which were used. As the source listing comprises 22 000 lines (code and comments) the system has become too large to be easily maintained and developed. This problem is aggravated by the lack of an underlying operating system. A system like VIPER should be written around a compact operating system kernel with a high level language being used to write the outer shells of the overall system.

2. The I/O structure of VIPER is ad hoc and all drivers are hard-coded into the total system. Input is interrupt-driven under software control but output operations have been left unbuffered and are sense-loop driven.

3. ...../2.9

3. Overlapped execution with swapping is not implemented. The CAMAC bulk memory module is accessed under program control due to the lack of suitable hardware for DMA operations. The cassette units are also not set up for DMA operations and in any event they are not suitable for use as swapping devices. The cartridge disc is driven via DMA and overlapped execution and swapping is theoretically possible when using this device, but as the unit used was essentially on loan, the simplest driver was used which would merely enable the system to run using a disc. (The same block transfer oriented driver is in fact used for both the disc and the CAMAC bulk memory unit except for the final block read and write routines.)

4. Executive-controlled swapping of data segments has not been implemented.

5. Not all protection modes and checks were incorporated to control access to shared data segments. Segments can be individually read and write protected, but can also be accessed by other than the password holder. Procedures are fully protected, however. The facilities which have been implemented are considered adequate to demonstrate the concepts presented.

6. The interpretive meta codes are stored in infix form as in the original BASIC rather than in the Polish form which is recommended. This latter format would have a marked effect on the performance of the system as the Polish code form takes less space and executes faster. This ommission does, however, enable a direct comparison to be made between the monoprogrammed PROSIC and the multiprogrammed VIPER. Some measurements have also been made to illustrate the difference between the two representations.

A research program is underway which is aimed at producing an improved version of VIPER which will overcome or eliminate many of these limitations. The specific steps which have been taken or are proposed are outlined in chapter 7.

TABLE ...../2.10

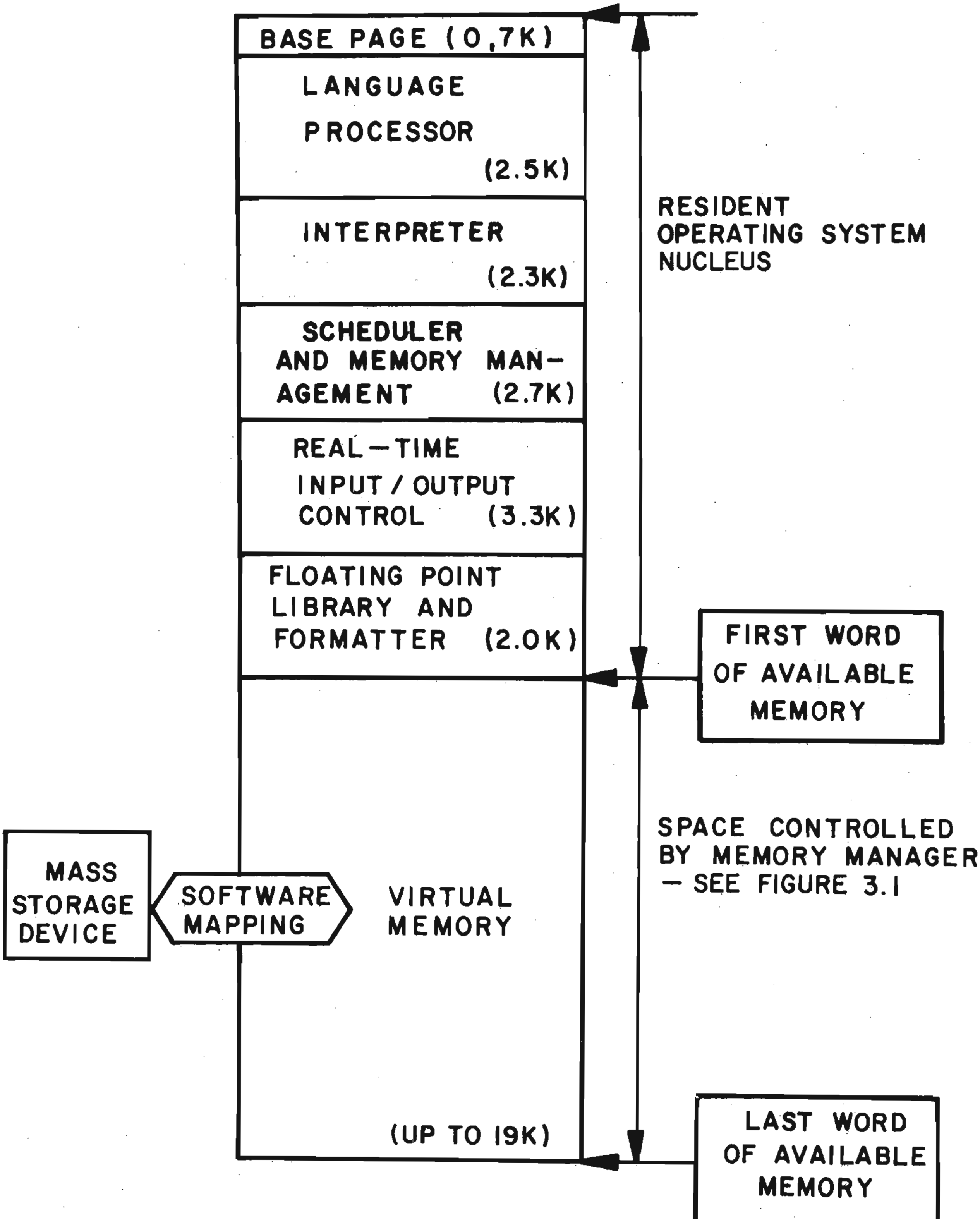
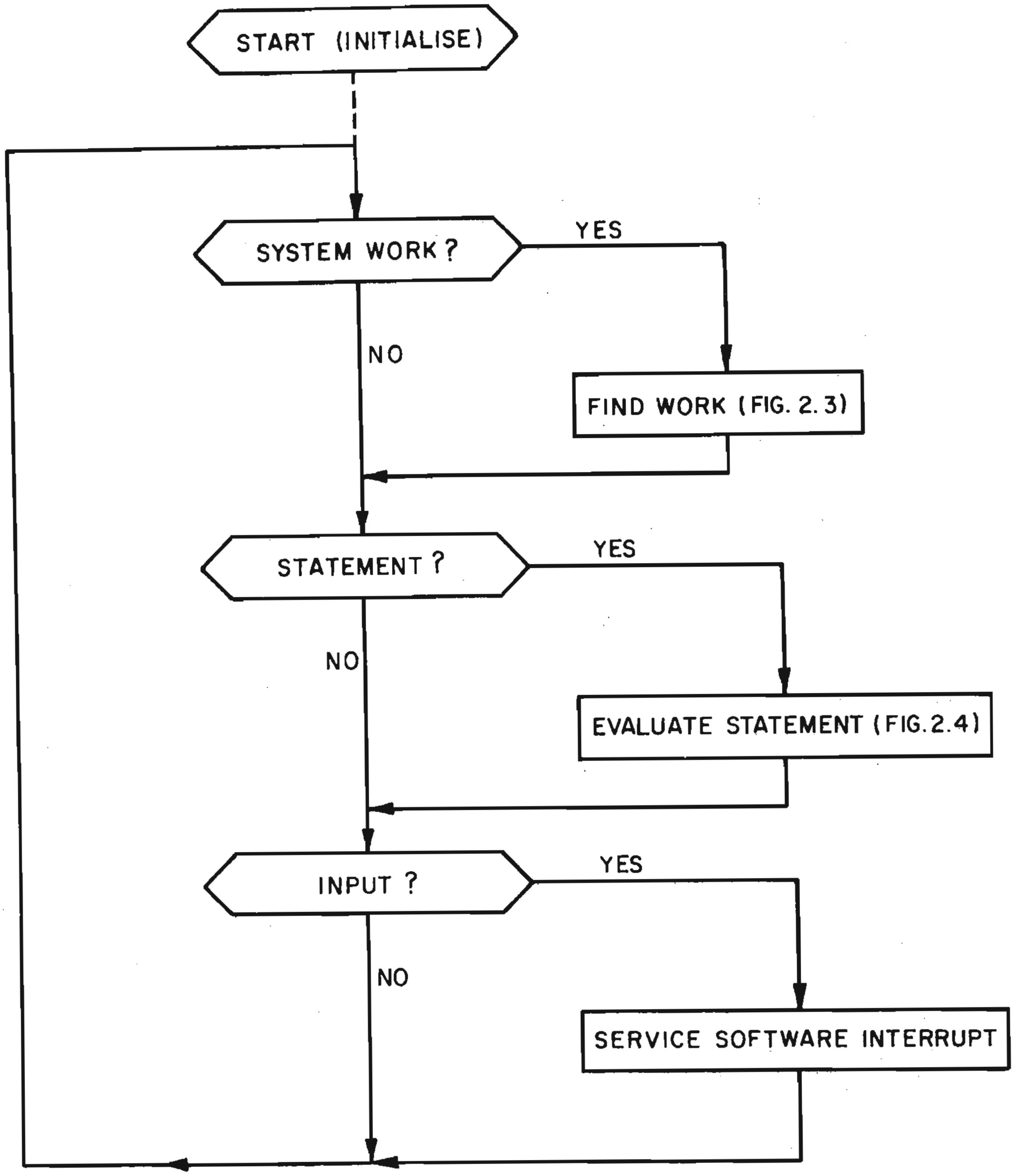


FIGURE 2.1 VIPER MEMORY MAP

(a) FLOW CHART



(b) ASSEMBLER CODE

```
MAIN    LDA WORK        WORK FLAG
        JAPM  FWORK     FIND WORK (SEE FIGURE 2.3)
        LDA  CNXP       CURRENT NEXT STATEMENT POINTER
        JAPM  EVAL      EVALUATE STATEMENT  (SETS NEXT CNXP)
        CALL TESTI      TEST FOR INPUT (SOFTWARE INTERRUPT)
        JMP MAIN        LOOP
```

FIGURE 2.2 INTERPRETIVE CONTROL

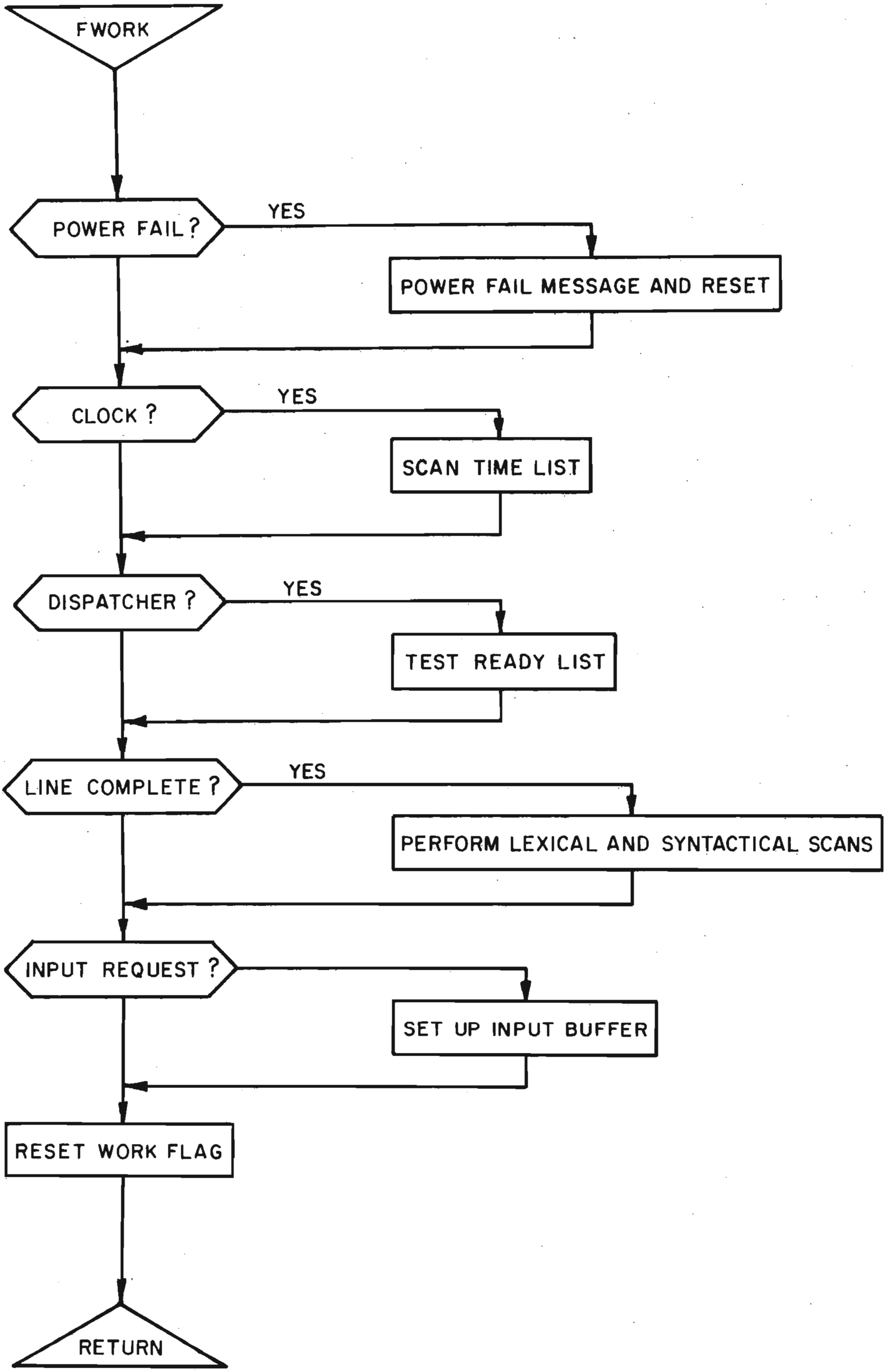


FIGURE 2.3 FIND SYSTEM WORK

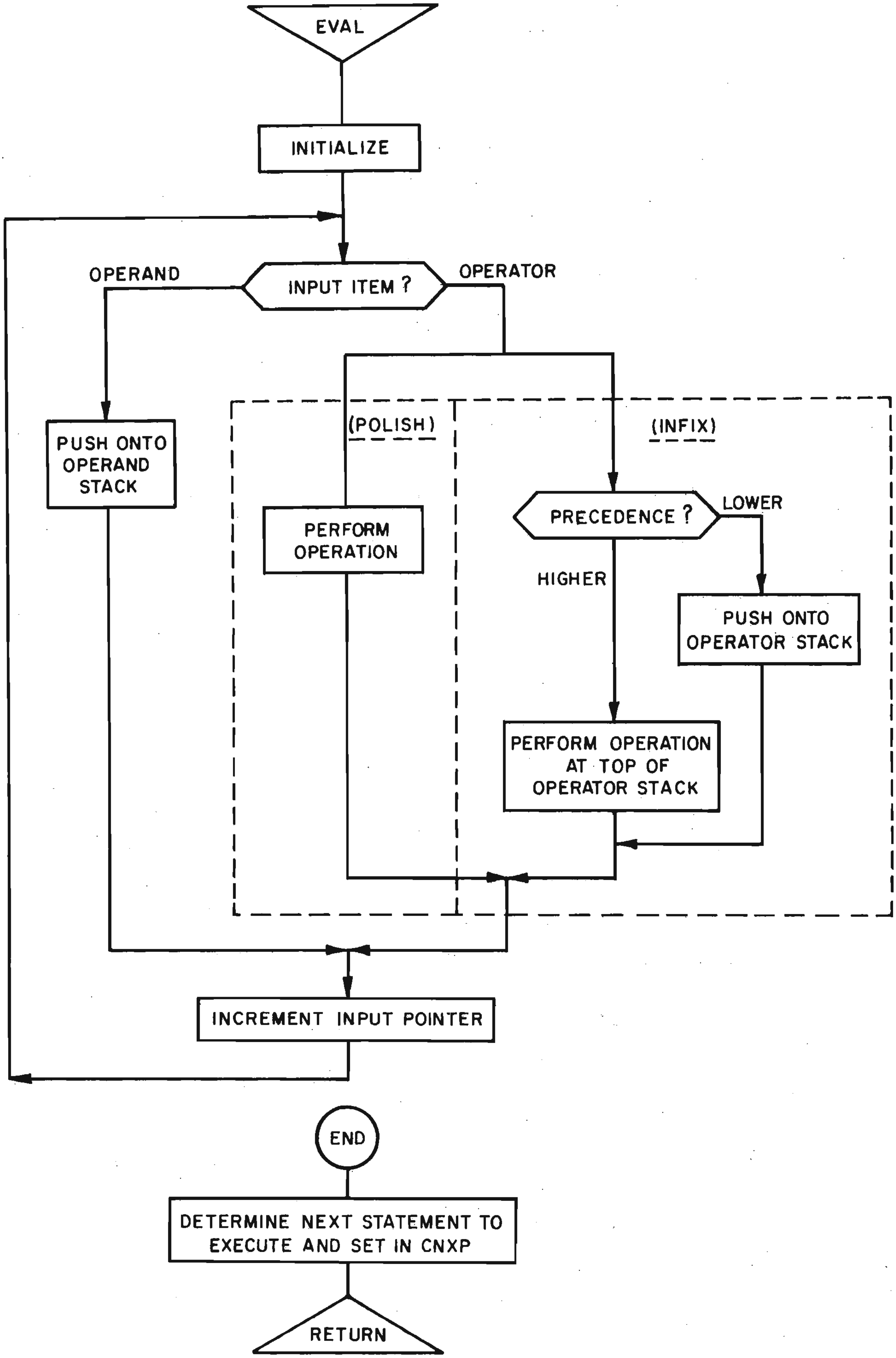


FIGURE 2.4 EVALUATOR

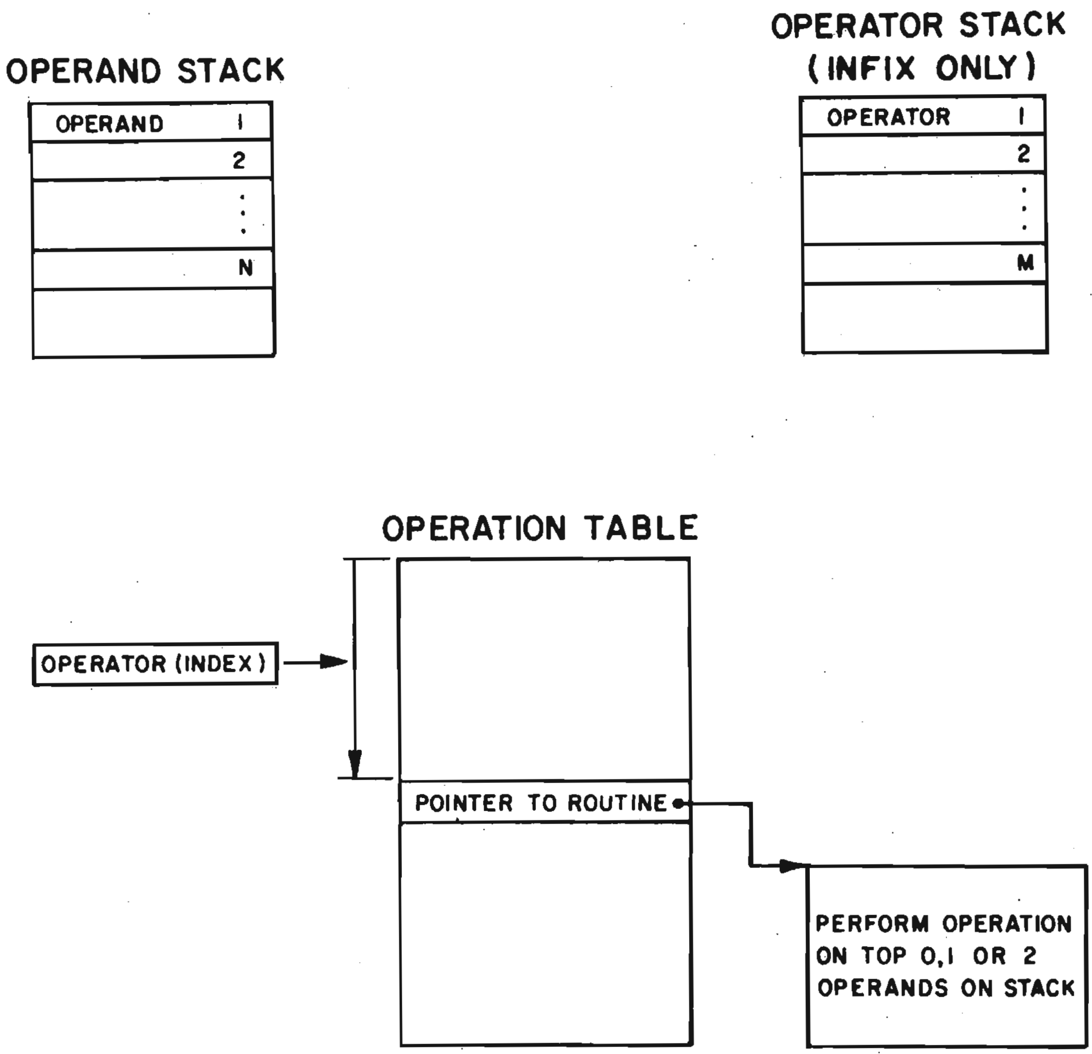


FIGURE 2.5 PERFORM OPERATION

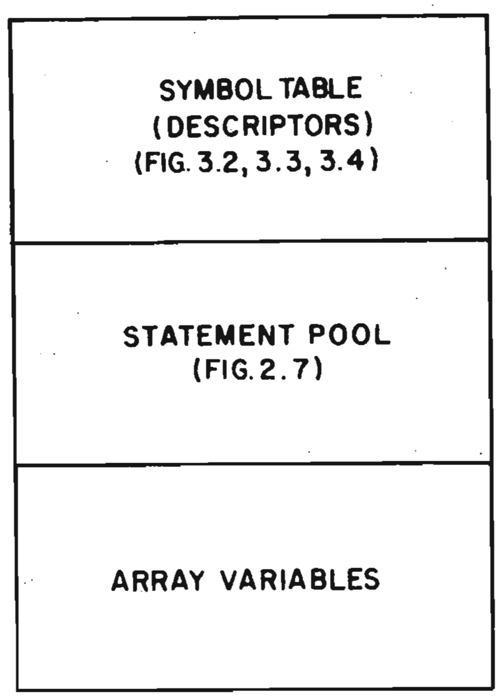


FIGURE 2.6 PROGRAM STRUCTURE

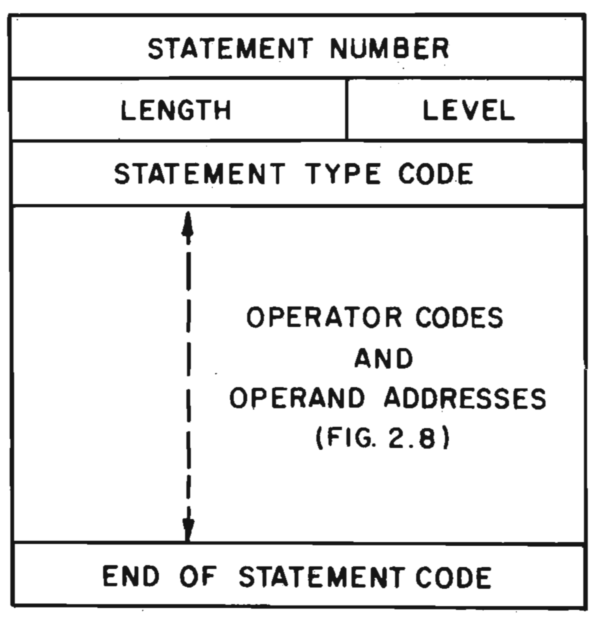


FIGURE 2.7 STATEMENT POOL ELEMENT STRUCTURE

15 8	7 0
− (CODE	CODE TYPE)

− ( ) = 1's COMPLIMENT

CODE TYPE USED TO DETERMINE PRECEDENCE, NEGATIVE (COMPLIMENTED) VALUE DISTINGUISHES CODE FROM ADDRESS

<u>EXAMPLE :</u> LET A = B + C

(a) INTERNAL FORM USED IN VARIAN BASIC AND VIPER

−(27	11)	LET
ADDRESS A		A (LOCATION IN SYMBOL TABLE)
−(67	8)	=
ADDRESS B		B
− (55	3)	+
ADDRESS C		C
− (0	15)	END OF STATEMENT

(b) SUGGESTED POLISH FORM

15 9	8 0
0	(B)
+	(C)
=	(A)

(B) = ADDRESS OF B

NOTE : ALL ADDRESSES ARE RELATIVE TO SYMBOL TABLE START

FIGURE 2.8 INTERNAL META-CODE FORMAT

TABLE 2.1

A SHORT EXAMPLE ILLUSTRATING SOME INTERACTIVE OPERATIONS

INPUT (Output not shown)	INPUT DEVICE ASSOCIATION	COMMENT
LOGON USER1	MASTER	USER1 = password (echo of input is suppressed during LOGON) Creates a procedure called USER1.
PROC ABC	USER1	Create a procedure called ABC and associate input device with it. ABC has default password USER1.
10 ...	ABC	Enter statement into ABC (in any order)
20 ...	:	
:		
PROC XYZ	ABC	Create XYZ (Input now associated with XYZ)
100	XYZ	Enter statements
50 ...	:	Enter statements
:		
CHANGE ABC	XYZ	Return to make a change to ABC (only permitted to password holder USER1)
200 ...	ABC	Change a statement in ABC
RUN XYZ EVERY 5 SECS	ABC	Set XYZ to execute periodically
RUN (ABC)	ABC	Execute ABC-(ABC) optional (defaulted) because of input device association
PRINT X	ABC	Examine variable X in ABC while ABC is running
MONITOR XYZ	XYZ	Monitor operation of XYZ (restricted rights)
PRINT Y	XYZ	Examine variable Y in XYZ while XYZ is running
DEBUG ABC	XYZ	Enter restricted mode (no changes to existing statements permitted)
100 PRINT X	ABC	Insert statement to examine X at line 100 (ABC still executing)
CHANGE (ABC)	ABC	Move to CHANGE mode to permit alterations.
110 ...	ABC	Make a change.
PRINT X	ABC	Examine X now
STOP (ABC)	ABC	Terminate execution immediately.
TURNOFF XYZ	ABC	Remove XYZ from time list.
SAVE	ABC	Save copy of ABC on external device.
SAVE XYZ	ABC	Save XYZ
LOGOFF	MASTER	End of session, return to Master Deletes procedure USER1.

TABLE 2.2 ...../2.18

TABLE 2.2

SOME EXAMPLES OF SHARED DATA MANIPULATION

CONSOLE INPUT	COMMENT
LOGON USERI	Password USERI will be associated with all commons created.
COMMON SIZES, N1, N2	Construct a data area (this is a command).
ACCESS (SIZES) = WRITEA	Permit write operation.
N1 = 100; N2 = 120	Initialise this COMMON.
PROC XYZ	Create procedure XYZ
10 COMMON SIZES, N1, N2	Link to SIZES to pick up N1 and N2 Default access is read only.
20 COMMON COM1, A(N1), B(N2)	Set up variable size data area.
30 COMMON COM2	No data area, semaphore only.
40 ACCESS (A) = READA+ WRITEA; ACCESS (B) = 0	A: read and write; B: not used here (no access)
⋮	
100 REGION COM1	Start of a critical region (Mutually exclusive access to COM1)
⋮	
160 A(...) = ...	Perform some operation on A
⋮	
180 SAVE COM1	Save current values on bulk storage device.
200 ENDREGION COM1	End of critical region.
210 FREE COM2	Unlock semaphore associated with COM2 (see ABC line 100 below)
⋮	
250 DELETE COM1	Delete COM1 and allocate new size.
280 COMMON COM1, A(N1*2)	
⋮	
PROC ABC	Create procedure ABC
10 COMMON COM2	Declare semaphore
⋮	
100 LOCK COM2	ABC will suspend until FREE COM1 in line 210 of XYZ
⋮	
LOGOFF	

TABLE 2.3

CHARACTERISTICS OF BULK STORAGE DEVICES USED

Device	Access times	Transfer rate words/sec	Block size words	Typical segment swap time*
Random access cassette: SYKES Compucorder 100 SYKES Compucorder 120	 1 to 45 secs 0,5 to 30 secs	 330 660	 Variable (= segment size)	 2 to 6 secs 1 to 3 secs
Cartridge disc PERTEC Model 36	40 ms/revolution 10 ms track to track	92 K	120 (= 1 sector)	55 ms
Bulk semiconductor Memory (RAM) (CAMAC resident)	1 μs 30 μs first word (software limited)	25 K (Program Control) 580 K + (DMA hardware)	Variable 64 typical "	30 ms 15 ms + (1,5 ms with hardware error detection)

Notes: *Segment size 600 words (= average program size in Case Study)
+Not implemented in VIPER, data given for information only.

CHAPTER 3

MEMORY MANAGEMENT

3.1 THE MEMORY MANAGEMENT PROBLEM IN INTERACTIVE SYSTEMS

Interactive programming systems require that any statement in a task can be changed, deleted or added in some sort of incremental compilation mode i.e. the entire task or procedure need not be re=compiled and link-loaded. A good interactive system should also support interaction during the execution of the task with monitoring and debugging facilities that do not require the suspension of the task before they are activated. In PROSIC, the forerunner of VIPER, it was demonstrated that even more general interactive facilities can be provided in a mono-programmed system (HEHER, 1976 a, b) which it would be desirable to extend to the multi-tasking environment.

The implementation of interactive facilities requires that the code (which is usually an interpretive meta-code form, but may be compiled machine code) defining a task be expanded and contracted as the interaction proceeds. In a multi-programmed system the difficulty occurs in attempting to allow multiple tasks or pro=cedures to simultaneously undergo this dynamic change in size and structure. Various ad hoc solutions to the problem have been pro=posed and implemented, resulting in equally ad hoc restrictions. For example, two of the real-time interactive systems described in section 1.4.3 which do support multi-programming, restrict inter=active operations to one particular task which must be compiled be=fore operating on any other task. Virtually no interactive operations are permitted on a task once the task is executing. The other two BASICs described in the introduction which have multi-user capability require a fixed memory partition to be assigned to a given task or user and also do not permit any interaction with the running task even though interpretive rather than compiled code is executed. A further equally serious problem, is that all four of these systems have limited (and dangerous) global areas which can be accessed by all users. Nor do any of them support a structured language with nested named procedures, an essential requirement for any modern programming language.

To ...../3.2

To permit interactive multi-programming using a block structured language, it is necessary to allow the segments of code to dynamically expand and contract while maintaining the linking between the various segments of code and data that co-exist in the system. The essential requirement is then that the segments of code used in the system must be dynamically relocatable i.e. it must be possible to move the segment while it is executing. As the performance of the memory management technique is dependent on the efficiency with which segments can be moved, extensive, or slow relinking of segments to perform relocation is undesirable. These requirements can be fulfilled most simply by segments of meta-code which are executed interpretively, and software virtual memory management is of particular relevance to this class of software. An important point is that the memory management features required, could not be implemented using simple base registers, which is a common method of achieving dynamic relocation. The reason is the real-time interactive nature of the software system, as will be clear from the structures described in the following section. The structures employed are superficially similar to an earlier memory management system described by RIETER (1967) but this system was designed for operations of a time-sharing type and would not permit the flexible access to shared data and code segments that is an essential feature of the real-time interactive system VIPER. Hardware virtual memory mapping devices are also not suitable for this type of relocation and they were in any event specifically excluded because of the desire to maintain processor independence. This was specified in order to permit the operating system to be transported to other mini or microcomputer systems in the future. The operating system MERT for example, (BAYER, 1975) which manipulates segments of code and data in a manner roughly analogous to VIPER, is constructed specifically to run only on a PDP 11/45 or 11/70 computer using particular hardware features of that machine for memory mapping and protection functions.

The use of interpretive meta-codes to provide the basic means of relocating segments has other advantages also. The interpretive structure can be utilized by the memory management system to imple= ment a variety of unique features which considerably enhance the

attractiveness ...../3.3

attractiveness of an interpreter. Furthermore, a number of recent implementations have shown that interpretive systems possess some important advantages over systems executing in-line code (OTTO, 1974; HELPS, 1974; ADIX, 1975; BERCHE, 1976; ZEH, 1976). Their only disadvantage, that of increased execution time, can frequently be overcome or reduced by various techniques such as mixed code (DAKIN, 1973; DAWSON, 1973; ZEH, 1976) or micro-coding (HELPS, 1974; REIGEL, 1972). Alternatively, initial development can be performed interactively with later compilation into in-line code. The desirability of this route for software development as opposed to batch compilation has been emphasized by CAINE and GORDON (1975). As the interpretive execution time of the meta-codes currently used in VIPER were acceptable for a range of experimental process control work undertaken in the past (and foreseen in the future) none of these techniques have been implemented in the current system. As the mixed code approach may cause relocation difficulties, micro coding would appear to be the most promising technique for overcoming any speed problems that may occur in future applications. It should also be noted that the execution time penalty of interpretive systems has also not prevented their being used successfully in a wide variety of applications (ADIX, 1975; AGRAWALA, 1976; BIANCHI, 1976; BERCHE, 1976; CAINE, 1975; DIEHL, 1975; FULTON, 1976; GAINES, 1976; HAASE, 1976; HELPS, 1976; NELSON, 1976; PURDUE, 1975; RIAMONDI, 1976).

3.2 STRUCTURES USED IN THE IMPLEMENTATION OF SOFTWARE VIRTUAL MEMORY MANAGEMENT

While developing the concept of Software Virtual Memory Management (SVMM) it became apparent that there were a variety of different techniques that could be employed. In many cases these involved trade-offs in space and time which were difficult to evaluate at the time the system was being developed. One of the major assumptions, for example, was that most of the important segments of a real-time task would fit into memory simultaneously and that the swapping of segments to and from input/output devices would occur with a relatively low frequency. (This assumption was validated by the results of the case study (chapter 6) where all tasks can fit into a

32 K memory system). In retrospect, however, it is felt that some alternative structures could have been used which would not seriously have affected the performance of a resident system and which would improve the performance of a system where a higher rate of input/output transfers was necessary.

The four sub-sections that follow consist primarily of a de= scription of the actual structures used in VIPER as it is felt that this approach contributes to a clearer understanding of some of the alternatives which are discussed in section 3.4. It must be em= phasised at this point, however, that although better structures may exist, the ones that have been used are adequate for many applications and for the application presented in the case study in particular.

The software system utilized divides memory into two main partitions, as was shown in Fig. 2.1. The resident area consists of the various operating system and language processor modules, while the remainder of the memory is available for virtual storage operations. It is the management of this latter memory area as shown in Fig. 3.1 that is the subject of this chapter. The language pro= cessor is placed permanently in the resident area because of the uniformity of command and programming languages, i.e. it is also used as the command interpreter. The information manipulated in virtual memory consists of segments of both code and data.

To control the division of the available memory into segments, two basic structures are employed: one to perform the physical linking of segments, and the other the logical linking. The physical par= titioning is performed in a straightforward manner by means of a doubly-linked circular list, as shown in Fig. 3.1. Each partition has forward and backward printers to the next and previous segments, and also a pointer to the end of the partition. Each partition, called a segment, is of arbitrary size but must be smaller than the physically available memory. A segment is in fact similar to a page in the hardware virtual memory analogy in that it is an in= divisible unit, with the difference that the segment size can vary dynamically. A task could, however, consist of a set of segments

whose ...../3.5

whose total size is larger than the physical memory. The advantage of this structure over that of a hardware-mapped page is that there is always a 1 : 1 correspondence between the page size and the segment size, as they are physically identical elements. This is of particular advantage in the structured programming language used where there is a natural emphasis on partitioning a task into a set of independent but co-operating procedures.

Segments may not only vary dynamically in size, but can also be created, deleted or moved to and from peripheral devices. Both the first and the last and all segments between them can be dynamically relocated. The position of the first and last segments can be adjusted to allocate memory for use by certain fixed segments which cannot be relocated, as shown in Fig. 3.1. These fixed segments are used for assembly language subroutines and could also be used for in-line code produced by compilation of interpretive code, as discussed in section 3.5. (A notable difference between this resident area and the resident area found in many commercial real-time operating systems for minicomputers, is that it can be expanded on-line.) Some examples of the segments used in VIPER are shown in Figs. 3.5 and 3.6.

3.2.1 Segment and variable descriptors

Each segment in the system is headed by a table consisting of one or more descriptors which describe both the internal structure of the segment and the external resources which it uses.

The first descriptor on the table is the segment descriptor which contains elements describing that segment as well as the list linking pointers. The general format of all descriptors and that of the segment descriptor are depicted in Figs. 3.2 (a) and 3.2 (b) respectively. The first word of the segment descriptor identifies the segment type and the length of the segment descriptor; while the NEXT, PREVIOUS and END pointers are used for list linking and free space control. The fifth element of the segment descriptor EXTERNAL is used for the logical linking of segments as opposed to the physical linking of the forward and backward pointers. The de= scriptors form in effect a "local name space" (LNS) similar to the

LNS of HYDRA (WULF, 1974; JONES, 1975). The capabilities defined within these descriptors are used to control access to both data areas and other procedures. As in HYDRA, the capabilities are manipulated only by the operating system and so cannot be tampered with by the user. In VIPER however, the descriptor table is also used for a variety of other purposes, as described in the following sections.

Each segment is identified either by a name or by its association with an event or device. Procedure segments and shared data segments for instance, are named, while segments used for input/output buffering are identified by the device with which they are currently associated. All segment (and variable) identifiers can be an arbitrary number of characters in length. Within the segment de= scriptor a segment normally, but not necessarily, contains additional information which describes the structure of that segment. The descriptor of a procedure segment, for example, (Fig. 3.5) contains an additional 12 words containing information on the access rights and sub-structure of the segment, in addition to scheduler parameters if it is a segment which is known to the scheduler. The same structure is used for all segments containing executable code, whether they are 'main' programs scheduled by a scheduler or sub-routines or co-routines.

In addition to the segment descriptor at the head of the segment, a procedure segment has a table of descriptors, which contains entries describing the data structures used by that segment, both internal and external, i.e. the symbol table plus space for variable values and pointers. Examples of the descriptor types used in VIPER are given in Figs. 3.3 and 3.4. Additional types for which provision has been made but which have not been used in VIPER as yet, are bit and string variables, function references and multi-precision variables. The various descriptors are of different sizes and can appear on the descriptor table in any order. The numeric value of a variable (if any) is contained in its descriptor as are the ASCII characters of the identifier. The ASCII identifier must be retained for the purposes of decompilation in an interpretive system, but is also very useful for a variety of other interactive features. Even

if ...../3.7

if compiled code were used, DASAI, 1977; PIERCE, 1974 and others have shown that there are good reasons for retaining the symbol table for symbolic debugging purposes. Each element of the descriptor table has a structure identical to that of the segment descriptor: a descriptor head, an information section of variable length (typically one to four words) and an identifier or arbitrary length. The variable-length information and identifier fields of the descriptor are specified by fields within the descriptor head. The descriptor head also contains a field which defines the type of descriptor. Within a 16-bit word these fields result in certain limitations, viz. a maximum descriptor length of 64 words, 32 descriptor types and identifiers up to 16 characters in length. Within many of the descriptors of both procedure and data segments are capability entries which protect the segment and define the right of access to the segment from other segments.

This organization of the descriptor table or local name space is very efficient, not only in terms of bit packing density, but also in terms of the accessing and manipulation routines, which are identical for all types of descriptor table elements. In the 25 procedures of the Case Study the average length of the descriptor table is 178 words which is 28% of the average segment size of 638 words. The space required is considered well spent in view of the uses and benefits of the table.

3.2.2 Father/son relationship*

The logical structure of the SVMM determines the hierarchical relationship between segments. The basic element is the father/son relationship that results from one segment invoking another, as shown in Fig. 3.7. The father contains external reference descriptors in its variable table which define the external procedures (sons) used by itself. If this procedure is currently a segment residing in physical memory, the descriptor in the father will contain an absolute pointer to the location of the procedure, which is now his son. Simultaneously with the establishment of this pointer, the external pointer is set up in the son to point back to the father.

This ...../3.8

*The reverse-gender notation which may have been more acceptable to modern trends seemed singularly inappropriate in view of the fact that these fathers lend, trade and otherwise dispose of their progeny in a most perfidious manner.

This double linking is essential if segments are to be moved efficiently, but is also useful for a number of additional functions.

The simple father/son relationship is similar, in the FORTRAN sense, to a main program (the father) calling a sub-routine (the son), but in SVMM this is not the only means whereby a father can acquire or create sons. All segments are in fact spawned fron one original master segment which is created when the system is generated. The logical structure is not static, however, and the relationship between segments changes dynamically. Segments may be assigned to new fathers or they may temporarily acquire a 'stepfather' as would occur during the re-entrant execution of a procedure. An example of this type of access is shown in Fig. 3.8. (Note: Provision for this re-entrant access has been made in VIPER but as it was not required for the Case Study experiment, it has not been implemented in the current version of VIPER.) Segments may also be permanently or temporarily fatherless if this defining segment was deleted or swapped out, for example. Fathers can also voluntarily release their sons if they are no longer required, with the links to the son and the return link from son to father being zeroed in this case.

If a segment is moved, two adjustments must be made, each re= quiring a search of a descriptor table. First, the descriptor table of the segment to be moved must be searched to find any active sons. The back pointers from these sons to the fathers are then adjusted appropriately. If the segment is being deleted or swapped out, the pointers are zeroed. Secondly the descriptor table of the father of this segment (if there is one) must be searched for references to the segment which is to be moved and the pointer in the external reference descriptor which refers to this son must be adjusted (or zeroed). The overhead involved in adjusting the externals when moving segments is therefore not negligible (2-3 millisecs on the VARIAN). Without a firmware move instruction, however, the time taken to perform the actual physical move is far more serious - 14 millisecs for 500 words. If a known procedure is referenced, i.e. one which is a son, negligible overhead is incurred because an absolute pointer to the segment exists. If, however, an unknown procedure is invoked a search of the resident segments must be made for the required segment.

(If ...../3.9

(If the segment is not found, a directory segment obtained from an external device should be searched.) A simple linear search is adequate because even with a hundred segments the maximum search time is of the order of 5 to 6 millisecs. Certain memory allocation algorithms are used, however, to reduce the typical search time to 1 to 2 millisecs and as even this occurs only the first time the procedure is referenced there is no need to maintain any associative or hash tables.

If the segment is resident, the mean search range will generally be far less than half the resident segments due to a locality of reference that results from the virtual memory operation. When a segment is created or obtained from a peripheral device the memory allocation algorithm tends to place the segment within the locality of the originating segment, i.e. the father (see 3.5). The search is therefore first made within the locality of the requesting segment, and continues until either the required segment is found or the search ends on a return to the original segment via the circular list. One example of father/son interaction may serve to illustrate the general nature of the strategy.

If a segment is spawned by a father within some locality of its father, but is later released by its original father and adopted by a new father (this may be either a new 'true' father or a stepfather) the locality of reference will quite likely have been destroyed, but only for the first reference. Thereafter the new father will enjoy direct access to his son until such time as he releases him. The worst case is therefore that of two or more fathers, who are not within the same locality, competing for ownership of the same son. As explained above the overhead associated with even this (unlikely) worst-case condition is not severe, being of the order of 2 to 3 millisecs, each time the son is transferred.

If a segment must be swapped out, the segment descriptor is left in memory and becomes a directory element containing information about the location of the body of the segment on the external device. As the remainder of the descriptor table is swapped out with the segment, including the external reference descriptors containing pointers from father to sons, the father/son links cannot be pre=served when the father is swapped out. (Conversely the links can be preserved when a son is swapped out because the pointer from son to father is maintained within the segment descriptor.) When a father is swapped back in and needs to reference a son again, a search for the son must therefore be made, taking typically 1 to 2 millisecs as described above. This overhead is one of the dis=advantages of using absolute memory pointers instead of indirect pointers via a resident directory. Preliminary investigations had shown, however, that in the typical applications envisaged most of the critical real-time tasks would be memory-resident and only the less frequently executed tasks would be swapped to and from a bulk storage device. The results of the case study (chapter 6) indicate that this assumption is valid. In an environment where the swapping rate is higher there may well be an advantage in using indirect pointers via a directory segment - as discussed in section 3.4.

Although superficially cumbersome, this maintenance of father/son linking is in fact quite simple and provides a powerful tool for determining the structural dependencies of the system and a means of constructing a hierarchical error-reporting and recovery mechanism.

3.2.3 Access to data shared between procedure segments

Another important type of logical linking is that used to gain access to data segments. A number of different structures were analysed in some detail for this linking and the one that is presented here is considered a reasonably good compromise between the opposing factors of access time and relinking overhead. At the simplest level, segments are defined and accessed in a manner roughly analogous to that of named COMMON in the FORTRAN sense as was illustrated in Table 2.2. Fig. 3.9 shows the linking used for segments of this sort. As a result of the virtual memory structure however, the segments can be operated upon as if they were files, thus they are conceptually quite different from the static COMMON block of FORTRAN. Furthermore, the structure of the data segments permits a semaphore to be incorporated

in ...../3.11

in the data segment descriptor which is used for synchronizing procedures which access the data segment. In addition to being available for manipulation directly by synchronization primitives, this semaphore has also been used to implement the "REGION" construct (HANSEN, 1973). The synchronization functions are described in more detail in section 4.4.

Other data area protection and synchronizing techniques such as "KNOWS clauses" GORD (1976) could also be implemented using the SVMM structures, but are not included in VIPER.

References to shared data items are performed as follows:

Each procedure which accesses the shared data contains a declaration descriptor (A). (The capital letters in parentheses refer to the labelled elements of Fig. 3.9.) This descriptor contains a pointer (H) to the data segment, an access code (G) defining the current access rights of this procedure, and the name of the data segment, as shown in Fig. 3.4 (e). Within the access code (G) is also an identity field which is used to identify variable descriptors associated with this declaration.

The data segment is headed by a defining descriptor (B), Fig. 3.5 (b), which contains the name of the segment, a pointer to the start of the data area (I), a password pointer, the location of this segment on a mass storage device and a semaphore. The descriptor head identifies the type of segment. The external reference element (C) of the defining descriptor is used to point to the procedure which is currently locking this data segment as a result of a sema= phore operation. (Procedures which are suspended waiting for access are kept on another list maintained by the dispatcher.)

In addition to the external reference pointer which defines ownership of the segment, the data segment has a descriptor table (J) which contains an external reference descriptor (D) and Fig. 3.4 (f), for each procedure which references it. This double linking of data and procedure segments is an extremely powerful tool for analysing the overall structure and data relationships of a set of tasks and enables many of the pitfalls of the strictly FORTRAN-type labelled COMMON to be avoided.

Within ...../3.12

Within each referencing procedure, each reference to the data segment is defined by a variable descriptor (E), Fig. 3.4 (a) and (c). The descriptor contains either an absolute (F_A) or relative (F_R) pointer to the location of that element in the data segment, as well as a copy of the identity word that occurs in the declaration descriptor (A). This identity is copied to all referencing variable descriptors which reference a given data segment, to enable the absolute pointer to be adjusted if the data segment is moved. The access field (G) in the defining descriptors (E) can be set independently to protect any particular element of the shared data segment.

The pointers (F) in the referencing descriptors (E) can be of two types:

1. Absolute.

2. Relative to the start of the data area in the shared data segment.

The relative pointers are used in order to preserve the location of data items in the shared data segment when either a procedure or shared data segment is swapped out. When a procedure segment is to be swapped, for example, the descriptor table is searched for all references to shared data segments and the corresponding pointers converted from absolute to relative by subtracting the position of the data segment (H) and the size of the data segment descriptor table (I) from the absolute pointer (F_A). (Relative pointers are flagged by being complemented i.e. a negative value represents a relative pointer.) No action is taken when a segment is swapped back in until the first reference to a shared data item occurs. At this point, the relative pointer (F_R) is converted back to an absolute pointer. This is performed by using the identity field (G) to index up to declaration descriptor (A) which contains (or can obtain) a pointer (H) to the data segment. In the data segment is a pointer (I) to the start of the data area which is then used to construct the absolute pointer (F_A).

This ...../3.13

This algorithm ensures that the more critical tasks and data areas which are likely to remain memory-resident have fast, direct access to the common areas, while the less critical tasks which may have been swapped out will have to re-establish their links (but with increased overhead only on the first access - thereafter they too will have direct pointers).

All references to items in data segments are checked for access violations. The overhead associated with this mapping and checking is of the order of 5% compared with local variable references, i.e. a procedure using only shared data would take approximately 5% longer to execute than the same program using only local variables. This overhead is considered minimal in view of the importance of preserving the integrity of shared data at all times. Furthermore typical tasks use a mix of data types. In the programs of the case study, for example, the average increase in execution time is less than 0,5%, with a maximum of 2% on one procedure (ENGUNITS) which makes many references to common elements. Table 5.1 shows the result of various measurements or shared data access times.

If a procedure segment which references a common area is moved, the descriptor table of the procedure must be searched for the common declaration descriptors (A) to find the data segments referenced by this procedure. The descriptor table of the data segment (J) must then be searched to find the pointer (K) in the descriptor (D) so that its value can be adjusted appropriately. The pointer (C) may also need to be adjusted.

If the data segment is moved the following operations must be performed. The descriptor table of the data segment (J) is searched for procedure references (D) (K). For each procedure found, the procedure descriptor table must be searched for the corresponding declaration descriptor (A). Having found this descriptor, the de= scriptor table must be searched once more to find all reference descriptors (E) which have a matching identity (G). The absolute pointer (F_A) in the descriptor can then be adjusted. (If the pointers

in ...../3.14

in (E) had been set relative as a result of a swapping operation, pointers (K) and (H) would have been zero and therefore no searching would have taken place.)

If a new descriptor is added to the data segment as a result of a new procedure referencing this data area (this can occur dynamically), then the procedure described above must be performed to adjust the pointers (F) in the reference descriptors. The pointers (K) in the procedure reference descriptors, need not be adjusted however. The value (I) in the data segment descriptor must also be updated to reflect the increased size of the data segment descriptor table.

One of the limitations of this method of accessing shared data is that the data itself cannot contain pointers to other data segments i.e. an indirect address within a data element. All addressing must be performed via the descriptors in order to allow the operating system to perform the necessary adjustments as segments are moved. This is not a serious limitation, however, as the interactive language elements of VIPER are intended for applications programming where pointer manipulation is both undesirable and seldom required. HOARE (1975b) has pointed out the dangers of using pointers within data areas and emphasised the importance of data reliability. Pointers are far better handled within the protected capability lists (COSSERAT, 1975) which are manipulated only by the operating system. Routines which do require pointer manipulation are coded in Assembler and located in the fixed segment areas - Fig. 3.1. (They could also be coded in a high level language for compilation into in-line code but this is not implemented on the current system. See also the comment in section 3.5.)

3.2.4 Parameter passing

Parameters are passed between segments by passing addresses. Parameter types are matched, and must agree. The actual structures used for parameter passing are illustrated in Fig. 3.10. When a father passes a parameter to a son, the relative address of the actual parameter descriptor (B) is copied into the corresponding formal parameter descriptor of the son (C), a single bit being set in the head of this descriptor to indicate that it is an external reference. A further bit is set in an access word (D) of the formal paramter to distinguish between formal parameters and external references to data items. To complete the uniformity of access mechanisms between parameters and shared data items, an additional bit field is established in the formal parameter access word as for shared data references (Element (G) of Fig. 3.9). This access subfield defines the type of operations permitted on this parameter.

Protection of parameter passing is performed with a capability-like mechanism with the access attributes of a parameter being passed (copied) from segment to segment. As in other capability-based systems (COSSERAT, 1975) the access attributes can be decreased but never increased in the copying operation. The VIPER implementation does not have the generality of other capability-based systems (FABRY, 1973; WULF, 1974; JONES, 1975; COSSERAT, 1975) which are intended primarily for the writing of operating systems, but the restricted set of operations permitted is adequate for the application-oriented software for which it is intended.

In VIPER the types of parameter passing allowed have been intentionally restricted to provide security. Table 3.1 lists these types and their default access states. All other mappings are illegal.

The detection of illegal mappings is performed at the CALL-SUB set-up time while access violations are checked on each reference to a formal. When passing array variables, only whole arrays can be passed i.e. no equivalencing can be performed and the dimensions of the actual array are used in double subscript references. Code or data outside of the array therefore cannot be overwritten. The checking that is applied by default is sufficient to detect the majority of programming errors, but if this is insufficient, additional checking can be added under program control. The default access states of the formals shown above, for example, can be changed from read and write access to read only if this is required (but not from read to write!)

Setting ...../3.16

Setting of the access states of the actual parameters can also be exercised to affect control of parameter passing. By forcing the state of an actual array variable to read only, for example, before passing it as a parameter, it can be ensured that it will not be written into. Conversely by setting its state to write-only until after the subroutine call will ensure that it is not used before being written into by the subroutine. Control in this way is performed with explicit program statements, as illustrated in Table 3.2. Although syntactically somewhat cumbersome, the infrequency with which the default states need be overridden makes the provision of more sophisticated syntactic structures unnecessary.

Parameter passing is in effect a form of 'domain crossing' in HYDRA (WULF, 1974) terminology, with templates specifying the capabilities of the formal to actual parameter translation. In VIPER however the template does not need to be passed as an actual parameter, as the system has access to the descriptor tables and extracts the information required for template matching. While more restrictive than the generalised HYDRA capability mechanism, this implementation is adequate for the simple high level language used. The template matching technique can also be used in Assembler Coded routines, however, with some restrictions on the permissible forms of parameter access.

Although there is a certain overhead involved in this detailed verification of parameter passing, the checking is considered essential in view of the fact that this interface is one of the most troublesome and error-prone areas in programming, as has been stressed by COSSERAT (1975), HOARE (1975a), GORD and MAHON (1976), ZEH (1976) and others. The overhead involved must also be viewed in the context of the interpretive system, as the time required to establish linking between formal parameters and actual parameters, is roughly equivalent to the execution time of a single statement with a similar number of operands.

On the VARIAN 620i, for example, (4 μs cycle time), the time to perform a CALL-SUB-RETURN sequence passing five parameters is 6,9 millisecs, (which compares favourably with the 6,25 millisecs taken

...../3.17

taken to perform a GOSUB with parameters in the original BASIC where no access checking is performed). Once the formal to actual parameter translation has occurred however, references to formals are handled very efficiently. An operation involving two formals such as X = Y+ Z, for example, executes in 2,4 millisecs in VIPER compared with 8,8 millisecs in the original BASIC. The same operation on local variables takes 2,3 millisecs so that mapping and accessing checking performed on each reference takes only 4% longer, an entirely reasonable overhead in view of the importance of this type of checking. (These absolute times can also be reduced by a re= organization of the interpretive meta-code, as discussed in chapters 5 and 7.)

3.3 BULK STORAGE MANAGEMENT

The three bulk storage units which have been used in testing VIPER were described in section 2.6 and listed in table 2.4. They are:

1. Random access cassette.
2. Cartridge disc.
3. Semiconductor bulk memory (CAMAC resident RAM).

The management of these three devices is described briefly here in order to clarify the need for and usefulness of alternative SVMM structures.

The use of bulk storage devices for program swapping in VIPER is complicated by the fact that the segments of code can change dynami= cally in size. It is therefore not possible to allocate a fixed area of a unit for storage of a particular module and to swap it to and from the same area each time. This is analogous to the problems of file system management where the size of files may expand and contract dynamically. There is a wide variety of bulk storage memory allocation algorithms in use, which can be broadly classified into sequential and block allocation strategies. The essential characte= ristics of these two strategies are described briefly below.

1. <u>Sequential allocation</u>. In these schemes the expected size of a module (file) is estimated and space allocated accordingly. If the module is shorter than expected, space is wasted, while if longer than expected, additional non-continuous space, an "extent", must be allocated on some other area of the device. Only a finite and relatively small (10 to 20) number of extents is typically permitted. Various heuristics are used to deter= mine how much additional space to allocate when the first allocation is filled. When a module is deleted it may or may not be possible to recover the space released. In the Hewlett Packard RTE File Manager, for example, this free space can only be recovered by a packing operation which literally moves all files on the disc to close up any gaps. This compaction operation is lengthy and can only be performed in special circumstances viz. no file on that unit must be currently open. This re= striction may prohibit any disc packing operations during times when the system is active and they would have to be scheduled during system maintenance periods. (In the system used in the Casy Study, chapter 6, a special utility was written to perform a disc pack at 12 pm, every night. At that time certain open files can be closed at the shift change to permit the pack to be performed. Two to three minutes of recorded data can be lost while the packing operation is in progress, however.)

2. <u>Block allocation</u>. The bulk storage device is divided into equal size blocks typically 64 to 256 words in size. A table is then maintained which has one bit to represent the availability of each block. When space is required blocks are allocated according to some algorithm and the appropriate bit set in the free block table. The directory entry for the file points to the first block while the remaining blocks are link-listed i.e. each block contains a pointer to the next block. Any number of additional blocks can be simply allocated if the file expands in size. When a file is deleted all the blocks it was using can be de-allocated and returned to the free block table. No packing operations are ever required and all the storage space is used efficiently. The disadvantage of the block structure is the

speed ...../3.19

speed with which files can be stored or retrieved. Due to the block linking and other system-related factors, the blocks must invariably be moved into a buffer first. This overhead typically takes a time equivalent to the time to transfer more than one block, so that when working with a rotating device like a disc, the writing operating can only use every third block. Transfers to and from bulk memory therefore take at least three times as long as in the sequential case.

Both algorithms, therefore, have certain disadvantages which it seems will not be overcome until a measure of intelligence is provided in the bulk storage unit itself. (It could then, for example, be treated as a sequential device externally even if organizing itself on a block algorithm internally. This aspect is discussed further in chapter 7.)

The cassette unit is used in a sequential mode only, i.e. an entire segment is written out sequentially. Under certain circum= stances a record can be overwritten with a new version of a segment and this has been used to operate a system with only a cassette for bulk storage. (With limited memory this configuration has of course a very poor performance.) The disc and CAMAC (RAM) bulk memory units are operated in a block mode, the block sizes used being 120 and 64 words respectively. A free-block-bit-table is kept in memory and this is used to allocate blocks of storage to requesting routines. When a segment is read back out of bulk storage (disc or RAM) the blocks are automatically de-allocated as no permanent directory is maintained of segments stored on these devices. The current address of a segment, if it is on a bulk storage device, is contained within its segment descriptor (see section 3.2.1 and fig. 3.5 (a)). This algorithm ensures that when using bulk RAM the combined space of the local (computer) memory (e.g. 18 or 19 K in a 32 K system) and the bulk RAM (typically 16 K to 64 K) are available for program storage. The bulk storage therefore provides in effect an extended local memory space which is the characteristic of virtual memory management.

None ...../3.20

None of the three devices used for bulk storage can be considered ideal: the bulk RAM because it is volatile, the cartridge disc because it is too big and too expensive and the cassette because it is too slow. The object of using these devices was to demonstrate the operation of VIPER with devices having a range of access times as well as to overcome the immediate memory space problems on a 16 K machine. Devices which would appear particularly suited for software virtual memory management operations are bubble memory for the fast access, non-volatile extension of local memory space and a floppy disc unit for storage and back-up. An important point is that these two devices are bracketed in terms of access times and transfer rates by the three devices which have been used, thus ensuring that they can effectively be used in a software virtual memory management environment.

3.4 ALTERNATIVE STRUCTURES

The primary disadvantage of the structures chosen is the need to release (zero) the links between father and son and between procedure and data segments when a segment is swapped out. When the procedure is swapped back in again, it must search by name for any external segments which it references before it can once more establish the direct links. (Once in memory, the direct links between segments are maintained even if a segment moves.) As mentioned in the introduction to this chapter, this algorithm was initially selected because it was anticipated that most of the time critical tasks would be memory-resident and only the less frequent tasks would find themselves being swapped out. Experience with the use of both disc and bulk semiconductor memories, however, indicates that SVMM is capable of supporting a much higher swapping rate, or equivalently, of running real-time tasks of a size which cannot fit into the local computer memory.

Although the existing structures work satisfactorily with the higher swapping rate, there is an overhead of 2 to 3 millisecs involved in this re-establishment of links to external segments. This is small compared with the swapping time of 30 to 70 millisecs, i.e. the overhead is of the order of 10% of the swapping time. As noted in table 2.4, however, if alternative bulk memory control hardware was used, the swapping time could be reduced to less than 2 millisecs, at which point the relinking overhead is substantial. An analysis of alternative organizations is therefore of interest in order to determine the efficiency of SVMM when using such high speed devices. The overhead incurred in establishing and deleting the links to segments can be reduced by maintaining a segment directory which is kept in memory. Entries in the directory would then point to the segment. Each segment would have an identity number associated with it from which the segments' position in the directory could be quickly computed. (The identity number could simply be the relative or absolute position of the entry in the directory.) The absolute pointer in a descriptor to another procedure would then be replaced by the identity number of that segment permitting the segment to be found by indexing via the directory. This identity number would be left intact when the segment was swapped out to a bulk storage device and would not need be zeroed as is the case when an absolute pointer is used. If a segment were moved, only the directory entry would have to be updated.

This mode of operation is proposed in an extension of VIPER which is discussed in chapter 7. To illustrate the problems that must be solved in formulating new structures, some of the difficulties involved with this approach are noted below. (Solutions to all these difficulties have not yet been found!)

1. Segments are dynamically created, and must be allocated an identity number and the corresponding directory entry. Over the lifetime of a system, which may extend over several months, as old segments are deleted and new ones created, the directory will grow steadily larger with no direct means of re-using old entries, for the reasons given below.

2. Before an old entry can be deleted or re-used, it must be ensured that no segment currently in the system or which is likely to become known to the system, references this particular identity number. As there are no direct links to inform the system

which 3.22

which segments are referenced by another segment, every segment in memory and on the bulk storage devices will have to be searched to find and delete references to the segment which it is required to delete. As segments which have been stored on removable devices, such as disc cartridges or cassette tapes, may not be accessible, they will have to have had all the ID elements in their descriptors deleted before being stored, i.e. the same as is done with absolute descriptors. This searching operation will be lengthy but as it may only be necessary infrequently, this may be acceptable. It is in effect a form of garbage collection, a process which is usually performed either when the system is idle or when space is short.

3. The alternative to this searching operation is to perform a check each time a segment is swapped in to verify that the ID element held in some descriptor does in fact match the name of the corresponding segment i.e. no search is involved, merely a test whether the name of the segment does match the expected name held in the descriptor. The test must either be done for every external descriptor on the table, which requires a search of the segment descriptor table (which may be even longer than the search for the segment directly!) or it can be performed on the first reference to the segment (as is done in the case of absolute pointers). In this latter case a flag must be set indicating that the test has been done. A possibly attractive solution is to change the relative ID value at this stage to an absolute value in a manner similar to the existing method of handling references to shared data segments (see section 3.2.3). These absolute pointers would then of course have to be converted back to relative pointers before the segment was swapped out - once more requiring a search of the descriptor table to reset all external descriptors.

4. One of the objectives in the development of VIPER was to plan towards its use in a multiprocessor environment. The relocatable segments of meta-code are particularly attractive in this environment as they can be sent to any processor in the network and executed in any available memory space. The information

carried ...../3.23

carried in their external descriptors specifies all the resources which may be required by that segment in its new environment. A bulk storage module (either RAM or possibly bubble memory) is an ideal element for shared storage in this environment and segments stored there could be swapped in and executed on any processor using current structures. If the identity element plus indexing were used instead, then either the directories would have to be identical in all processors, or it would have to be noted when a segment changes processors and the ID elements adjusted (zeroed) at that time; or the ID elements must be deleted in segments which are stored in the shared module (which contradicts point 2); or the checking technique in 3 above must be used.

From the various points which are made above, it is clear that there are no simple, clear-cut alternatives to the structures which have been used in VIPER. The VIPER structures were arrived at after many months of careful thought and it could seem that they are the best under the assumptions that were made viz. most time critical tasks reside in memory. In other environments the factors affecting parameters such as swapping rates, segment size, the number of segments in the system and multiprocessor operation, must be known before optimally efficient structures can be synthesised. In instances where these factors are not known or vary unpredictably, the simplest most straightforward structures may be if not the best, at least not significantly worse than the best. This difficulty of selecting efficient algorithms in an ill-conditioned environment has been observed by SPANG (1974).

3.5 MEMORY ALLOCATION

There are three events which the memory allocator must handle:

1. A request by an existing segment for more space.
   This space must be obtained adjacent to (i.e. at the end of) the segment.

2. A new segment is to be created. The space can be obtained anywhere in memory.

3. ...../3.24

3. A segment must be swapped out to make space for either a new segment or an increase in size of an existing segment.

3.5.1 <u>Additional space</u>

Four events can cause an existing segment to require additional space.

1. The addition of new lines of code to the statement pool of a procedure descriptor.

2. The addition of new descriptors to the descriptor table of either procedure or data segments.

3. The allocation of space for a local array variable.

4. An entry is added to one of the system segments. (Scheduler segment, password segment or syntax recursion list.)

5. The body of a segment is swapped back in from bulk storage.

All these operations can occur dynamically i.e. while a procedure segment is executing or between successive references to a data or system segment.

In general, segments are scattered over memory and are not necessarily contiguous. Bits of free space may exist between segments. If a segment requires more space, a test is first made to see if sufficient free space exists between the segment and the next. If there is, the segment merely expands into the free space and no movement of segments is required. If there is insufficient space, then a compaction operation is performed in the vicinity of the segment requiring space such that the minimum number of segments is moved to obtain the necessary space. In situations where only a few words of space are requested e.g. adding a descriptor to a table, more than the requested space is obtained, if compaction is required. The extra space obtained is left as free space at the end of the segment so that if another request for space is made shortly thereafter (as is quite likely) it can be satisfied immediately without moving any

segments. ...../3.25

segments. If the required space cannot be obtained by compaction then a segment must be swapped out, as described in section 3.5.3.

3.5.2 New Segments

New segments are created when:

1. A new procedure is started.

2. A new shared data segment is formed.

3. An I/O buffer is required.

4. A reentrant data block is required for decompilation (back listing).

5. An old procedure is restored from an input device.

The allocation strategy adopted for new segments is essentially first fit i.e. the first free space area which is large enough is used. In a detailed study of memory scheduling AGRAWALA (1975) has commented on this allocation strategy: "In a swapping system, determining where to place the next arrival in memory can be a very complex task. Heuristics are usually employed to help solve the problem. Quantitatively, now much better are such strategies than first fit, which KNUTH (1968) endorses."

ROBSON (1977) has also shown that the worst case fragmentation is serious for all sytems, but is much worse for best fit than for first fit systems. In addition, fragmentation is not nearly as serious in VIPER because free space can also be collected easily by moving segments. In fact, due to the dynamic properties of segments a certain amount of fragmentation may be quite desirable.

The only heuristic employed in VIPER is to attempt to separate the temporary and permanent segments. Procedure and shared data seg= ments, for example, are likely to settle down to a fixed size after debugging is complete, and are likely to remain in memory permanently if they are associated with time critical tasks.

These ...../3.26

These segments are therefore allocated from the 'bottom' (first segment of fig. 2.1 and 3.1) end of memory upwards, while the temporary segments, such as I/O buffers, reentrant data blocks and scheduler lists which change frequently in size, are allocated from the 'top' (last segment) of memory downwards. This process is simplified by the doubly linked list of segments which permits searches for free space to be made with equal ease in either direction.

If first fit is not possible, i.e. no free space of the required size is available, then one of two actions can be taken:

1. If the total free space in memory (i.e. the sum of all the pieces) is larger than the required area, the space can be obtained by compaction, a process which requires the relocation of one or more segments.

2. One or more segments can be swapped out of memory to obtain the required space.

The decision on which of these actions to perform is even more difficult and complex than the free space selection problem mentioned by AGRAWALA. On the VARIAN which lacks a firmware move instruction, the time taken to move a segment is typically 20 to 25 ms depending on its size and structure. If more than two or three segments must be moved it may therefore be quicker to swap a segment out (30 - 60 ms) than to perform a compaction operation. (If a firmware move instruction was available the movement time could be reduced to 5 or 6 ms, but there would still be some point at which it would be faster to swap than to move.)

In the initial design of VIPER there was no experience to draw upon so the simplest strategy was adopted: if there is sufficient free space it is obtained by compaction, otherwise a segment is swapped out. With a little care in the placement of segments this has been found to work surprisingly well, for the following reasons. The compaction and allocation algorithms tend to cause all the segments

which ...../3.27

which are more or less fixed on size to be packed one after each other from the bottom of memory upwards, with most of the free space occur= ring between the end of this pile and the top of memory, with only a few segments being scattered in this free space. The compaction operation therefore very often involves only a few of these segments. Occasionally free space will occur in amongst the pile of fixed segments, as a result of some interactive operation for example, but the time taken to recover this space is then of little consequence. If frequent movements are taking place these are most probably due to extensive interactive operations by a number of users working simul= taneously, in which case one can be expected to pay some overhead for the facilities one is using. In any event, in process control applications, which usually run 24 hours per day, it is almost impossible to perform such operations more than a small proportion of the time, so that as far as the system is concerned it operates most of the time in a quasi-static environment.

In the latter respect the memory management problem in real-time systems is significantly different from that occurring in batch or time sharing applications (AGRAWALA, 1975; ARDEN, 1975). SPANG (1974) has clearly demonstrated this point by showing that a slight change in the characteristics of one task in a set of 17 repetitively executing programs could change the number of swapping requests by 50%.

3.5.3. De-allocation (swapping out)

When insufficient free space is available a segment in memory must be swapped out to provide the necessary space. Chosing the best segment to swap out i.e. the one which is least likely to be needed in the near future, is as difficult as a "best-fit" strategy when swapping in. Unless the characteristics of the tasks are known and the algorithm is designed accordingly, nearly any algorithm will degrade under certain conditions and will end up swapping out segments unnecessarily (SPANG, 1974; AGRAWALA, 1975).

The algorithm adopted in VIPER swaps out procedure segments in the following order:

1. ...../3.28

1. Segments which are dormant, i.e. which are not on any scheduler list. These segments may have been swapped in to perform either syntactic or editing operations (e.g. the addition of a new line) or for an interactive operation (e.g. examination of the value of a variable in the descriptor table).

2. Segments which are on any suspension list (operator, I/O, semaphore, unit lock or memory).

3. Segments which are on the time list; longest next-time-to-run first.

4. Segments which are on the ready list waiting to run; lowest priority first.

Provision had also been made in the design of VIPER to permit shared data segments to be swapped out, but this has not been implemented as yet. They can be moved to and from bulk storage devices, but only under user control i.e. with program statements or commands. One difficulty with swapping of these segments is the determination of which segment to swap out. A sufficient condition is when all pointers (K) in the procedure reference descriptions (D), Fig. 3.9, are zero, as this implies that all the referencing procedures have been swapped out. User commands can also be used to explicitly release a common area which would also zero the pointers in the reference descriptor. Simple and efficient algorithms can be devised to implement this strategy which would appear adequate for the use in VIPER.

3.5.4 A comment on memory allocation algorithms

No detailed theoretical studies have been undertaken to determine whether the memory allocation and scheduling algorithms are optimal. In general, optimal memory management is of somewhat more concern to large multi-environment real-time operating systems (BAYER, 1975; SPANG, 1974) than it is to a small specialized system like VIPER. The time taken to obtain space for a new segment in SVMM, for example, can be compared with the time taken to recompile a program segment from source code. This recompilation method is used in many disc-based "extended" BASICs to provide an overlay facility; user designated

lines ...../3.29

lines of code being discarded to allow new lines to be loaded into the same space. A complex BASIC system using this kind of overlaying has been described by CARY (1976). It should be noted that this technique not only incurs a significant time penalty but also requires care on the part of the user in constructing the overlay modules.

Even if extensive compaction is required to find space, the time taken to load a new segment in VIPER is an order of magnitude less than the time taken to recompile an overlay module. If no compaction is required, the time to perform the loading operation is at least two orders of magnitude faster. Furthermore, if the segment is already resident in memory, as is more likely to occur when using SVMM than when using overlays, the SVMM "loading" operation can be said to be three to four orders of magnitude faster.

Having achieved a gain of this magnitude there is little incentive to expend effort on optimal management, even if it were possible to achieve a further 50 or 60% improvement. This is particularly true in VIPER where many if not all of the time critical tasks are likely to remain memory resident. Only if the SVMM operations were to be improved to support a higher swapping rate, as is discussed in chapter 7, would a more detailed and thorough examination of memory management be required.

A further point in favour of simple algorithms is their compact= ness and efficiency. MADNICK (1974) has pointed out how complex scheduling algorithms can become self-defeating due to their time and space overheads. Due to the 32K word direct addressing constraint of the VARIAN (and of nearly all current mini and micro computers), space consumed in the resident operating system nucleus is space lost for use in the local portion of the virtual memory space. This factor, together with the difficulty of deciding in many cases what is a better algorithm, is sufficient reason for using the simplest possible algorithms which perform with reasonable efficiency.

3.5.5 Memory allocation extensions

An interesting aspect of memory allocation occurs if on-line compilation

is ...../3.30

is possible, i.e. if relocatable interpretive code can be converted to fixed in-line code. These in-line code segments would be placed in the fixed segment area shown in Fig. 3.1 and would therefore reduce the memory available for virtual memory operations. They cannot be relocated once placed in a fixed segment. Assuming some ratio between the execution times of interpretive code and in-line code, it is clear that given a set of tasks, the advantage of faster execution time as a result of executing in-line code must be weighed against the slower effective execution time that results from reducing the memory available for virtual memory operations. The optimum allocation will vary with the task demands and hence with time, so that an estimation of the optimal memory allocation strategy is a non-trivial problem. In many instances, however, a few tasks can be identified which consume a large proportion of the available processing time (particularly in real-time systems with some repetitive tasks) and in this event a significant increase in overall efficiency could be gained by compiling these tasks into in-line (resident) code. The operating system can be used to identify which tasks are consuming the most overhead, and the most time-consuming operations can be compiled either automatically or under operator control.

An important feature of SVMM is that tasks can be added in-line into the fixed-segment-resident areas shown in Fig. 3.1. This is in strong contrast to many commercial real-time executives, where the tasks must be partitioned into memory-resident and bulk-storage-resident tasks at generation time, and no more tasks (or at best only a very limited number) can be added later. In addition, in SVMM the most recently-added resident task can be deleted and the space used by this task recovered for virtual memory operations. This possibility of executing in-line code must however be balanced against the loss of interactive capability which results when a procedure is not interpretive. As this ability to interact on-line with any procedure is one of the major advantages of SVMM, restraint should be exercised to ensure that this advantage is not sacrificed to obtain marginal gains in throughput.

The 3.31

The technique of 'throw-away' compiling developed by BROWN (1976) which is a middle path between interpretation and compilation, may also be a useful tool for optimization the memory allocation and throughput of the system. Using this technique, a relocatable segment (or portions of it) would be dynamically compiled into in-line code in the fixed segment area. When either additional space is required or interactive operations are required on the segment, the entire compiled segment is thrown away, to be compiled again when next executed. If the interpretive meta-codes are kept in reverse-polish form (which is in any event a desirable representation) this dynamic compilation is fast and efficient as only the code generation portion of the compilation must be performed.

FIGURE 31 ...../3.32

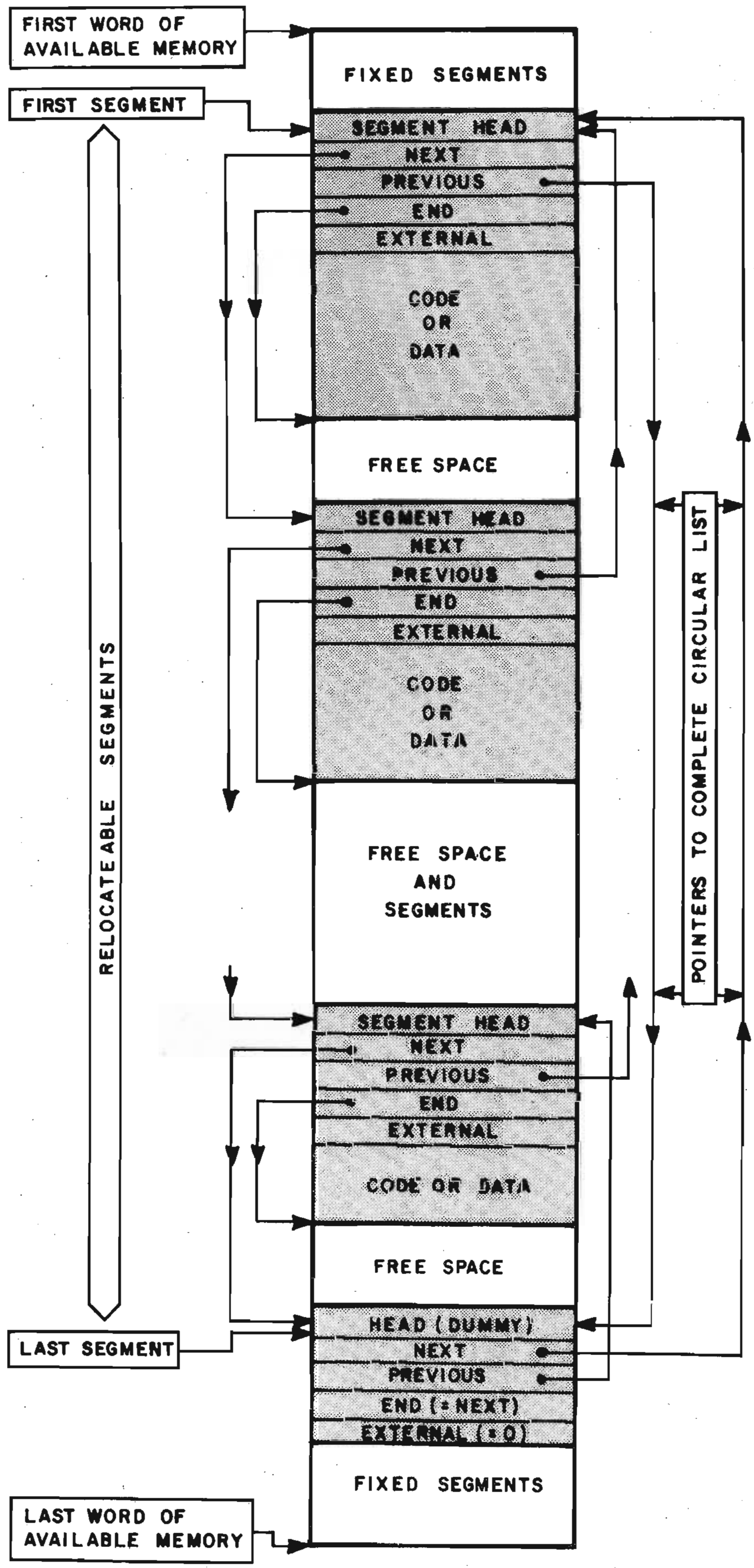


FIGURE 3.1 PHYSICAL MEMORY PARTITION

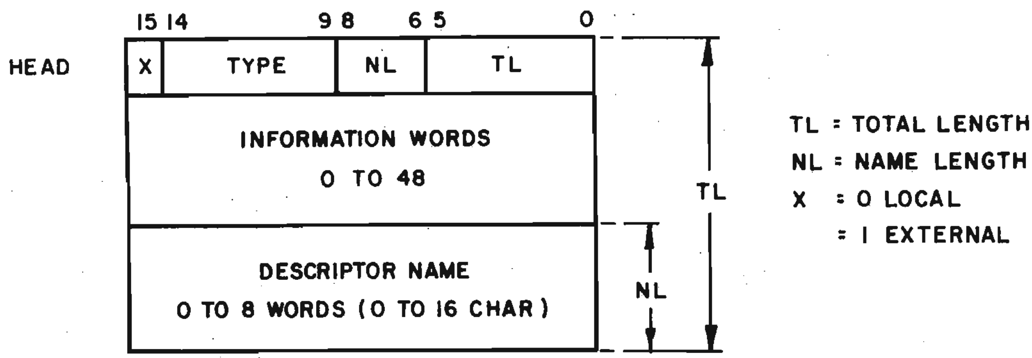


FIGURE 3.2 (a) DESCRIPTOR FORMAT (ALL DESCRIPTORS)

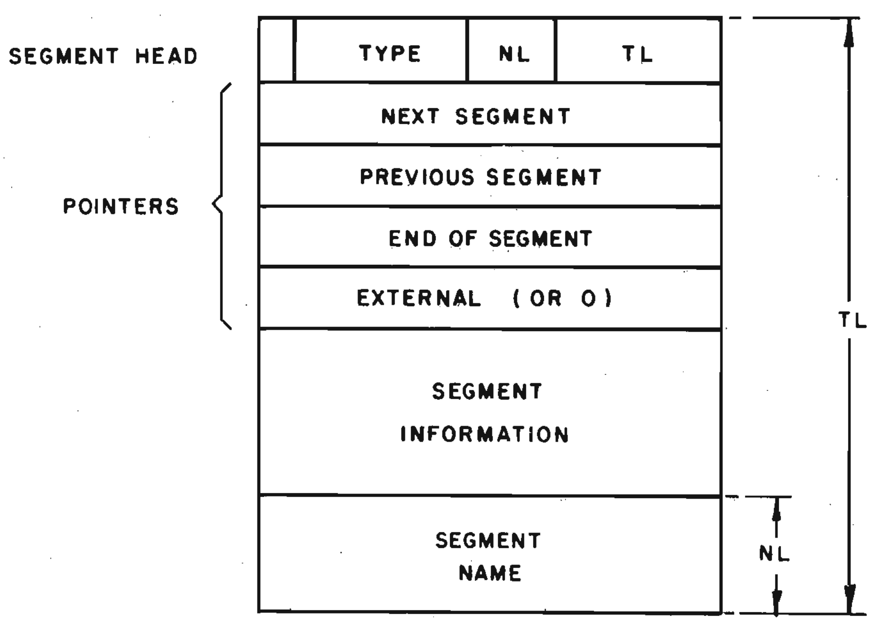


FIGURE 3.2 (b) SEGMENT DESCRIPTOR FORMAT

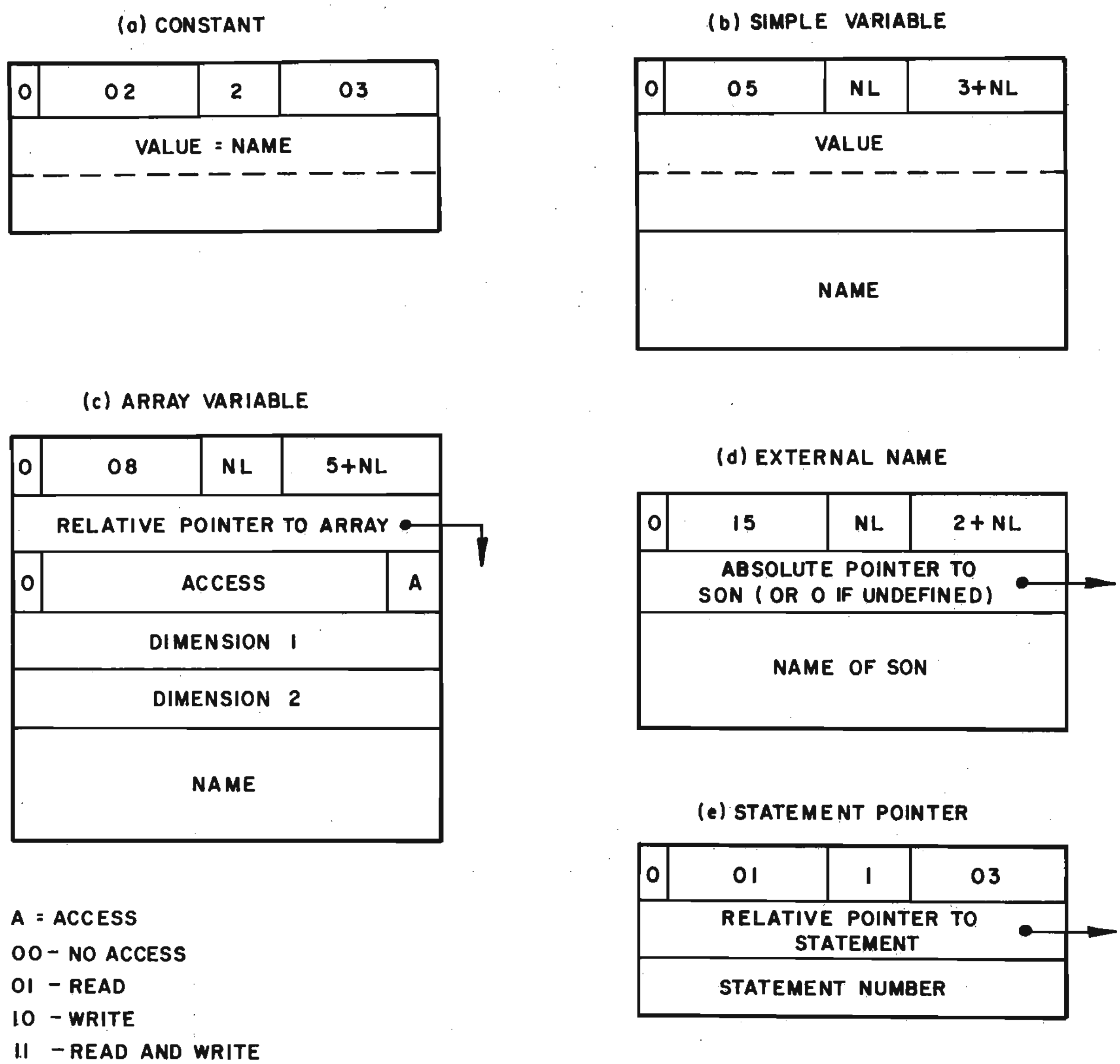


FIGURE 3.3 LOCAL VARIABLE DESCRIPTORS

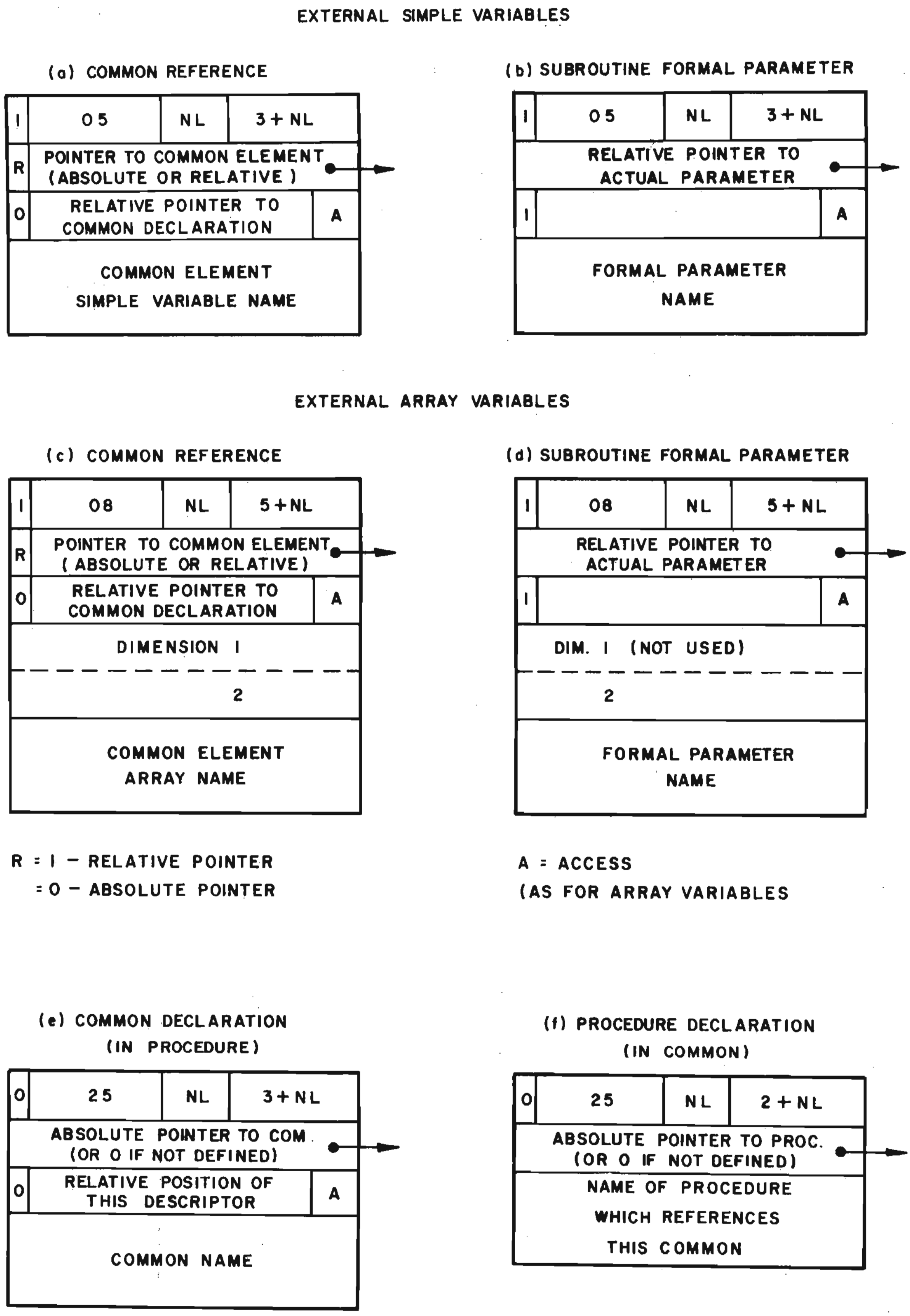


FIGURE 3.4 EXTERNAL ACCESS AND LINKING DESCRIPTORS

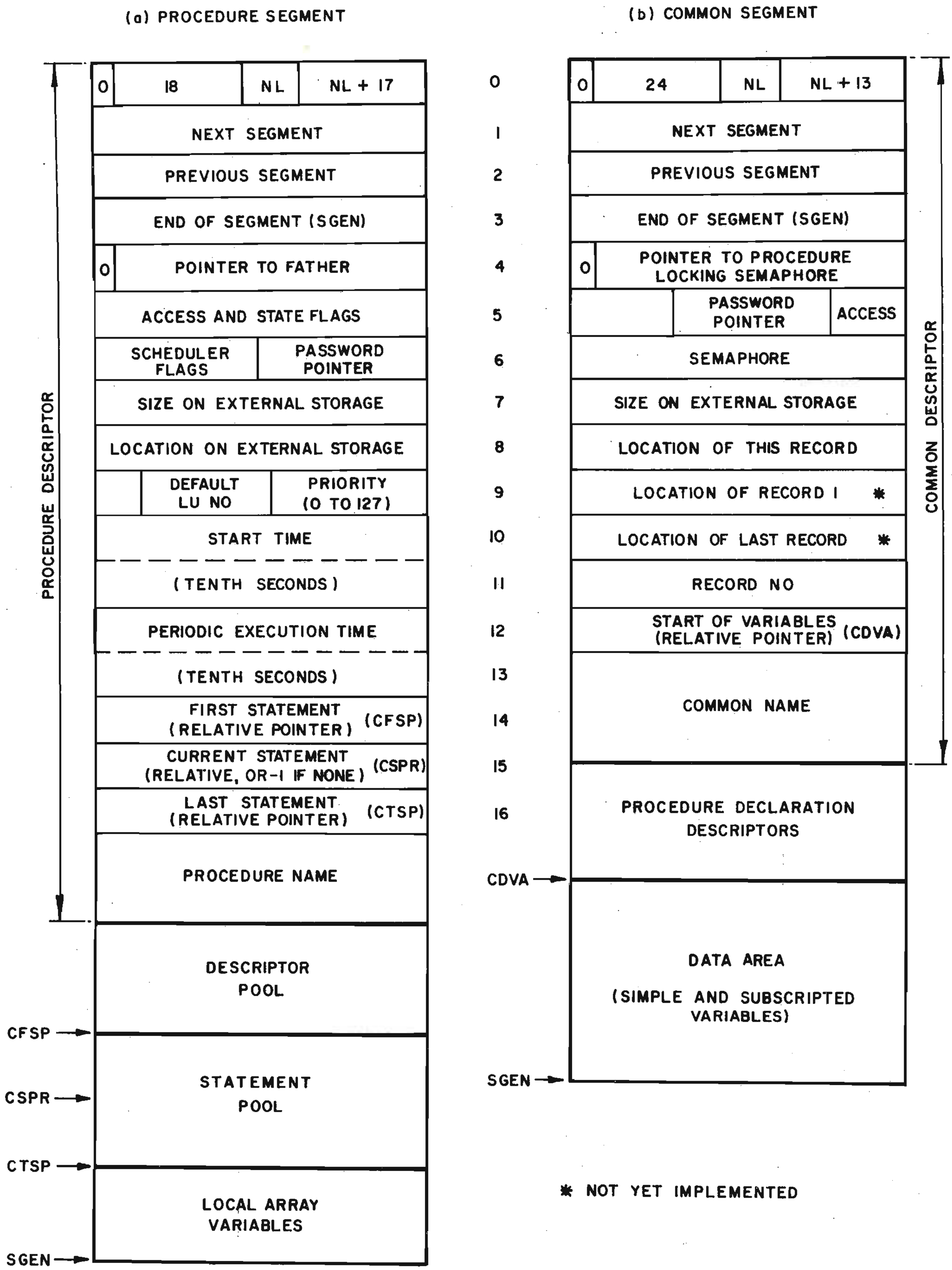


FIGURE 3.5 PROCEDURE AND COMMON DESCRIPTORS AND SEGMENTS

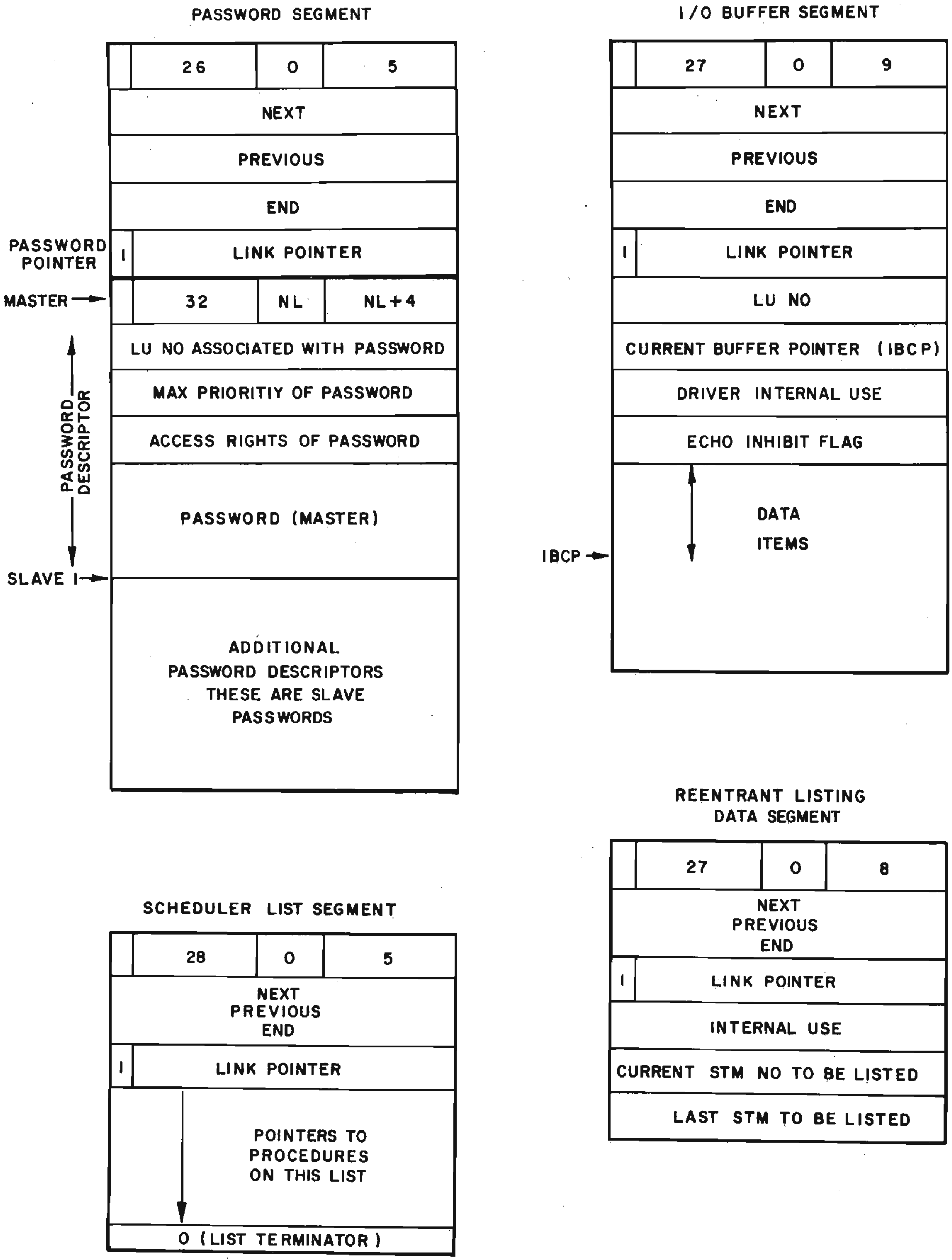


FIGURE 3.6 SYSTEM SEGMENTS

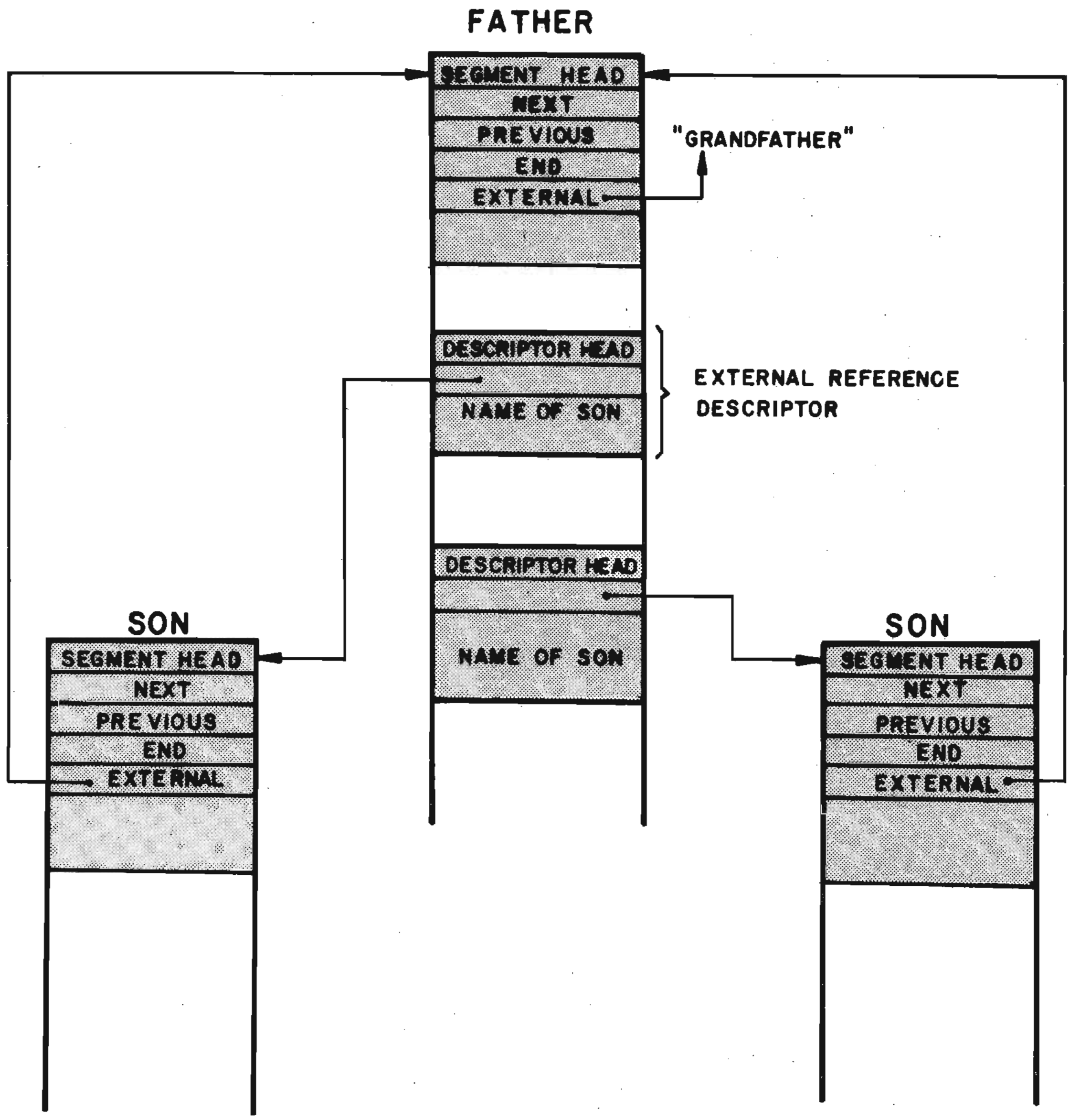


FIGURE 3.7 FATHER / SON RELATIONSHIP

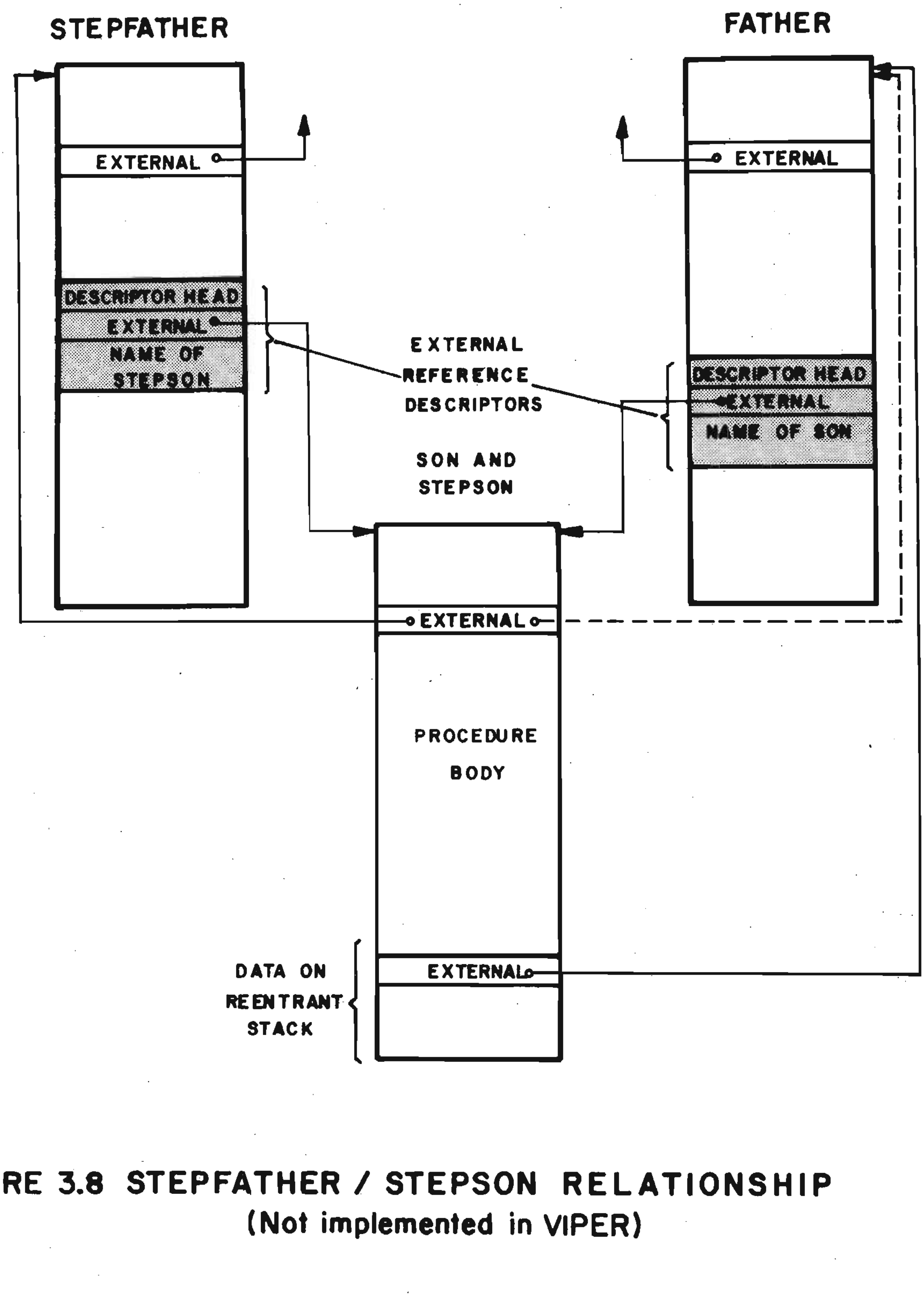


FIGURE 3.8 STEPFATHER / STEPSON RELATIONSHIP
(Not implemented in VIPER)

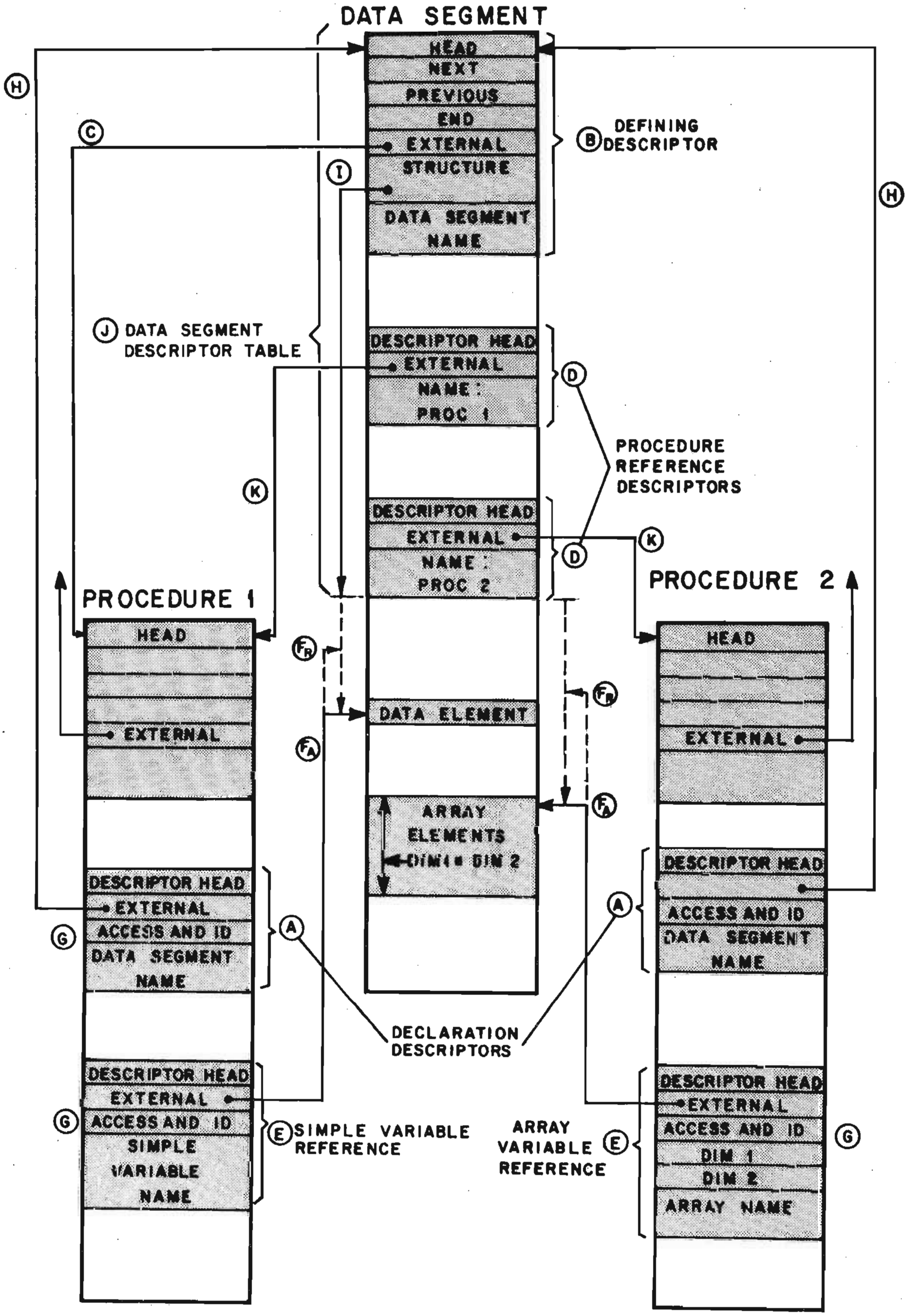


FIGURE 3,9 SHARED DATA ACCESS

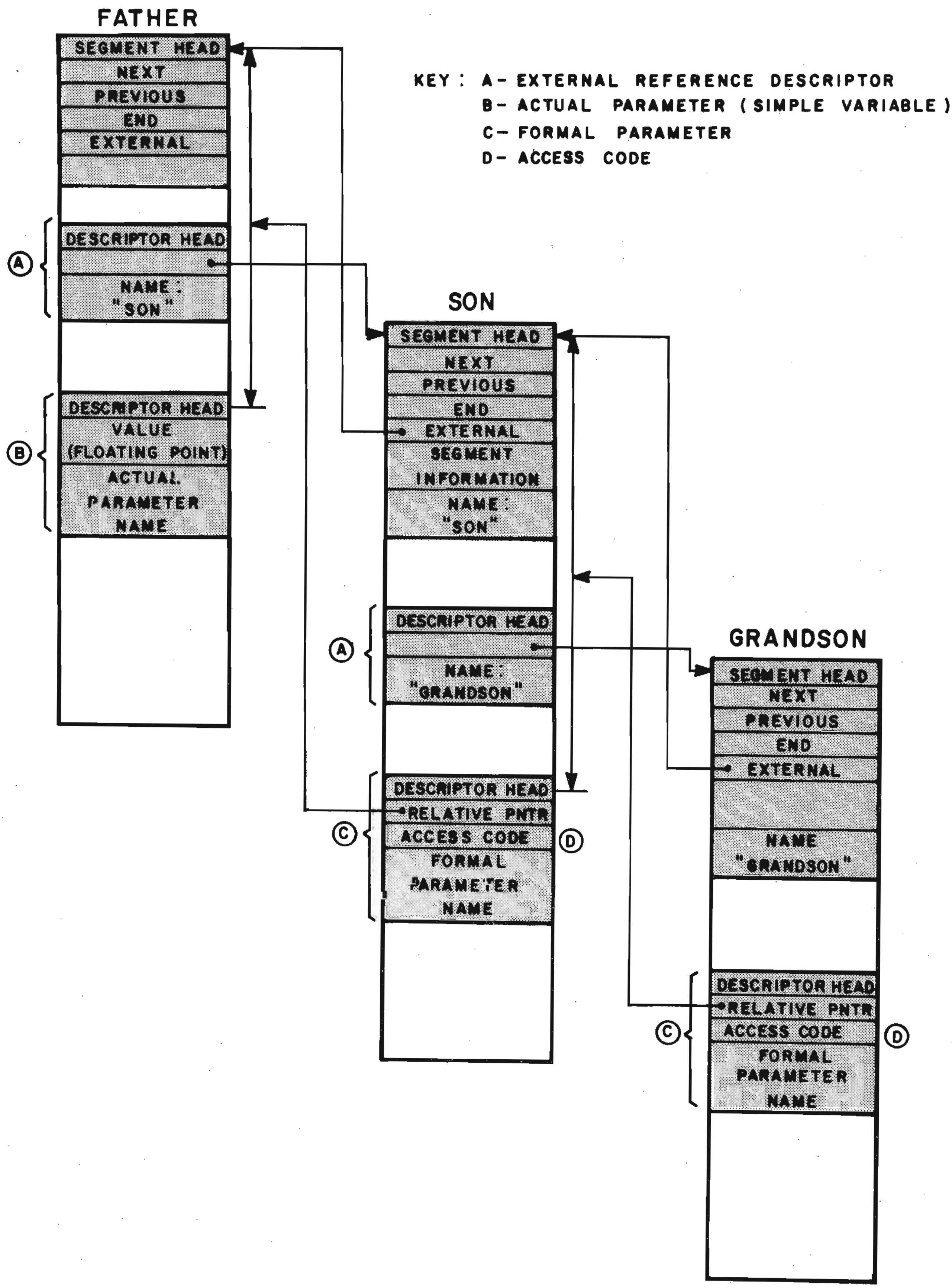


FIGURE 3.10 PARAMETER PASSING

TABLE 3.1 : DEFAULT ACCESS STATES OF PERMISSIBLE ACTUAL TO FORMAL PARAMETER MAPPINGS

Actual parameter type	Formal parameter type	Default access applied in son
Local simple variable	Simple variable	Read and write
External simple variable (common or a formal)	Simple variable	Copy formal access = actual
Array variable (local or external)	Array variable	Copy formal access = actual
Constant	Simple variable	Read only
Expression	Simple variable	Read only

TABLE 3.2 : SOME EXAMPLES OF EXPLICIT ACCESS OPERATIONS IN VIPER

Statement	Comment
DIM A(N)	Local array, access = read and write
.	
.	
ACCESS (A) = READA	Force to read only for call
CALL SUBX (A,B)	
ACCESS (A) = READA+WRITEA	and back to write for local use
.	
.	
.	
SUBROUTINE SUBX (X,Y)	
ACCESS (Y) = READA	Drop access of Y
.	
.	
.	
CALL SUBY (X,Y)	Pass access of X unchanged access of Y is modified.

CHAPTER 4

INTERACTIVE MULTIPROGRAMMING FACILITIES

In this chapter the techniques for producing better software which were listed in section 1.2 are discussed in more detail. The discussion is in two interrelated (and intermixed) parts: the first deals with the more abstract concepts with reference to current literature and the second deals with the implementation of the facilities in VIPER. The five topics covered are:

1. Structured programming.
2. Interactive operations.
3. Protection and error control.
4. Synchronization.
5. Documentation.

4.1 STRUCTURED PROGRAMMING

"I take structured programming to be a term of art signifying a style of programming in which the flow of control is determined by procedure calls and by statements of the type IF ... THEN ... ELSE ..., rather than by the indiscriminate use of GOTO statements. Further, it is usually advocated that the program should be written in a top-down manner. These recommendations, it is claimed, lead to a disciplined method of programming with the following advantages.

1. The program, being modular in nature is easy to understand and check.
2. There is a possibility of proving it correct.
3. It is easier to maintain and modify."

(WILKES, 1976)

The term structured programming has acquired a variety of meanings, but this concise statement by WILKES captures the essential properties of this programming discipline. The development of structured program= ming techniques is a current topic of research and a wide variety of control structures have been proposed and discussed (DAHL, 1972;

MEISSNER, .../4.2

MEISSNER, 1976; BARTH, 1974; NEELY, 1976).

Because of this fluidity, only the simplest and most widely used structures are used in VIPER and no attempt was made to either develop or expand new structures.

The two essential requirements for structured programming are:

1. Modularity of program modules, permitting top down design and step-wise refinement.

2. Suitable control structures which permit indiscriminate use of GOTO statements to be naturally avoided.

A claim of this thesis is that the SVMM facilities complement the goals of structured programming and contribute towards the construction of an efficient software system.

4.1.1 Modularity

In VIPER each named code module, which may be either a procedure or subroutine, exists as a separate segment which can be independently moved to and from bulk storage devices. One of the goals of structured programming is to break-up a task into modules each of which is no more than one to two pages in size (30 to 70 lines of code). In SVMM, therefore, a well-structured program is naturally divided into blocks a few hundred words in size, each of which represents a natural "page" which can be swapped to and from a bulk storage device. This 1 : 1 correspondence between pages and segments is in marked contrast with hardware virtual memory mapping devices where the page boundaries are randomly scattered over the procedures constituting a task. (DENNING, 1970; AGRAWALA, 1975). Only the segments which are currently required (or which are being used frequently) are likely to remain in memory and the other segments will tend to be moved out of memory. Together with the fact that the meta-code segments are smaller than their machine code counterparts, with the result that more of them can fit into memory, this correspondance between pages and segments is likely to result in less time being spent in swapping segments and in

a ...../4.3

a reduction in the probability of pages "thrashing" in and out of memory.

An equally contentious aspect of structured programming is that related to the use of block structure (as in ALGOL type languages) as opposed to Main-subroutine structure (as in FORTRAN). One of the advantages of block structured languages is the better organization of variable referencing which avoids either long parameter strings on subroutine calls or excessive use of COMMON. The use of blank (global) COMMON has, in particular, been pointed out to be most undesirable (NEELY, 1976; HOARE, 1975). The primary criticism of the use of COMMON concerns the fact that it imports variables into a procedure which may not be required there and which may be accidently over= written. These errors can be very difficult to locate.

The main - subroutine - labelled common approach was adopted for VIPER, however, for the following reasons:

1. In a real-time process control environment the use of a COMMON areafor the plant data base is unavoidable.

2. Block structured languages are conceptually more difficult to understand for the process oriented user who is familiar with FORTRAN and BASIC.

3. The ease of using labelled COMMON in VIPER and the protection facilities which are provided, overcome the objections which have been voiced at the use of shared data areas of this type.

4. When synchronization problems are taken into account, the labelled COMMON area is a natual structure for the use of the REGION construct (HANSEN, 1973) thereby simplifying access contention problems.

5. Debugging operations are more difficult in a block structured language because of the need to identify the scope of variables (PIERCE, 1974).

One ...../4.4

One of the claims of this thesis is therefore that the data structuring and protection facilities provided by SVMM enable structured programming techniques to be used in a simple, easy to learn, FORTRAN type environment.

In the programs of the Case Study presented in chapter 6, the FORTRAN programs were already modular in nature. In the VIPER implementation, even further modularization was possible. The program SERVO (Appendix B page B3.18 and B2.5, B2.20) and the error message handling facilities (B3.6 and B2.17, B2.24) illustrate how this modular decomposition can be used to simplify the programs.

The modularity of programs in VIPER, together with the inter= active, operations, also permits an informal, but flexible, top-down or step-wise refinement design strategy to be used. This aspect is commented on in section 4.2.4.

4.1.2 Structures

The control structures incorporated in VIPER are as follows:

1. IF - THEN - ELSE - ENDIF
2. FOR - NEXT
3. DO WHILE - END DO
4. CASE - ENDCASE
5. GOTO

This restricted set of relatively simple structures was chosen as they were considered adequate for the type of software likely to be written in VIPER. Examples of the use of these structures are given in Table 4.1 and in Appendix B.2. To simplify the incremental compilation of lines of code, lines containing a control structure must appear on their own in VIPER. Although a little cumbersome at times, this restriction does ensure that the control statements are highly visible and cannot be obscurred by surrounding code. This is particular true of multiple rested IF - THEN - ELSE - ENDIF clauses

and ...../4.5

and the enforced simplicity that occurs in these nested structures is an open invitation for the insertion of end-of-line comments. This has the double advantage that the programmer is more likely to insert comments in this naturally occuring space, and secondly, that this is the very point at which comments are most likely to be needed to explain the program flow.

The one control structure included which is slightly more complex is that of the CASE - ENDCASE. This statement can assume many different forms (BARTH, 1974; MEISSNER, 1976). In its most general form Meissner claims that "at the advanced level, an extended CASE form is introduced that provides the opportunity to remove the last vestiges of undisciplined GOTO statements from FORTRAN programming". A slightly restricted form of this advanced CASE is implemented in VIPER which sacrifices some of the power of the most general form for syntactical simplicity. Examples of the use of the CASE are given in Table 4.1 and in Appendix B.2.

The simple GOTO was retained in VIPER as it has quite clearly been shown (KNUTH, 1974; DEMILLO, 1976) that it is sometimes required even in well-structured programs to avoid awkward and clumsy con= structions. An interesting observation arose, however, from the Case Study presented in chapter 6. In the translation of approximately 1 300 lines of FORTRAN code into VIPER not a single GOTO was required whereas the FORTRAN code contained nearly 100 of them. This observation indicates that the control structures chosen are adequate for the relatively simple logic structures that generally occur in process control work.

Despite the simplicity of the structures they have a markedly beneficial effect on both the clarity and ease of understanding of the control programs. The VIPER programs are generally considerd far more readable than their FORTRAN counterparts. (See Appendix B).

One of the most important aspects of structured programming in an interpretive system is that it can be used to automatically perform the indenting that provides the invaluable visual aid to program structure

structure. An example illustrating this facility is given in Table 4.1. The manual insertion of indenting is a tiresome and frequently overlooked chore which is especially difficult when programs are changed or updated. Furthermore, real programs are subject to a steady flow of changes and improvements over their lifetimes (HOARE, 1975; KERNIGHAN, 1977) so this problem is not just a development phenomena. In VIPER the automatic indenting is coupled with a proof of the structural correctness of the program. This proof is not only an assurance that the program is correctly structured, but is also a useful teaching aid in that it gently prompts the user to use the correct constructions, pointing out the cause of the error and where it occurs. With this interactive assistance users un= familiar with structured programming can rapidly learn the rules.

In addition to the control structure indenting there is another aspect of program layout which is of importance in real time programming. Programs which execute cyclically nearly always require an initialization section where control loop variables and items in common areas are given initial values. The static initialization performed by FORTRAN type DATA statements is only a partial solution as the initialization requirements can encompass all programming functions, including input/output operations and computations based or process variables. In a FORTRAN environment this function can be performed by using a flag in a common area for each program. This flag is tested in the program to enable a jump around the initialization section to be performed on subsequent cyclic executions of the program. In a real-time language oriented system this flag testing and setting should be provided in the language to enable this function to be implemented naturally. This is achieved in VIPER by providing a statement START which indicates the end of the initialization section and the start of the repetitively executed code. The initialization code is indented to distinguish it from the body of the program. Examples of the use of the facility can be found in nearly every program of the case study listed in Appendix B.2 as well as in table 4.1.

4.2 INTERACTIVE OPERATIONS

The term "interactive" has acquired a variety of meanings in computer applications. Two basic divisions which can be identified are:

1. Interactive program development.

2. Interactive dialogue in an applications environment (e.g. data-base management and information systems).

The send category is important in process control applications as part of the interface between the computer system and the process engineers and operators, but it is the first category which is of primary concern to this thesis. Similar ergonomic principles apply to both divisions (PALME, 1976) and in the development of interactive dialogue systems using interactive programming systems, GAINES (1975, 1976) has shown that the two topics can be closely related.

Even the term interactive program development is not well-defined. It is used by some authors to mean time-sharing type computing services (ARDEN, 1975) and by others to mean incremental compilation and direct execution such as is possible with BASIC (BERCHE, 1976; CHU, 1976; GAINES, 1975; HILDEN, 1976; WILCOX, 1976). Another context in which the term interactive is used is in mini-computer operating systems where the user drives the system directly from a keyboard to edit, compile, load and test programs in a rapid development cycle. The term interactive arises from the fact that on modern disc-based operating systems these operations can be performed in one or two minutes as opposed to 15 to 30 minutes on older magnetic tapes or paper tape oriented operating systems. Although a great improvement on past systems, this type of operation is not considered interactive in the context of this thesis.

Although the primary aim of VIPER is to provide excellent program development tools in a real-time interactive multiprogramming environment, the provision of dialogue facilities which can be used by process engineers and operators is also an important property. No explicit process dialogue functions are provided in VIPER, however, and the facilities which exist arise from the generalised interactive programming and debugging operations.

The ...../4.8

The interactive facilities which are provided in VIPER fall into four interrelated and overlapping categories.

1. Symbolic debugging of programs on-line and in real-time.

2. Monitoring of on-line real-time programs; examination of plant variables and perturbation of outputs.

3. Creation of new programs and editing of old program.

4. Testing the modules of a task as they are developed. (Top-down design and step-wise refinement.)

Only two functions need to be implemented to enable these facilities to be provided:

1. The ability to add (or delete) a statement to a procedure at any time whether it is executing or dormant.

2. The unification of the command and programming languages.

These functions unify the language elements, the debugging and monitoring commands and the file manipulation commands into a single coherent set with a common syntax and enable the interactive mode of operation to remain active on executing tasks. The operation of a process can therefore be dynamically monitored and symbolically debugged using the same command and programming language that is used to write the program. In PROSIC, the monoprogrammed predecessor of VIPER, the essential simplicity and naturalness of this on-line real-time debugging and monitoring facility proved to be an extremely powerful tool which was readily accepted by the process oriented users. To enable these facilities to be extended to VIPER, however, the properties of SVMM are essential, as this level of interaction could not otherwise be supported in a multi-user multi-tasking environment.

4.2.1 ...../4.9

4.2.1 Debugging

"Probably the most overlooked area of programming from the point of view of development and system effort spent versus computer and programming time involved, is debugging."

(GLASS, 1968)

"It is now common practice to use a high-level language for develop= ment of both systems and applications software, even on small computers. However, it is unfortunately true that while compilers abound the same cannot be said of good runtime diagnostic and debugging aids."

(PIERCE, 1974)

"Program debugging can often be the most tiresome, expensive and unpredictable phase of program development ...even the best-designed and best-documented programs will contain errors and inadequacies which the computer itself will help to eliminate. A good programming language will give maximum assistance in this."

(HOARE, 1975)

These three comments together with the perspicuous comments by WILKES (1976) quoted in section 1.2.2 emphasise the importance of the program debugging and the extent to which it has been neglected.

There are four basic functions of any debugging operation:

1. Examination of the process state i.e. display of current values of local and global data items.

2. Insertion of breakpoints: A breakpoint is a point up to which a program executes before passing control to the system with a suitable message to indicate that a breakpoint has been reached, together with an indication of which breakpoint has been hit.

3. Selective execution of blocks of code (usually coupled with 2).

4. Insertion of new code either to assist with the debugging or to fix any bug which has been found.

A ...../4.10

A typical debugging session consists of the interactive application of above four functions to trace, detect, locate and fix errors in the code.

In the majority of operating systems, and even on small stand alone minicomputer systems, a variety of facilities are provided for performing the above operations in machine level terms: to determine the state of a variable for example, a memory location is examined; to insert a breakpoint, a trap or jump is inserted at the required memory location; execution of a code sequence is performed with a simple jump to the start of the code with a breakpoint at the end of it; patching of new code is permitted by the ability to alter memory locations (i.e. machine code patch).

On a minicomputer these operations can usually be performed interactively, but on larger systems they are often severely re= stricted and can only be used in a batch mode. The examination function, for example, typically consists only of a dump of the entire memory space of the process.

The implementation of these debugging aids in machine level terms is adequate for assembler programming (which is what they are intended for) but is totally inadequate for the debugging of high level language modules which are written by application programmers. Without other help, these (and many other) programmers are reduced to using WRITE statements imbedded in the code to examine variables at various points. The frustrations and inadequacies of this procedure for debugging real-time software was noted in section 2.2.

In addition to the obvious disadvantages of such techniques I have encountered at least one situation where even as crude a tool as a WRITE statement could not be used. This pathological case is worth documenting as it illustrates the dilemnas which frustrate users in their debugging operations.

A pathological debugging problem

The problem occurred in the course of using the Hewlett Packard RTE-2 Executive on the Huletts Refinery Project. (This project is described as the case study.) In RTE-2 the memory is divided into two partitions, foreground and background with other memory areas being reserved for system operations, (in addition to the resident operating system). In the configuration used for the project the maximum size of the foreground partition was 6K words out of a total of 32K. This size was adequate for nearly all the control programs, provided they did not contain any formatted input-output statements, as the formatter routines immediately increase the size of a program by 3K words. Many of the programs could therefore no longer run in the foreground partition if WRITE statements were added. As a background program was not permitted to write into foreground COMMON, a program could not be temporarily debugged in the background partition. Nor could the system supplied assembler debug routines be used as they applied only to background programs which did not reference COMMON at all. The only solution to the dilemma was to tempo= rarily place certain variables in COMMON and to provide special message functions which could pass a few integer values from the program in question to another program from where they could be printed.

As if program debugging is not difficult enough as it is!

The object of high level, user oriented debugging systems is therefore to avoid the use of machine level concepts and to apply the four debugging operations listed above directly to high level language modules. Debugging systems which operate in this way are frequently called symbolic debugging systems. The basic requirements for symbolic debugging are runtime access to the symbol table of a procedure and the ability to associate statement line numbers with memory locations at run time. In compiler based systems this requires passing information from both the compiler and link-loading stages through to the debugging package.

Systems ...../4.12

Systems which use symbolic debugging techniques have been described by DANIERI (1976), DASAI (1977), GLASS (1968), GOULD (1977), ITOH (1973) and PIERCE (1974). In all the systems which they describe, however, the debugging operation must be decided upon before the program is compiled and run and even then only in some cases (PIERCE, 1974; DASAI, 1977) are the debugging commands interactive in the sense that they can be turned on or off during the execution of the program. In only one instance are the debugging commands closely related to the programming language; PIERCE (1974) uses a subset of CORAL for the debugging process. These systems are, however, a considerable improvement on the machine level debugging which must otherwise be used.

The size of the debugging system or package is also of particular importance. The very powerful PL/I checkout compiler (CUFF, 1972) for example, requires several hundred kilobytes. Even a compact "interpreter emphasising debugging capability" GLASS (1968) uses 50K words and the system described by PIERCE (1974) which uses a "greatly restricted subset of CORAL" requires 3K words for the debugging section. In VIPER, on the other hand, where the total executive occupies only 13K words, all the debugging facilities are estimated to occupy only a few hundred words. (An exact estimate is difficult to obtain because the facility is closely related and in= tegrated with the normal mode of operation.) In the earlier mono= programmed PROSIC (HEHER, 1976a) it took less than 150 lines of assembler code to provide similar facilities.

The simplicity, economy and versatility of the debugging facilities in VIPER results from four factors.

1. The symbol table is always available as it must be retained to permit programs to be backlisted (decompiled).

2. Associating a trap or other debug operation with a source statement line number is straightforward because the line numbers are also stored in memory with the program code.

3. The unified command and programming languages.

4. ...../4.13

4. The ability to enter a statement into a procedure at any time whether it is executing or dormant.

The use of the same language for programming and debugging, and the unification of the command and programming languages can therefore be regarded as an essential feature of a software system for a small computer and not as an expensive luxury. The savings in code which result from using a common command and language processor have also been noted in an implementation of POP-2 (BURSTALL, 1971).

As an example of a debugging operation in VIPER consider the use of a simple PRINT statement to monitor the operation of a re= petitive real-time task. The statement can be issued either as a command to examine the current value of any variable known to the procedure, or as a statement which is entered on-line into the procedure at a specified position. The procedure may be executing or dormant, memory-resident or bulk-storage resident. (The SVMM will perform the necessary seek and swapping-in in the latter case.) By adding and deleting PRINT statements within the procedure as it is executing, the program flow can be traced dynamically using what is in effect a software probe which selectively displays the required data at any point in the procedure. This procedure is considerably more flexible and general and easier to use than the shotgun "trace" command which has been implemented in many debugging systems (e.g. GLASS, 1968). (A trace operation was tried in VIPER and was rapidly discarded as being far too unweildy.)

Any legal statement can be used as a probe, or any sequence of statements. (A little care must of course be exercised when using structured statements which are always paired e.g. FOR-NEXT.) As another example, consider the use of some sequence of statements which constitute some debug or monitoring operation, such as printing a table or checking a table for consistency. If these statements were coded as a subroutine, called SUBX for example, they could be invoked directly with a command

CALL SUBX (<parameter list>)

or inserted at any place, or at any number of places in the executing procedure by

<line no> CALL SUBX (<parameter list>)

The parameter list is optional, and if it was too cumbersome the necessary data required by the debugging subroutine could be temporarily placed in a shared (common) data segment. When the debugging operation is complete both SUBX and the data segment can be deleted.

Example

The subroutine MESSAGE in the Case Study (page B2.17), has a local array PM which contains a record of the previous messages that have occurred in the applications software. This array need normally only be known locally to MESSAGE, but if a record was required of these previous messages, a call to a subroutine executed as a command, thus

CALL PRINT.PM (PM,CPM)

within the context of MESSAGE (which could have been established with a DEBUG MESSAGE command) would permit this array to be printed out. This ability to examine the interior data structures of procedures is a unique property of SVMM.

The interactive mode of operation together with the SVMM permits the entire language to be used as an extended set of debugging facilities which can be applied to any segment which is known to the system.

4.2.2 Monitoring

Closely related to the debugging mode of operation is the monitoring of values of variables in the plant data base. In addition to the direct readings which are obtained from plant instruments and trans= ducers, there are usually a number of derived variables which contain information which is of interest to operating staff. A selection of these variables is usually placed in a particular common area and made available for examination by means of special keyboard or display devices. These specialised display devices and their associated software are an expensive component, however, and may not be justified in small or experimental installations. In VIPER, by using the flexible interactive commands and the shared data areas (if necessary) the value of any variable in the system can be quickly and simply displayed. While not intended as a substitute for process operators' display pannels, the facility is an invaluable aid to the process engineer who invariably needs more data and information than the process operator, particularly when investigating a particular process problem or proposed change in processing strategy. The facility can also be used in the design phase by helping to determine what facilities are required in any proposed hardware display panels. In VIPER a restricted subset of the debug-mode-operations has been provided which has special access attributes tailored for these monitoring operations - as described in section 4.3.2.

Another aspect of monitoring is the direct measurement or adjustment of process input and output devides. In the case study for example the routines CDAC (Control Digital Analog Converter) and WCOUT (Write Contact Output) are used to output control values to particular devices, appearing in the form -

<line no> CALL CDAC (CHAN,VOLTS)

or

<line no> CALL WCOUT (CHAN,STATUS) (STATUS=0 or 1)

and which will write a voltage or set a contact respectively on the specific channel.

The same statements can be used as commands, however, by ommitting the line numbers, and will then directly perturb the value of the designated channel. Together with others, commands of this form constitute a direct method of monitoring and commissioning plant instrumentation on-line with a minimum of disturbance to the system. Used incorrectly, these output commands could of course cause unwanted disturbances. In VIPER this is prevented by permitting a password to be associated with the commands which can be used to prohibit access to all but authorised users.

4.2.3 Text creation and editing

The methods whereby new program text is created were described in sections 2.2 and 2.3 and illustrated in tables 2.1 and 2.2. Line numbers from the basis of editing operations. It has been pointed out that in a structured language line numbers are not strictly necessary (CHU, 1976; LAWRENCE, 1975). In VIPER the only statement which requires a label is the GOTO, which is seldom used in any event, as was noted in section 4.1.2. If a label (possibly non numeric) was provided for the target of a GOTO, no line numbers would be required from a structural point of view. Although super= ficially minor there is in fact a profound difference in operating philosophy between line numbered and non-line numbered systems.
In my experience, editing operations are significantly easier and the overall operating commands simpler when line numbers are used. There are also good reasons for retaining line numbers for labels if labels are required. A GOTO is an undisciplined transfer of control which can go anywhere; but if the target is a sequentially numbered line identifier, it is far easier quickly to follow the program flow, particularly when working with a limited display of text on a CRT screen. GAINES (1976) has emphasised this latter point and has stressed the desirability of using line numbers in interactive systems.

4.2.4 Module testing

One of the recommended practices associated with the art of structured programming, is the independent testing of individual modules of a task as they are written. Some sophisticated software tools have been developed for this type of operation (e.g. CUNNINGHAM, 1976; HENDERSON, 1974) particularly when top-down design or stepwise refine= ment strategies are being used. VIPER makes no specific provision

for ...../4.17

for this design procedure but the ease with which modules can be individually tested, together with the flexible data structures which simplify the generation and linking of test data, enables this practice to be carried out using the standard interactive facilities. Of more importance than a formal design procedure, (which is possibly of relevance only to large software problems which would most probably not be coded in VIPER anyway) is the informal flexibility of being able to test and examine the operation of a procedure in a variety of ways before it is finally integrated into an overall task.

This type of testing was used extensively in the development of the software for the case study. All these programs were entered and tested in Pretoria before being used in the factory in Durban. This required numerous test programs to provide dummy inputs, outputs and simulated process data to enable both the scan and control programs to be exercised.

4.3 PROTECTION AND ERROR CONTROL

The most important property of the protection facilities is that they are applied to executable code (and data) segments and remain in force on active tasks. The ability of users to modify procedures, access data areas or execute tasks can therefore be controlled dynamically. The application of file-system-like protection facilities to active segments in the system is a unique property of SVMM.

The protection mechanisms have two goals - the first is to provide facilities which are easy to use and the second is to ensure that they are impossible to circumvent. These two goals conflict at times so that in practice a modicum of effort must be expended to achieve the highest level of protection; on the other hand good protection facilities are always applied by default without any explicit user action.

There are three aspects of protection and error handling which are of importance in VIPER:

1. The inherent protection provided by the interpreter.

2. Explicit protection provided by the SVMM structures.

3. Error control and recovery.

4.3.1 Inherent protection

The protection facilities which are usually provided in most interpretive systems are as follows:

1. Detection of undefined variables.

2. Array bounds checking.

3. Subroutine call parameter list matching (number of parameters only).

Checking of arithmetic operations for underflow, overflow and other illegal states is also usually performed, which, although not strictly a protection operation, is a useful monitoring function.

Despite the limitation of these three facilities they do perform a useful service which can save a great deal of time during program debugging. A short example may help to illustrate this point.

During the commissioning of the FORTRAN version of one of the control programs of the Casy Study, it was observed that the program sometimes malfunctioned during override conditions. The fault had appeared only three times in 6 weeks of continuous running. Attempts to trace the source of the error required that the program be re=compiled and loaded with debugging statements added, but each time this was done, the fault cleared itself. The error was eventually traced to an undefined variable; the random number that resulted sometimes being within a suitable range so as not to cause an error, and which always ended up being reset (cleared) when the program was reloaded. An interpretive system would have pinpointed the exact line and variable which caused the fault on the very first execution of the override condition.

A compiler which notes variables which have not been assigned values would have helped in this case, but this is not always possible as a variable may be assigned a value on one path through a program and not in another.

The point to be noted in connection with this example is not the length of time that it took to locate the error, nor that the error was eventually found, but the fact that other errors of this type may exist in programs which could go undected for long periods of time (perhaps forever) and yet still be causing a program to compute incorrectly some of the time.

Array bounds checking is also an important protection function as it ensures that neither code nor data can be overwritten. Un=fortunately the checks are sometimes bypassed once an array is passed as a parameter to a subroutine. This is particularly un=desirable property, as errors which are propogated across module boundaries are always more difficult to detect. The comment made above in connection with undefined variables also applies here: that the serious problem is not so much the occurrence of the error but the possibility that it may go undetected. This is a particular possibility when another data area is overwritten, but can occur even when code is damaged.

The time consumed by these run-time checks has been criticised. The use of a check-out or debugging compiler has been suggested which introduces overhead only while testing; the debug or checking code being removed in the production version of the software*. Alternative methods of reducing the run-time overheads are possible (e.g. BROWN, 1976(c)), but additional work is required in this area. In VIPER

where ...../4.20

*This procedure has been likend to wearing life-jackets while practising on dry land and then taking them off when going to sea.

where run-time overhead is not of particular concern, a check is always made for undefined variables and for array bounds overflow.

The testing of subroutine parameter strings for matching lengths is of limited usefulness, and far more rigorous checking is required here in order to produce reliable software. The facilities provided in VIPER for testing this interface were described in section 3.2.4.

4.3.2 Explicit protection

The explicit protection functions provided in VIPER can be divided into two classes:

1. Segment access, including the control of source text modifications.

2. The protection of shared and local data areas and of parameter passing.

Similar mechanisms are used for both classes, but the environ= ments in which protection is applied are different.

4.3.2.1 Procedure segment access

The basic means of controlling access to procedure segments is by using a password. Before any input is accepted from a user at a key= board he must LOGON with an appropriate password. (The LOGON command is also used by the system manager - known as the MASTER - to introduce new users. These functions are described in Appendix A2).

A password is not necessarily associated only with a particular user. Its primary function is to logically partition tasks into sets of co-operating procedures. The set of procedures and their associated data elements controlling a particular section of a plant, for example, could be associated with a particular password, while the modules of an operator interface could be given another. In this context the LOGON command identifies a logical subset of procedures which the user wishes to access. It also serves the usual protection function,

however, .../4.21

however, in that if the appropriate password is not specified, no modifications can be made to a procedure.

There are seven access states and substates of procedures for which provision has been made:

CHANGE		- Password holder only
DEBUG		- Password holder only
MONITOR	- Free	- Default mode
	- Password	- Substate specified by ACCESS command
EXECUTE	- Free	
	- Password	
	- None	- No access

CHANGE, DEBUG and Free-MONITOR modes are entered by typing the name as a command, e.g.

CHANGE <procedure name>

whereas entry into the substates of EXECUTE and Password-MONITOR is controlled by ACCESS commands. If the input is already associated with a particular procedure the procedure name can be omitted. To move from DEBUG to CHANGE mode, for example, within the same procedure, the command CHANGE on its own is sufficient. The states DEBUG and CHANGE are available only to password holders, provided that password has been validated for these modes. A password has attributes associated with it which can restrict the states which a user is allowed to enter. The substates of EXECUTE and MONITOR may permit non-password holders to perform an operation but the state can only be changed by the password holder.

1. CHANGE

In this mode any alteration can be made to a program, even if the program is executing. It is the basic mode used for editing programs and with a little care is also useful as a debugging mode in that permanent changes to the program can be made immediately.

2. DEBUG

This mode possesses a restricted set of the CHANGE mode access rights. The procedure can be listed, variables examined and breakpoints and statements inserted, but no existing statements can be deleted or modified. Statements which are added while in this mode can later be deleted, however, as they are flagged as temporary DEBUG statements. Provision had been made to automatically delete all debug statements once the mode is excited but this has not been implemented in VIPER. In the earlier monogrammed PROSIC it had been found that owing to the size of the programs (300 - 500 lines), debug statements could be inadvently left in a program. In the modular VIPER, however, where the average procedure is much shorter (34 lines in the Case Study) this problem has not occurred. A simple alternative would be merely to flag any debug statements in the listing of a procedure.

A very useful function which is available in the debug mode is a statement execution frequency count. This counts the number of times that each statement in a procedure has executed and displays the current number when the procedure is listed - as illustrated in table 4.2. KNUTH (1971) has stressed the importance of execution counts and has advocated their use in all software systems. They are an invaluable aid in determining the most frequently used parts of a program, and can in addition be used to determine which statements have never been executed. The simplicity and economy of this feature in VIPER - it takes only about 75 lines of code to implement - illustrates the versatility of an interpretive system.

3. MONITOR

This mode permits the state of a procedure to be examined using commands such as PRINT and LIST, but no statements can be added or changed. This restriction ensures that nothing can be done which interferes with the execution of a procedure and this mode can therefore be made freely available for process staff to use.

...../4.23

use. In view of the general goal of the SVMM to enable users to co-operate, the default state of MONITOR is free, i.e. any user can look at a segment which is in 'free monitor' mode. If it is in the state 'password-monitor', then only a password holder can perform monitor functions. The state of a procedure can of course only be changed by a password holder. The sub-state 'password-monitor' is specified with an access command, as shown below.

4. EXECUTE

The access attribute EXECUTE can be in one of three states: free execute, password execute and no access. The latter category ensures that a program is locked out and cannot be executed by any user. The default state here is password execute, i.e. only a password holder can invoke a procedure unless the owner specifically decides to make it freely available.

The state required is specified by an access command:

ACCESS (<procedure name>) = <attribute>

The procedure name can be ommitted if the current procedure is intended. The attribute is a three bit operator which has a numerical value of 0 to 7:

0 - No access
1 - Password execute
2 - Free execute
4 - Password monitor

Symbolic, instead of numeric, attributes could be provided as is done for data segment access. (The data segment access statement is of the form: ACCESS (<data element name>) = READA/WRITEA where READA and WRITEA are symbolic attributes.) Symbolic execute attributes have not been provided in VIPER as the numerical values are considered adequate. It has been found in practice that these substates are not used frequently in the

direct ...../4.24

direct applications software, i.e. the software used by the process staff. They are, however, useful for controlling access to software modules which are used for system housekeeping and management tasks. The numerical equivalents are also used for display purposes as the access attributes can be used in arithmetic expressions e.g.

```
X  =  ACCESS (<proc name>)
PRINT X
```

or even more directly

```
PRINT ACCESS (<proc name>)
```

From these access states and the defaults that are used, it is evident that users are generally unaffected by the password constraints unless they wish to modify or execute another user's procedures or permit a user to access their procedures.

4.3.2.2 Data Access

There are two different aspects of data accessing. The first is related to specifying the access attribute of a shared data segment i.e. who can access that segment; the second to the individual access states of data items which may be either local array variables, elements of a shared data segment or formal parameters. Tables 2.2 and 3.2 have illustrated operations of the second type.

The object of protecting shared data segments is to limit access to those procedures which need to reference the data, granting only sufficient rights to permit the required operation. The most general method of performing this access control is to associate a capability list with each data area which specifies the individual rights of each accessing procedure. No other procedures would then be allowed to access the segment. The skeleton of such a capability list exists in the procedure reference descriptions that are necessary on the data segment descriptor table for linking purposes. (Fig. 3.9.)

In ...../4.25

In reviewing the requirements of process control systems in general, and of the Case Study in particular, it was, however, felt that this generalised procedure could be unnecessarily complex and that simpler mechanism would give adequate protection. This works as follows:

A shared data segment always has a password associated with it. Originally this is the same as the password of the procedure from which it was created but this can be changed. The segment can then be in one of two modes, password protected or public access. If it is password protected only procedures with a matching password can access it, both read and write operations from other segments being prohibited. A public segment on the other hand is not password protected and is freely accessible to be read by anyone, with the read only attribute being granted by default. To write into a public segment, a procedure segment must specifically request access to either a particular element or to all elements.

To continue to provide a measure of protection to these public segments, however, it was decided that only procedures with a matching password would be granted write access. In problems with complex data structures which are shared between disparate tasks which do not have the same password, this restriction may lead to cumbersome use of artificial passwords. This restricted access algorithm was adequate for the tasks envisaged for VIPER, however, and was attractive to use because of the simplicity of the commands required to implement it. Complex commands are likely to discourage the use of the protection facilities altogether, a point which has been emphasised by PALME (1976).

In the spirit of VIPER, which is to promote co-operation rather than to discourage it, the default attributes of shared data segments are public access, read-only. If password protection is required it must be specifically requested with a command of the form.

ACCESS (<data segment name>) = 4

Only the password holder can issue the command.

The other form of the access command

ACCESS (<data item name>) = 0/READA/WRITEA/READA+WRITEA

are used to set the access from within a procedure to either a data segment or a particular data item. Examples of operations of this type are to be found in Table 2.2 and in many of the Case Study programs (Appendix B).

The access attributes READA and WRITEA have numeric values, as in the case of procedure segment access. The numeric equivalent of the access command above is

ACCESS (<data item>) = 0/1/2/3

and the current access state of either a segment or a particular element can be determined with display commands such as

PRINT ACCESS (<data item name>)

where the value returned is between 0 and 3 :

0 = no access
1 = read access
2 = write access
3 = read and write access.

4.3.3 Error control

There are three types of errors to which attention must be given in an operating system:

1. Expected errors

   These can result from certain commands e.g. RUN <prog name> where it is known that there is a possibility that the name may not exist or that it may be in an illegal state (e.g. already running).

2. Unexpected errors

   These usually, but not necessarily, indicate either a logic or coding error, or a hardware error.

3. ...../4.27

3. Errors originating from within the operating system itself.

It is generally accepted that a programming system must provide orderly control of the first type of errors within the programming language. A particular approach has been recommended for real time BASIC systems (PURDUE, 1975) which has been implemented in at least two systems (KOPETZ, 1976; BIANCHI, 1976). The action to be taken following the occurance of errors of the other two types is a subject of debate (KOPETZ, 1975; GOODENOUGH, 1975; POPEK, 1977) and there would appear to be no consensus on the action which should be taken in these situations. The basic point of divergence is whether automatic recovery from type 2 and 3 errors should be attempted or whether the task or system in which the error originated should be halted until the error is either fixed or converted to a type 1 error.

4.3.3.1 Expected errors

If no action is taken to detect an error the standard procedure is to print a diagnostic message on a logging device and then halt the procedure or task where the error originated to prevent it from executing further. To permit a task to perform its own error handling, some mechanism must therefore be provided for inhibiting the transfer to the normal system error handler and forcing a transfer to a user supplied code sequence. This trapping operation can be performed either locally or globally. Table 4.3 illustrates these two different types, the first example is from the Hewlett Packard RTE FORTRAN and the second is the recommended approach in real time BASIC (PURDUE, 1975; ESONE, 1977).

In VIPER the global RTE-B approach was adopted although implemented some what differently to avoid the use of an instructured GOTO. The statements ERROR-ERETURN are provided as a structured pair which can be unbedded anywhere in a procedure (but usually either within the initialization section or at the end of the procedure). Table 4.4 illustrates the use of these statements. From the example it can be seen that although these facilities do provide the necessary control, they are somewhat clumsy to use. It is also not clear whether they are adequate in a structured programming environment where it may be necessary to report errors back up to higher level module. This is a subject which requires further investigation and development.

4.3.3.2 Unexpected errors

KOPETZ (1975) has argued for the systematic handling and attempt at recovery from even unexpected errors such as arithmetic underflow and overflow, divide checks and certain hardware errors. In the discussion which followed his paper however, it was clear that there is no consensus on this point and that many workers in the field are of the opinion that no automatic recovery should be attempted in these situations. In the design of the language EUCLID, POPEK et al (1977) for example, have noted that "we know of no efficient general mechanisms by which software can recover from unanticipated failures of current hardware. Anticipated conditions can be dealt with using the normal constructs of the language; most proposals for providing special mechanisms for exception handling would add considerable complexity to the language". The occurence of the error should be clearly noted of course, and every assistance should be given to the programmer to assist him in determining the location and cause of the error.

In my own experience there is a real danger, if the first "KOPETZ" approach is adopted, that the error handling code can become as complex, as the original programming. This additional code not only adds to the cost of software, but is in itself a possible source of error; adding the additional complication of handling errors within error handling code. In considering the actual process control software with which I have worked it is difficult to see what this unexpected error handling could hope to achieve. More fundamentally, and far more serious there would appear to be a definite possibility that attempts at automatic recovery would allow (or force) a task to continue which was executing incorrectly. In a process control environment it would appear better to stop the task and notify the operator to allow him to implement appropriate back-up procedures.

VIPER ...../4.29

VIPER is therefore a supporter of the second approach where any error which is not expected is logged, with the name of the procedure and the line number where the error occurred indicated. The offending procedure is removed from the ready list and flagged as containing an error to prevent repeated execution (and repeated printout) in case the procedure is part of a task which is running periodically*.

4.3.3.3 System errors

An operating system should operate without errors, but this is seldom achieved in practice. The two approaches outlined above can be taken here also, i.e. error recovery and error abort. Error recovery systems are of value particularly in large complex operating systems which consist of many independant modules, or which use a kernel approach. As VIPER is a relatively small system which does not have a kernel and which is entirely memory resident, the second approach was adopted, i.e. the system is halted on the occurrence of the error.

Every effort must therefore be made to locate and fix any errors which do occur and the system itself should assist in the earliest possible detection of any errors, particularly when the system is being developed. The time and space overheads of vigorous self-testing and checking are of little consequence at this stage and it has been found that these tests can locate incipient errors which may otherwise only manifest themselves at a later stage.

In VIPER for example, the double-linked lists that are used for both the physical and logical structures, and the very well-defined structure of each segment, permit regorous tests of the structural integrity of the system to be performed. These checks are always performed, for example, when the structure has been altered in any way, and are invaluable in preventing an error from propagating its ill effects before being detected.

There ...../4.30

*This algorithm may also be said to work on the assumption that it is less embarrassing to have a task stop at midnight than it is to have the computer room knee-deep in paper in the morning. The former has been known to pass unnoticed, the latter, never!

There are good opportunities for error recovery in the SVMM in that if any one pointer is found to be in error, it can be corrected owing to the double-linked nature of all lists. In VIPER however, the redundant information is used for assertion checking in a manner analogous to that recommended by RAMAMOORTHY (1975) and POPEK (1977). At various points in the executive (particularly at points where the structure has been modified) it is asserted that a given structure or set of relationships exists. By verifying that the assertion is correct, the computation can be allowed to proceed with a high degree of confidence that the preceding computation was performed correctly. In the development of the SVMM system these assertion checks have proved to be an invaluable debugging aid and they are considered to be a vital element of the error-detection features of the executive.

4.4 SYNCRONIZATION

The semaphone principle developed by DIJKSTRA (1968) is the basic building block for the synchronization of processes and the control of access to shared data. It is, however, an awkward element to use in real-time programming for several reasons (KYLSTRA, 1977).

1. If a lock (wait) operation is encountered in the program text it is not immediately clear whether or not it is an entry to a critical section (in which case it should be followed by a free (signal) operation further on).

2. If it is the entry to a critical section it may not be immediately obvious from the text what the shared variables are.

3. It is difficult to check whether all critical sections are properly protected by a semaphone.

4. It is difficult to check for the possibility of deadlock.

For these reasons other language constructs have been proposed such as the "REGION" construct (HANSEN, 1973) the "MONITOR" concept

(HOARE, .../4.31

(HOARE, 1974) and "KNOWS" clauses (GORD, 1976). These facilities can be implemented with simple semaphones or with more general constructs such as those proposed by SCHROTT (1976) or RADUE (1975).

HOARE's monitor concept has been noted to be one of the most general and secure structures, but it would appear to be more suitable for operating system construction than for an application oriented software system like VIPER. Reviewing the synchronization and protection requirements of such systems, the "REGION" construct was selected as the one which appeared most natural for use with the shared data segments which are used so extensively in VIPER. This operates as follows:

Given a shared data area which is declared with a statement

COMMON <com name>, <data list>

a critical region where mutually exclusive operations are required is defined by:

```
REGION <com name>
      <critical region statements>
END REGION <com name>
```

Two or more procedures declaring an area in this way are guaranteed to be mutually exclusive in the critical region. The REGION statement sets a semaphone associated with the data area and can only proceed to execute the critical region statements if the semaphone is not already locked. If the semaphone is locked the procedure is suspended and waits for the semaphone to be cleared (unlocked) by an END REGION statement.

The use of a REGION-ENDREGION pair ensures that the operating system can check that no area is inadvently left locked. The indenting that is performed between the pair also ensures that the region which is critical is immediately apparant. Examples of the use of the REGION - ENDREGION construction are given in Table 2.2 and in a number of the programs of the case study, Appendix B.2 pages B2.5, B2.14, B2.20 and B2.21.

Other ...../4.32

Other syncronization operations are occasionally required which do not fit naturally within the region construct. Two operations

LOCK <com name>

FREE <com name>

are therefore provided for these purposes. One use of these statements, for example, is during interactive operations. If a data structure was to be examined using the on-line interactive DEBUG or MONITOR operations it may be desirable to prohibit modification of the data while the debug operations was in progress. Typing the command

LOCK <com name>

would then set (lock) the semaphone associated with the data area <com name> and prevent any procedure from entering a corresponding critical section defined by the REGION ENDREGION statements. When the debugging operations were complete, the data area could be released with the command

FREE <com name>

Any task which had been suspended waiting to enter the critical region would then be reactivated to continue processing.

These simple but powerful facilities assist in the modular decomposition of tasks into separate and independant sub-tasks which are much simpler to code and debug. A particularly good example of this is to found in the case study where the FORTRAN program SERVO was decomposed into the three tasks SERVOTIP, SERVO.HOUR and SERVO.8.HOUR. (These programs monitor and record the operation of a servo-balance scale unit which weighs the raw sugar entering the refinery). Not only are the VIPER programs easier to write, read and debug, but they require only 760 words to be used routinely in memory on each tip of the scale versus 5328 in the FORTRAN version. (Table 6.1).

4.5 DOCUMENTATION

The importance of good documentation in programming systems has been stressed by many workers in a range of programming areas, from commercial applications to real-time systems programming. (DE BALBINE, 1975; GILB, 1975; HOLT, 1975; KERNIGHAN, 1973, 1977; McMONIGALL, 1974; NEELY, 1976; NEWMAN, 1974; OSTERWELL, 1976; SCOWEN, 1974). The purpose of documentation is to allow programs to be read and understood both by their original implementors and by others, because real programs have been noted to be subject to a continual flow of changes and improvements over their lifetime.

This is particularly true of process control systems where changes in process operating conditions or strategy can frequently require changes in associated software over a life of five to twenty years. Considering the documentation requirements of VIPER, it is apparent that they are even more rigorous because VIPER is designed particularly for experimental or investigatory work, an environment where the maintenance of good documentation is as difficult as it is important.

An additional factor militating against good program documentation in VIPER is its interpretive nature. Because of the incremental compilation into internal meta-code, source text is never stored and text layout to improve program visibility cannot be used as it can with compiler oriented languages. BASIC, on which VIPER is based, is also notoriously difficult to document and read because of the clumsy comment facilities and lack of syntactic structure. (The only thing worse than BASIC is APL which has been strongly criticised, KERNIGHAN, 1973; DIJKSTRA, 1972.) Special effort must therefore be made to assist and encourage the documentation of interpretive programs.

A second aspect of documentation which is of importance, particularly in real-time systems, is the documentation of the overall structure of a task. This is concerned with the relationships between programs and the hierarchy of programs and data structures which constitute a task. This aspect has been termed system documentation as apposed to program documentation which was commented on above.

4.5.1 ...../4.34

4.5.1 Program documentation

There are two aspects of program documentation which contribute to the clarity of program code:

1. Language structure.

2. Comment facilities.

4.5.1.1 Language structure

A structured language is one of the most important aids to program documentation and is absolutely essential to enable interpretive systems to back list (decompile) a program in an intelligible format. This aspect was commented on in section 4.1.2 and an example of the VIPER facilities given in Table 4.1. There is a strong case for all interpretive systems which perform the backlisting of programs to use structured languages, for the sake of documentation if nothing else.

A second aspect of language structure is related to variable and procedure naming conventions. The restrictions in BASIC (a letter and a digit for simple variables and a letter only for array variables) are atrocious and quite unnecessary, as an extension of PROSIC has shown (HEHER, 1976 (b)). In VIPER, all names, including variables, data areas and procedure names can be up to 16 characters in length. (This length restriction is arbitrary and arose purely out of the desire to pack additional information in the 16 bit de= scription head, as shown in Figs. 3.1 and 3.3.) These long names are an invaluable aid to clear documentation, as can be seen from the programs in Appendix B, and reduce the requirement for trivial comments to explain the meaning of variables. The increase in the size of the symbol table as a result of the longer names is of minor consequence compared with the benefits accruing from their use. (In the case study it is estimated that using only short one or two letter names would save approximately 10% in the total space required by the programs.)

Another ...../4.35

Another aspect of language structure which has invited comment is that of conciseness. (KERNIGHAN, 1973). FORTRAN, and to a lesser extent, BASIC, suffer from a lack of conciseness which results in program modules being physically larger than necessary. As the ease with which a program module can be understood is related to its size there is an incentive to allow more compact representations. (Conciseness, in the dictionary sense of "short and clear", is not to be confused with the sententious contraction of a language like APL which can reduce a page of code to a single incomprehensible line.) Considering the structure of a large number of FORTRAN programs, KNUTH (1971) has shown that nearly 50% of the statements in typical programs are assignments, 60 to 70% of which are simple assignments with one argument. An experiment was therefore made in VIPER with providing multiple assignments on one line; numerous examples of which are to found in the programs in the case study. The average length of fourteen of these programs was measured to be 48 lines compared with 73 lines for their FORTRAN equivalents (comments excluded, see Table 6.1). A major portion of the contraction is attributable to the compound assignment statements.

As the assignment statement does not affect the program flow, this conciseness does not detract from program clarity. It is the control structures IF-FOR-CASE and the like which determine the flow and these are pivots on which the understanding of a program hinges; contracting the "straight-line" code enhances the lucidity of the control structures. The comment conventions adopted in VIPER which are discussed in the next paragraph also contribute to maintaining the conciseness of programs.

4.5.1.2 Comment facilities

The importance of comments in program documentation has been stressed by SCOWEN, (1974); KERNIGHAN (1973, 1977) and HOARE (1975). All languages make provision for comments in one form or another, but the point these authors make is that the actual syntactical forms used are of crucial importance. The ease with which comments can be inserted, and their readability once inserted, are an important factor in

determining ...../4.36

determining the extent to which the facilities will be used by programmers.

End-of-line comments are especially recommended as they are easily inserted, are directly associated with a line of code, and can be made highly visible. End-of-line comments were first tried out in PROSIC where they were combined with a horizontal tabulation facility to permit the construction of tabular comment areas. This achieved the first two goals above, but did not achieve a high degree of visibility. In VIPER with the longer assignment statement and the indenting, this visibility was likely to be even worse, so the horizontal tabulation was replaced by a simple right justification of all end-of-line comments. This appears to achieve the desired visibility without detracting from the ease of insertion. The right justification has been recommended by NEELY (1976) in a description of a structured FORTRAN preprocessor, but it should be noted that the right justification is tedious and difficult to achieve in a compiler oriented system. The line must first be typed, its length determined and then moved to the right with a text editor, an operation which destroys the essential simplicity of use. In VIPER the comment is inserted immediately after the last character of code, the start of the comment being demarcated by a control character. It is in the backlisting operation where the length of the comment can be determined apriori, that the right justification takes place. Table 4.1 illustrates this mode of operation.

One of the severe problems associated with commenting inter= pretive programs is that the comments remain in memory together with the code and therefore use memory space which would otherwise be available for code segments. As the comments in a well documented program may take nearly as much space as the code, this could double the swapping rate in a situation where all the segments cannot fit into memory. This is regrettable because the comment code is only required when the program is listed (decompiled), an event which occurs relatively infrequently. The knowledge of this space penalty would also deter the programmer from adding comments freely.

A ...../4.37

A simple and elegant solution is available using the SVMM facilities. The comments can be kept in a separate segment which could normally be resident on a bulk storage device and would only be swapped into memory for either listing or updating operations. Only a minimal space penalty would therefore be incurred in adding as many comments as were necessary. Fig. 7.1 outlines a structure in which this concept is incorporated.

(This facility has not, however, been implemented in VIPER because of the very small memory which was available for the initial development work on the case study programs. The code to handle this separate manipulation of comment segments was sketched out and was estimated to take 200 to 250 words which just could not be spared on the 16K computer that was in use at that time.)

4.5.2 System documentation

Typical real-time programming tasks are made up out of a number of independent modules which operate on one or more data bases. In maintaining and operating these systems it is important to understand the relationships between the various modules of the task, including information such as which modules call others (the hierarchial relation= ship) and which modules access particular data areas. The relation= ships amongst modules is of importance because the interface amongst them is known to be one of the most troublesome and error prone in real-time programming.

A number of software tools have been proposed and developed for the documentation and verification task (DE BALBINE, 1975; McMONIGALL, 1974; OSTERWELL, 1976; RYDER, 1974). The primary assumption of these documentation systems is that "the only precise and by definition up-to-date source of internal documentation for most software in existence today lies in the programs themselves" (DE BALBINE, 1975). The purpose of the system documentation exercise is therefore to extract from the source listing of the program one or more of the following items of information:

1. ...../4.38

1. A list of all main programs and the subroutines (modules) which they reference (applied recursively).

2. A list of all common data areas and the modules which reference them.

3. Checks and diagnostics on illegal references to common data areas (mismatched sizes or data types).

4. Checks and diagnostics on actual/formal parameter lists in= cluding verification of parameter type matching and illegal references.

5. Tests for undefined variable references; redefined variables without use; and illegal or dangerous type usage.

6. Cross reference lists of local and global variables and labels.

In all the systems mentioned in the literature, these functions are performed off-line by separate processing programs operating on the source listing of the task to be processed. They are typically very large programs, in the range 10 000 to 25 000 high level language statements, which illustrates the complexity of producing this information from source listings.

In VIPER items 3, 4 and 5 are tested dynamically at execution time (in addition to other checks and protection functions described earlier). Furthermore the information required for items 1, 2 and 6 is available and readily accessible within the descriptor tables of the segments.

Only one documentation module has been included in VIPER to date, but is provides a powerful means of analyzing the overall structure of the task. The output of this documentation aid for the programs of the case study is shown in Fig. 4.5.

For each module in the system the following information is provided:

1. Module name and the name of its current father, if any.

2. A list of all the external modules (subroutines and programs) to which reference is made. Each entry is also flagged (with a*) to indicate whether or not it is currently linked to this module.

3. A list of all the common data areas which are referenced, with a flag as above.

4. Schedule and status bit information which describes the current state of the program.

Each common data area is also listed together with information on its size and all modules which reference this area. Each module name entry on this list is also flagged as above if it is currently linked to the data area in question.

A list of all the assembly language subroutines which are available in the system can also be provided.

The important point about this information is that it is obtained dynamically on line and represents the actual state of the system at that moment.

The facility is invoked with a statement

CALL MAP (<param>)

which, as always, can be used either as a program statement or as a command. The parameter <param> is used as a qualifier to obtain partial listings of information:

param = 0 - list and map all modules
< 0 - status information only, no cross reference list
= PASSWORD (proc name)
- provide mapping and status information only for those modules which match the password of the specified procedure
(<proc name>) optional, if ommitted current assumed).

A cross reference list of local variables used in a procedure is not provided in VIPER, but could easily be implemented as the

information ...../4.40

information is readily available. In the case study, it was found that the relatively small size of the program modules made a cross reference virtually unnecessary. Any variable could be located by inspection within a short space of time.

TABLE 4.1 AN EXAMPLE OF THE STRUCTURING OPERATIONS

```
#PROC STRUCTURE.TEST
#1 PROC
#10 PRINT "SIMPLE STRUCTURE TEST"
#20 START END OF INITIALISATION CODE
#100 FOR I=1 TO 7 MAIN LOOP
#110 PRINT I,
#11_20 IF I<3 BINARY IF ON ITS OWN FOR VISIBILITY
#130 THEN PRINT "  I<3", THEN,ELSE AND UNARY IF CAN ONLY
#140 ELSE PRINT "  I>=3", BE FOLLOWED BY A NON-CONTROL
#150 IF I=4 PRINT "  I=4", STM ON THE SAME LINE
#160 ENDIF
#200 IF I>=5
#210 THEN THE FOLLOWING CONTROL STM MUST BE ON A NEW LINE
#220 FOR J=1 TO 4
#230 CASE J=1 OUTER CASE INDEX=J
#240 PRINT "  CASE J=1",
#250 CASE I=6 NESTED CASE INDEX=I
#260 PRINT "  CASE I=6",
#270 CASE I=7
#280 PRINT "  CASE I=7",
#290 ENDCASE I END OF INNER CASE
#300 CASE J>2 AND I>6 COMPOUND CASE CONDITION, INDEX=J
#310 PRINT "  CASE J>2 AND I>6",
#320 ENDCASE END OF OUTER CASE
ERROR 3 IN LINE  320 OF STRUCTURE.TEST     (Example of syntax error handling.)
  320 ENDCASE
#320 ENDCASE J END OF OUTER CASE
#330 NEXT J
#340 ENDIF
#350 PRINT "  ◆"
#400 NEXT I END OF LOOP, LINE NO LINKS FOR STM
#999 END PROC NAME ADDED BY SYSTEM
#LIST


VIPER  REV  A7  12/04/78  20:53:01.7  18/04/78

#     1 PROCEDURE STRUCTURE.TEST
   10    PRINT "SIMPLE STRUCTURE TEST"
   20 START STRUCTURE.TEST                             END OF INITIALISATION CODE
  100 FOR I=1 TO 7                                                      MAIN LOOP
  110    PRINT I,
  120    IF I<3                              BINARY IF ON ITS OWN FOR VISIBILITY
  130      THEN  PRINT "  I<3",                 THEN,ELSE AND UNARY IF CAN ONLY
  140      ELSE  PRINT "  I>=3",                   BE FOLLOWED BY A NON-CONTROL
  150        IF I=4 PRINT "  I=4",                          STM ON THE SAME LINE
  160    ENDIF
  200    IF I>=5
  210      THEN                  THE FOLLOWING CONTROL STM MUST BE ON A NEW LINE
  220        FOR J=1 TO 4
  230          CASE J=1                                           OUTER CASE INDEX=J
  240            PRINT "  CASE J=1",
  250            CASE I=6                                        NESTED CASE INDEX=I
  260              PRINT "  CASE I=6",
  270            CASE I=7
  280              PRINT "  CASE I=7",
  290            ENDCASE I                                        END OF INNER CASE
  300          CASE J>2 AND I>6                   COMPOUND CASE CONDITION, INDEX=J
  310            PRINT "  CASE J>2 AND I>6",
  320          ENDCASE J                                          END OF OUTER CASE
  330        NEXT J  220
  340    ENDIF
  350    PRINT "  ◆"
  400 NEXT I  100                              END OF LOOP, LINE NO LINKS FOR STM
  999 END STRUCTURE.TEST                                 PROC NAME ADDED BY SYSTEM
RUN
#SIMPLE STRUCTURE TEST
1   I<3  ◆
2   I<3  ◆
3   I>=3  ◆
4   I>=3  I=4  ◆
5   I>=3  CASE J=1   ◆
6   I>=3  CASE J=1   CASE I=6   ◆
7   I>=3  CASE J=1   CASE I=7   CASE J>2 AND I>6   CASE J>2 AND I>6   ◆
RUN
#1   I<3  ◆
2   I<3  ◆
3   I>=3  ◆
4   I>=3  I=4  ◆
5   I>=3  CASE J=1   ◆
6   I>=3  CASE J=1   CASE I=6   ◆
7   I>=3  CASE J=1   CASE I=7   CASE J>2 AND I>6   CASE J>2 AND I>6   ◆
```

TABLE 4.2 STATEMENT EXECUTION COUNT

```
TRACEON
#RUN
#1   I<3   ◆
2  I<3   ◆
3  I>=3   ◆
4  I>=3  I=4  ◆
5  I>=3  CASE J=1   ◆
6  I>=3  CASE J=1  CASE I=6   ◆
7  I>=3  CASE J=1  CASE I=7  CASE J>2 AND I>6  CASE J>2 AND I>6   ◆
LIST


VIPER  REV  A7  12/04/78  21:09:41.5  18/04/78

#     1 PROCEDURE STRUCTURE.TEST                                    0
   10   PRINT "SIMPLE STRUCTURE TEST"                               0
   20 START STRUCTURE.TEST                                          1
  100 FOR I=1 TO 7                                                  1
  110   PRINT I,                                                    7
  120   IF I<3                                                      7
  130     THEN  PRINT "  I<3",                                      2
  140     ELSE  PRINT "  I>=3",                                     5
  150       IF I=4 PRINT "  I=4",                                   5
  160   ENDIF                                                       7
  200   IF I>=5                                                     7
  210     THEN                                                      3
  220       FOR J=1 TO 4                                            3
  230         CASE J=1                                             12
  240           PRINT "  CASE J=1",                                 3
  250           CASE I=6                                            3
  260             PRINT "  CASE I=6",                               1
  270           CASE I=7                                            2
  280             PRINT "  CASE I=7",                               1
  290           ENDCASE I                                           3
  300         CASE J>2 AND I>6                                      9
  310           PRINT "  CASE J>2 AND I>6",                         2
  320         ENDCASE J                                            12
  330       NEXT J  220                                             8
  340   ENDIF                                                       7
  350   PRINT "  ◆"                                                 7
  400 NEXT I  100                                                   9
  999 END STRUCTURE.TEST                                           -1

TRACEOFF
#
```

TABLE 4.3 ERROR HANDLING PRACTICES

(a) FORTRAN: Example of local error handling (Hewlett Packard RTE FORTRAN)

1. No user error handling (all errors handled by system)

   CALL EXEC (CODE, <parameter list>)

2. User error control

   CALL EXEC (100000B+CODE, <parameter list>)

   GOTO <label>

   <normal code>

<label> <error handling code>

(b) REAL-TIME BASIC (KOPETZ,1976, BIANCHI 1976)

<statements>

ON ERROR GOTO <error line no.>

<statements>

<error line no> <error handling statements>

RESUME| RESUME <line no> | GOTO SYSTEM

Notes:

1. The ON ERROR GOTO is an executable statement and can appear anywhere in the program body. On occurrence of an error, control is transferred to the last specified <error line no.>

2. RESUME restarts execution at the line causing the error.

3. GOTO SYSTEM transfers control to the operating system.

TABLE 4.4 AN ERROR HANDLING EXAMPLE

```
VIPER  REV  A7  12/04/78  11:01:59.4  19/04/78

*     1 PROCEDURE STARTUP
     50 COMMON SPECS,NADC,ES,DELT
     60 LET ACCESS((SPECS)=READA+WRITEA
     70 LET NADC=30 ; ES=30              NO OF ADC CHANNELS/SIZE OF ENG COMMON
     80 LET DELT=30 ; DELTCS=5
     90 CALL TIME(YEAR,MONTH,DAY,HOUR,MIN,SEC)                 READ CURRENT TIME
    100 RUN SCANCS EVERY DELTCS SECS
    110 RUN SCANADC EVERY DELT SECS
    120 RUN WATCH.DOG EVERY DELT SECS
    130 RUN SERVOHOUR EVERY 1 HOURS  AT HOUR+1:0:0  RUN EVERY HOUR ON THE HOUR
    140 LET NEXTSHIFT=8*INT(HOUR/8)+6      SHIFTS ARE AT 22:00,06:00 AND 14:000
    150 RUN SERVO8HOUR EVERY 8 HOURS  AT NEXTSHIFT:0
    160 RUN FILTER.REPORT EVERY 8 HOURS  AT NEXTSHIFT:0
    170 PRINT "HULETTS FACTORY SOFTWARE STARTED UP AT" ;
    180 CALL PTAD                          PRINT TIME AND DATE TO LOG STARTUP
    190 END STARTUP

    300 ERROR
    310   CALL ERRORSN(LINE,ERNO)                   PICK UP STM NO AND ERROR NO
    320   IF ERNO=351 OR ERNO=232                            351=PROC NOT FOUND
                                                              232=ILLEGAL STATUS
    330     THEN                                         PRINT ERROR DIAGNOSTIC
    335       IF ERNO=351 PRINT "PROG NOT FOUND:",
    336       IF ERNO=232 PRINT "ILLEGAL STATUS",
    340       IF LINE=130 PRINT "SERVOHOUR ",
    350       IF LINE=150 PRINT "SERVO8HOUR ",
    360       IF LINE=160 PRINT "FILTER.MONITOR ",
    370       IF LINE<130 PRINT "ERROR AT LINE " ; LINE" ,PROG NOT FOUND"
    400     ELSE  PRINT "ERROR " ; ERNO" IN LINE " ; LINE" OF STARTUP"
    410   ENDIF
    500 ERET                               CONTINUE PROCESSING AT NEXT LINE
```

TABLE 4.5 SYSTEM DOCUMENTATION EXAMPLE

```
VIPER  REV  A7  12/04/78  20:16:02.1  19/04/78

  MASTER ()
    EXT:LIMERATIO* MAP*
    COM:
PROCEDURES:
  LIMERATIO (MASTER)
    EXT:MESSAGE* CDAC*
    COM:SPECS* VOLTS* ENG* BITS* GASFLOW*
  CLFLOW ()
    EXT:FILTERCOEF* MESSAGE CDAC LIMERATIO
    COM:SPECS* ENG* BITS*
  GASFLOWA ()
    EXT:MESSAGE CDAC
    COM:SPECS* ENG* BITS* GASFLOW*
  GASFLOWC ()
    EXT:MESSAGE CDAC
    COM:SPECS* ENG* BITS* GASFLOW*
  SATFLOW ()
    EXT:FILTERCOEF MESSAGE CAMAC
    COM:SPECS* ENG* VOLTS* BITS*
  REMELT ()
    EXT:DECLR* MESSAGE CAMAC
    COM:ENG* BITS* SPECS*
SUBROUTINES:
  MESSAGE (LIMERATIO)
    EXT:PRINT.MESSAGE TIME
    COM:
  CAMAC (CDAC)
    EXT:
    COM:
  FILTERCOEF (CLFLOW)
    EXT:
    COM:
  CDAC (LIMERATIO)
    EXT:DECLR* CAMAC*
    COM:
COMMONS:
  SPECS () 6 STARTUP SCANADC ENGUNITS ENGLIMITS SATFLOW* REMELT*
         CLFLOW* GASFLOWC* GASFLOWA* LIMERATIO*
  VOLTS () 60 SCANADC ENGUNITS SATFLOW* LIMERATIO*
  ENG () 60 ENGUNITS SATFLOW* REMELT* CLFLOW* GASFLOWC* GASFLOWA*
         LIMERATIO*
  ENGLIM () 120 ENGUNITS ENGLIMITS
  BITS () 12 SATFLOW* REMELT* CLFLOW* GASFLOWC* GASFLOWA* LIMERATIO*
  GASFLOW () 10 GASFLOWC* GASFLOWA* LIMERATIO*
#
```

CHAPTER 5

PERFORMANCE

There are a variety of criteria which can be applied to gauge the performance of a real-time system. These include both time factors and resource utilization. Time factors which may be of importance include throughput, response time to asynchronous or external events and task completion deadlines (deadline scheduling). Local memory, bulk storage and back-up storage requirements are examples of resources whose utilization must be considered.

The criteria which is considered almost exclusively in this chapter is that of throughput, i.e. how fast can the system perform its tasks. The reason for restricting attention primarily to this one area, is the interpretive mode of operation. There are many misconceptions concerning the performance of interpreters and the purpose of this chapter is to clearly indicate the capabilities and limitations of interpretive systems in general, and of VIPER in particular. A second reason for restricting attention to the execution time performance is that the other time criteria are of less importance to an interactive user-oriented system like VIPER.

The execution time of programs, which determines the throughput, is important in real-time systems for a slightly different reason than in batch oriented systems. In batch systems, if programs execute 20% faster, then the system can possibly achieve a 20% higher throughput and consequent increase in revenue i.e. achieving a faster execution time has a direct monetary incentive. In real-time process control applications however the CPU is typically busy only a certain proportion of the time on a cyclic basis; which is reportedly as low as 5% even in a relatively large installation (GALLIER, 1965). Provided the total set of cyclic tasks is executed in time it is therefore irrelevant whether the CPU is busy 5% or 90% of the time.

The execution time is important, however, to the extent that it determines the range of tasks to which the system can be applied. This is particularly true for VIPER because its modular properties permit it to be applied to a wider class of problems than simple BASIC-type systems. It has been observed that in certain cases BASIC is limited more by its structural inadequacies than by its execution time .....

time ...../5.2

time penalty.

This chapter is therefore primarily concerned with the execution time of VIPER, both in comparison with other BASIC-type systems and in comparison with systems executing in-line compiled or assembled code. Certain measurements which have been made to demonstrate the extent to which the present execution times of VIPER can be improved, are also reported. Discussion of other performance criteria such as memory and bulk storage requirements is defered to chapter 6.

There are four techniques which can be used to evaluate the performance of a software system:

1. Micro-analysis. This technique examines and compares the performance of individual operations and statements. While useful in under= standing the operation of system and in making comparisons between closely related systems, it is of little use when comparing dis= similar systems.

2. Macro-analysis, which is concerned with the performance of groups of statements which constitute a task, but still in abstract terms, i.e. not related to any particular program or task.

3. Bench marks, which are used directly to compare the performance of the same program in two different systems. The difficulty of per= forming an accurate, unbiased evaluation of the relative performance of interpretive systems has been noted by HAMMOND, (1977); LIENTZ, (1976) and HAASE (1976, due to the strong dependencies on the type and structure of the programs used for the benchmarks. To quote HAMMOND "In order to compare the two compilers and the interpreter, they must be made to process a typical BASIC program. Unfortunately a typical BASIC program is as difficult to find as the soap powder advertiser's typical housewife, and as unconvincing if found."

4. Case studies, which consider a typical application of the system or systems under consideration and consider their overall performance in performing the tasks which are required in the application.

In performing an evaluation of VIPER and of Software Virtual Memory Management, all four techniques mentioned above have been applied. The results of particular measurements in categories 1 and 2 are presented in Table 5.1 and the results of some simple benchmarks in table 5.2. The results of a case study are presented in chapter 6.

5.1 PERFORMANCE IN COMPARISON WITH INTERPRETIVE SYSTEMS

5.1.1 Comparison with VARIAN BASIC and PROSIC

VIPER was derived from a BASIC interpreter, and the essential inter= pretive processes have not been changed significantly. The first two columns of data in table 5.1 show the results of measurements on PROSIC and VIPER on the VARIAN 620i computer. Measurements on the Varian BASIC are not shown because they are identical to those for PROSIC. From these figures it can be seen that for simple operations in small programs, VIPER and PROSIC are almost identical in speed. This shows that the extra mapping and protection functions in SVMM incur only a small overhead.

One of the most notable differences between VIPER and PROSIC, is that in PROSIC the time to execute the control statements FOR-NEXT, IF and GOTO increases as the size of the program increases. This has a severe affect on the performance of medium to large programs, and in the 200 - 300 statement range VIPER is likely to be two or three times faster than PROSIC.

Four factors contribute to this improvement:

1. Task partitioning. In VIPER the partitioning of a task into a number of independent procedures reduces the time taken to perform typical branching operations. A 500 line task, for example, executes in less than half the time when partitioned into procedures with an average size of 50 lines. (A similar improvement can be obtained in BASIC by performing a partial compilation of the program before execution but this restricts the interactive facilities.)

2. ...../5.4

2. Structural linking. Using special descriptors on the descriptor table, fig. 3.3(e), for structural linking, an improvement in the performance of individual statements can be obtained. Compared to PROSIC, in VIPER the FOR-NEXT pair for example, executes in half the time in a 70 statement program and in 10% the time in a 600 statement program. The figures in group 1 of Table 5.1 illustrate this trend.

3. Structured programming. VIPER uses a structured language where the program flow follows well defined paths, a property which can be used to reduce the time taken for branching operations. This effect is shown in Table 5.1 groups 5 and 6.

4. Formal-actual parameter mapping. The linking structures used in SVMM significantly reduce the time taken for formal parameter referencing, as shown in group 8 of Table 5.1. This aspect was also discussed in section 3.2.4.

The mapping and protection of references to shared data items defined by COMMON, are also performed efficiently as shown by the figures in group 9 of Table 5.1. The increase in execution time ranges from 2,5 to 6,9%, which is minimal in view of the importance of protecting this type of data.

One of the specific claims of this thesis is therefore that Software Virtual Memory Management techniques can be used to enhance the performance of interpretive systems and that the overhead introduced by the virtual memory mapping and protection operations is acceptable in view of the overall improvement in performance which is obtainable.

In the fourth and fifth columns of table 5.1 measurements of VIPER's performance on MIKROV, the microprocessor based Varian emulator (VAN AARDT, 1977), are tabulated. The measurements in column four were obtained using the same version of VIPER as was run on the Varian 620i and the improvements directly reflect the higher speed of the emulator.

Column ...../5.5

Column five of table 5.1, and some results in Tables 5.2(a) and (b), show the result of measurements on VIPER using a different interpretive structure. The evaluator section of the interpreter was rewritten to handle code in Polish form and in addition, floating point firmware was used. The purpose of these tests was to obtain some idea of the performance improvement which could be obtained using readily available hardware and software enhancements. The syntactical routines were not modified for these tests and the various short test sequences were hand translated from infix to Polish form. (The rewriting of the syntactical and back-listing routines to compile and decompile to and from the Polish representation is being delayed pending the availability of a high level systems programming language. This aspect is discussed further in chapter 7.) The measurements which were obtained in this way indicate clearly the advantage of these enhancements. It should also be noted that these figures are conservative, as a further 20 to 30% improvement is obtainable by simplifying the code used in the initialization and control of the interpretive operation. The improvements which it is thought can be reasonably obtained are documented in Table 5.3. The overall improvement which is noted in Tables 5.1 and 5.2 is about 3 to 1 with a factor of 4 or 5 to 1 being achievable with this "streamlining" operation. A point which was observed in making these measurements, is that as the time spent on the floating point arithmetic and on the precedence determination operations is reduced, the proportion of the time taken by the virtual memory mapping and protection function increases. This effect is shown in Table 5.3. The example shown in the table is the worst case, as when floating point operations are involved, the mapping operations take proportionally less time. An estimate of this effect is shown in the second half of Table 5.3.

This data illustrates that there is a limit to the performance which can be attained when using software virtual memory management. Further improvements could only be obtained by moving some of the mapping and stack operations into firmware. This is one of the intrinsic limitations of software virtual memory management, and in applications where executing speed is of primary importance SVMM may not be a suitable technique.

5.1.2 .....5.6

5.1.2 Comparison with other BASIC's

Some figures comparing VIPER with Hewlett Packard BASIC are given in the last two columns of Table 5.1. The results of some simple benchmark tests which are given in Tables 5.2(a), (b) and (c) extend this comparison to a four other BASIC and interpretive systems.

The comparison with the Hewlett Packard BASIC is of interest because the HP21MX computer was used for the FORTRAN versions of the case study programs. From an examination of the source listing of the HP BASIC it was determined that its interpretive mode of operation was similar to PROSIC viz, interpretation of meta-codes stored in infix form. The measurements of individual micro-operations therefore re= flects to a large extent the difference in the average instruction execution time of the various machines. From the figures in Table 5.1 it can be seen that, excluding the trigonometic functions, the HP BASIC is 40 to 60% faster than PROSIC or VIPER on the Varian 620i and 30 to 50% faster than the MIKROV. This difference corresponds roughly with the difference in average instruction execution time recorded in notes (9), (10) and (11) of Table 5.1. Like PROSIC and Varian BASIC, the performance of the HP BASIC deteriorates rapidly as the program size increases. In programs with 50 to 100 statements, even the infix form of VIPER would outperform the HP BASIC. The anomalous results obtained for the trigonometric functions illustrates the difficulty of making objective comparisons between even similar systems. This anomally also distorts the results of the benchmark measurements, as noted below.

One other result which is of interest in Table 5.1 is the data for the HP Fast BASIC (GANS, 1975) as it illustrates the improvement which can be obtained by placing the floating point functions in firm= ware rather than software. The overall improvement in typical programs would appear to be of the order of 2 to 1 i.e. using floating point firm= ware the execution time can be halved.

Table 5.2 shows the results of measurements from some simple bench= mark programs. These benchmarks are of interest despite the simpleness of some of them because results of measurements on several other com= puter systems have been published (FULTON, 1977; MAPLES, 1977; VAN MEURS, 1977). These results are also shown in Tables 5.2(a), (b) and (c) together with listings of the programs. Some of the tests were also run using the HP BASIC and FORTRAN.

From ..../5.7

From the results of these benchmark measurements five observations can be made:

1. The performance of VIPER is considerably better than the simpler PDP and INTEL BASIC systems and comparable to the performance of systems running on much more powerful machines such as the PDP 11/45 and Data General 840. From this observation it can also be stated that the Software Virtual Memory Management operations do not affect the performance of VIPER vis-a-vis that of ordinary interpreters.

2. The benchmarks which have been published are inadequate and at times misleading. The excellent performance of VIPER in some of the benchmark programs can be attributed largely to the efficiency with which the trigonometric functions have been implemented. (This occurs as a result of a trade-off in space versus speed. The Varian BASIC trigonometric functions take twice the space of the HP functions but execute in one quarter of the time.)* There is a need for better benchmark programs to be developed.

3. Interpretive programs are reasonably efficient when executing scientific type calculations involving largely floating point operations. Where integer arithmetic is used extensively, as in the sort segment of Benchmark 3 - Table 5.2(c), the compiled programs execute in dramatically less time.

4. Programs which interpret source code directly, such as ABACUS/10 - Table 5.2(c), are more than an order of magnitude slower than systems executing either infix or polish meta-code forms. A number of early BASICs used this interpretation technique and at least some of the prejudice against interpreters can be traced to experience (and rumour) with these early systems.

5. ...../5.8

---

*Contrary to appearances, the benchmarks were not chosen because of VIPER's superiority in this respect; they were the only ones found in the literature. It was only after these somewhat anomalous benchmark results were obtained that the SIN and ATAN functions were added to Table 5.1 to show the cause of the discrepency.

5. The interactive system ABACUS/X described by FULTON (1977) executes compiled code, using an incremental compiler. Other BASIC-like systems which execute compiled code have been described by KOPETZ (1976) and WILKENS (1976). In these systems the conversion to in-line code is either performed line-by-line at input time, or from an internal meta-code format immediately prior to execution. Even in this latter case the conversion is very fast because only the code generation must be performed without any lexical or syntactical scanning being required. Because of the high speed of the conversion (typically a few tenths of a second) the operation is virtually unnoticed by the user and the system still appears to have the attributes of an interactive interpreter. In one-off batch or "student" jobs this is an excellent approach, but as the compiled module has all the characteristics and disadvantages of code generated from conventional compilers, this technique cannot be used in a real-time multiprogramming environment without sacrificing the interactive facilities to a greater or lesser extent.

5.2 PERFORMANCE IN COMPARISON WITH SYSTEMS EXECUTING IN-LINE CODE

No detailed comparison using benchmark programs has been made to determine the difference between VIPER and similar programs executing compiled code. The results of the case study of chapter 6, and the scattered results recorded in Table 5.2, are, however, adequate to demonstrate the general nature of the difference.

In the remainder of this section some results from the literature are quoted and some observations made on the factors which influence the difference between the two types of systems.

A detailed comparative analysis of the relative performance of interpretive and in-line code has been performed by HAMMOND (1977). On a set of five "representative" test programs interpretation was an average of 5 times slower than in-line code. In three quite different applications using different computers and different software

organizations, .../5.9

organizations, ADIX (1975), HELPS (1974) and FOSTER (1973) have reported similar figures for the ratio between interpretive and compiled code.

In the case study the ratio between the execution time of 12 programs which were written in both VIPER and FORTRAN has been measured to be 6.6 to 1. An estimate of the true ratio between interpretive and compiled code is difficult to make from this result, however, because of a number of conflicting factors. These factors are discussed and taken into account in chapter 6 where it is concluded that the execution time ratio between interpretively executed code in VIPER, and compiled in line code, is of the order of 6 to 1. This corresponds closely with the results obtained by other workers which were noted above.

A comparison between the performance of the SVMM and other mini= computer real-time executives is rather more difficult owing to the fundamentally different nature of the two processes. Even an approximate answer can be given only if the characteristics of the tasks to be performed are known reasonably well. A few general observations can be made, however. A real-time process consists typically of a large number of concurrent tasks of various priorities, and as a result the processor is switched frequently from one task to another. If all these tasks are executed in one, or at best a few, memory partitions, the CPU is busy only a small percentage of the time because of the time spent rolling tasks in and out of memory. In the SVMM system however, execution of one task can, in general, proceed concurrently with the swapping of another task, so that the CPU can be kept busy a greater proportion of the time. Even if concurrent execution with swapping is not allowed, (as in the current version of VIPER) the compactness of the interpretive code ensures that many more modules are simultaneously resident in memory. The swapping rate is then reduced accordingly. In the case study for example none of the cyclic real-time tasks need be swapped at all.

The corollary that follows from this observation is that the ratio between the total throughput in a system like VIPER and in a compiler-based system is generally less than the ratio between the execution times of individual programs in the two systems.

In the case study, the foreground partitions of the HP RTE-3 operating system in which the FORTRAN programs ran, were measured to be busy about 15% of the time. (The majority of this time was spent in swapping tasks as the CPU itself was only busy about 2% of the time.) The same set of tasks in VIPER keep the MIKROV CPU busy 12,8% of the time. In terms of the real time tasks which can be supported, the two systems can therefore said to be closely related in capacity, despite the fact that the actual computing speed of the VIPER programs is 6,6 times slower than the FORTRAN programs.

A claim of this thesis is therefore that in a real-time multi=programming environment, an interpretive system using SVMM can perform as well, or better, than a compiler oriented system executing in-line code with swapping. Furthermore, this performance is achieved without recourse to large, expensive and unreliable electromechanical bulk-storage devices, and even more importantly, without sacrificing either the interactive facilities or the protection functions of the interpretive system.

TABLE 5.1 EXECUTION TIME OF STATEMENTS

GROUP	STATEMENT TYPE (The statement numbers indicate the association of statements within a group. Groups 2 to 9 all execute within group 1 statements.)	TOTAL NUMBER OF STATEMENTS (2)	STATEMENT EXECUTION TIME - MILLISECONDS (1)						
			VARIAN 620i (9)		MIKROV (10)		HEWLETT PACKARD 21MX (11)		
			PROSIC (3)	VIPER (4)	VIPER (4)	VIPER (5)	BASIC (6)	FAST BASIC (7)	FORTRAN IV (8)
1	3 FOR I = 1 TO 10 000	2	1,9	1,96	1,72	0,65	1,13	0,46	0,017
	9 NEXT I	50	3,4	1,96			2,17	1,51	
	10 END	100	4,6	1,96			3,48	2,80	
2	4 R = RND(I)	3		1,81	1,43	0,57	1,13	1,12	0,635
3	5 X = R	4	1,2	1,16	0,94	0,36	0,60	0,60	
	5 X = I*R	4	2,35	2,37	1,95	0,62	1,91	1,05	0,027
	5 X = I+R	4	2,20	2,24	1,89	0,61	1,43	1,03	0,017
	5 A(1) = R	4	2,30	2,08	1,74		1,90	1,89	
	5 X = SIN(R)	4			4,14		16,18	3,64	1,16
	5 X = ATN(R)	4	9,3		8,50		22,57	4,77	2,44
4	5 IF R<0,5 THEN 9	5					1,44	1,15	
	6 X = R	50					3,40	3,11	
		100					5,37	5,07	
	5 IF R>=0,5 LET X = R	4		2,41	2,04	0,79			
5	5 IF R<0,5 THEN 8	7					1,87	1,59	
	6 X = R	50					4,49	4,18	
	7 GOTO 9	100					7,20	6,77	
	8 X = I								
6	6 IF R>= 0,5	7		4,13	3,38	1,30			
	6 THEN X = R	50		4,13	3,38	1,30			
	7 ELSE X = I	100		4,13	3,38	1,30			
	8 ENDIF								
7	5 GOSUB 100		1,7				0,72	0,72	
	100 RETURN								
	CALL SUBX			3,5					
	SUBROUTINE SUBX								
	RETURN								
8	5 GOSUB 100, R, I, X, 4, 5		6,25						
	100 SUB A, B, C, D, E								
	102 RETURN								
	CALL SUBX (R, I, X, 4, 5)			6,90					
	SUBROUTINE SUBX (A, B, C, D, E)								
	RETURN								
	101 C=A+B		8,8	2,40					
9	2 COMMON COM1, R, I, X, A(2)	% INCREASE FROM GROUP 1, 2, 3							
	FOR - NEXT I	2,5%		2,01					
	X = R	6,9%		1,24					
	X = R*I	5,1%		2,49					
	A(1)=R	2,9%		2,14					

NOTE: The numbers in parenthesis (1) to (11) refer to the notes on the next page.

TABLE 5.1 (CONT.) NOTES

(1) Time for actual statement indicated i.e. excluding FOR-NEXT overhead and time to generate random number.

(2) In PROSIC and HP BASIC the time to execute a statement is dependent on the total number of statements in the program, including REMS. The statements need not be inside the FOR-NEXT loop.

(3) PROSIC is similar to VARIAN BASIC with some small improvements.

(4) VIPER - Infix form for meta codes.

(5) VIPER - Meta-codes stored in Polish form, using floating point firmware.

(6) HEWLETT PACKARD stand alone BASIC HP 20392A Sept. 1974.

(7) HP BASIC modified to use floating point firmware (University of Natal Fast BASIC - GANS, 1975).

(8) FORTRAN IV running under RTE-2 on 21MX with hardware FAST FORTRAN firmware.

(9) VARIAN 620i: 1,8 μs memory cycle time, 4μs average instruction time.

(10) MIKROV INTEL 3000 based emulator of Varian V70 instruction set: 450 ns memory cycle time, 3,5μs average instruction time (VAN AARDT 1977).

(11) HP 21MX 660ns memory cycle time, average instruction execution time approximately 2,5μs

TABLE 5.2 BENCHMARK DATA

(a) BENCHMARK 1

COMPUTER AND LANGUAGE	TIME PER LOOP MILLISECS	
Published Data (MAPLES 1977)		
Data General 840 Multi-user BASIC	4,5	
DEC PDP 11/45 BASIC	3,2	
DEC PDP 8E FOCAL	38,0	
INTEL 8080 BASIC	75,0	
INTEL 8080 compiled BASIC (Lawrence Livermore Laboratory)	22,0	
VIPER - Varian 620	14,4 (1)	13,1 (2)
VIPER - MIKROV	12,0	10,7
VIPER - MIKROV + Polish + Firmware (Note 5 Table 5.1)	4,2	-
Hewlett Packard 21MX (See notes 6, 7 and 11 Table 5.1)		
1. HP BASIC	10,7	
2. HP Fast BASIC (ex University of Natal)	6,7	
3. HP FORTRAN IV	0,18	

BASIC

```
 10  REM SIMPLE BENCHMARK
 15  REM *, /, -, +
 20  REM
 30  LET  A = 1
 40  LET B = RND(A)
 50  LET C = A + B
 60  LET A = A + 1
 70  LET E = B/C
 80  LET F = A*E
 90  LET C = C-F
100  IF A = 1001 THEN 200
110  GOTO 50
200  PRINT "THE LOOP IS DONE"
210  END
```

VIPER (1): as BASIC except

```
100IF A = 1001 GOTO 200
```

VIPER (2)

```
30 DOWHILE A<=1000
40 LET C=A+B;A=A+1;E=B/C;F=A*E;
                          C=C-F
50 END DO
```

TABLE 5.2(b) BENCHMARK 2

COMPUTER AND LANGUAGE	EXECUTION TIME-SECS	
	PROGRAM 1	PROGRAM 2
Published data (VON MEURS 1977)		
DEC PDP 11/40 with DOS/11 V8.08 operating system		
1. DEC FORTRAN V004A	3	21
2. DEC BASIC V008A	45	134
3. BACO (Tagged data structure interpreter)	14	47
VIPER - MIKROV	14,5	41,9
VIPER - MIKROV Polish Notation + Firmware	5,1	22,8 (1) 13,2 (2)
Hewlett Packard 2/MX		
1. HP BASIC	8,7	83,6
2. HP FAST BASIC	6,3	24,7

(1) Measured, SIN function not using floating point firmware.
(2) Estimated, SIN function using " " "

Program 1

```
10 LET X = 0
20 LET X = X + 0,1
30 IF X <360 GOTO 20
```

Program 2

```
10 LET X = 0
20 LET PI = 3,1415
30 LET Y = SIN (2*PI/360*X)
40 LET X = X+0,1
50 IF X <360 GOTO 30
```

TABLE 5.2(c) BENCHMARK 3

COMPUTER AND LANGUAGE	N=250 (BASIC Array limit)		EXECUTION TIME-SECS 1.	EXECUTION TIME-SECS 2.
Published data (FULTON 1977)				
Data General 840				
1. FORTRAN IV } Standard Data General Translators			13,05	9,91
2. Extended BASIC } Standard Data General Translators			46,84	145,30
3. ABACUS/X - Incremental Compiler			13,18	11,14
4. ABACUS/10 - Interpreter			77,89	1 600,57
VIPER - MIKROV			18,1	158
Hewlett Packard 21MX				
1. HP BASIC	13,4	29,0	53,2*	79,1*
2. HP FORTRAN IV	0,51	0,55	2,01	1,50

1. Computation Segment 2. Sorting Segment *Extrapolated

BENCHMARK -- GENERATE SOME NUMBERS AND SORT THEM

```
C       FORTRAN
        DIMENSION A(1000)
        N=1000
        TYPE "<7>START"
C
C       COMPUTATION SEGMENT
C
        DO 100 I=1,1000
        Q=I
        X=SIN(Q)*COS(Q)
        X=X*4000.
        X=SQRT(ABS(X))
100     A(I)=AINT(100.*X)
C
C       SORTING SEGMENT
C
        TYPE "<7>SORT"
        N0=N
220     N0=N0/2
        IF(N0.LE.0) GO TO 380
        K=N-N0
        J=1
260     I=J
        M=I+N0
280     IF(A(I).LE.A(M)) GO TO 350
        T=A(I)
        A(I)=A(M)
        A(M)=T
        M=I
        I=I-N0
        IF(I.GE.1) GO TO 280
350     J=J+1
        IF(J.LE.K) GO TO 260
        GO TO 220
```

```
0001.1000 C ABACUS/X
0001.2000 SET N=1000
0001.3000 TYPE "<7>START"
0001.4000 /REL A(N)
0001.5000 FOR I=1,1000;DO 99

0002.0100 C SORT ROUTINE
0002.0150 TYPE "<7>SORT"
0002.0200 SET N0=N
0002.0300 SET N0=N0/2
0002.0400 IF (N0<=0) GOTO 2.19
0002.0500 SET K=N-N0
0002.0600 SET J=1
0002.0700 SET I=J
0002.0800 SET M=I+N0
0002.0900 IF (A(I)<=A(M)) GOTO 2.16
0002.1000 SET T=A(I)
0002.1100 SET A(I)=A(M)
0002.1200 SET A(M)=T
0002.1300 SET M=I
0002.1400 SET I=I-N0
0002.1500 IF (I>0) GOTO 2.09
0002.1600 SET J=J+1
0002.1700 IF (J<=K) GOTO 2.07
0002.1800 GOTO 2.03
0002.1900 TYPE "<7>FINISH"
0002.2000 QUIT

0099.0500 C COMPUTATION OF VALUES TO SORT
0099.1000 SET X=FSIN(I)*FCOS(I)
0099.2000 SET X=X*4000
0099.3000 SET X=FSQT(FABS(X))
0099.4000 SET A(I)=FITR(100*X)
```

TABLE 5.2(c) BENCHMARK 3 VIPER VERSION

```
VIPER  REV  A7  3/04/78   11:10:35.9  24/04/78

#     1 PROCEDURE ABACUS.BENCH
   10 LET N=1000
   20 DIM A(N)
   30 FOR I=1 TO N
   40   LET Q=I ; X=SIN(Q)*COS(Q) ; X=X*4000
   50   LET X=SQR(ABS(X)) ; A(I)=100*X
   60 NEXT I   30
   70 PRINT "*"
  100 LET N0=N/2
  110 DOWHILE N0>0
  120   LET K=N-N0 ; I=J=1
  130   DOWHILE J<=K
  140     LET M=I+N0
  150     IF A(I)>A(M)
  160       THEN  LET T=A(I) ; A(I)=A(M) ; A(M)=T
  170         LET M=I ; I=I-N0
  180         IF I>=1 GOTO 150
  220     ENDIF
  230     LET J=J+1 ; I=J
  240   ENDDO
  250   LET N0=INT(N0/2)
  260 ENDDO
  300 PRINT "*"
  999 END ABACUS.BENCH
```

TABLE 5.3 INSTRUCTION BREAKDOWN

Approximate number of machine instructions executed by VIPER when interpreting infix and Polish representations of statement:

LET X = R

(Table 5.1 Group 1)

Operation	Number of Instructions		
	Infix	Polish	"streamlined" Polish
Initialization	25	20	2
Stack operations	30	20	20
Precedence determination	135	-	-
Assignment	10	10	10
Mapping	30	30	25
Next statement calculation	20	20	2
Total no of instructions	245	100	59
Measured execution time, ms	0,94	0,36	(0,23)*
Proportion spent on mapping	12%	30%	42%

*Estimated

Estimated time spent on mapping in operations involving arithmetic functions			
Using floating point software	8%	12%	13%
Using floating point firmware	11%	25%	30%

CHAPTER 6

CASE STUDY

The case study deals with a process control project at the Huletts Sugar Refinery at Rossburgh in Durban. This project was a co-operative venture between the National Electrical Engineering Research Institute (NEERI) and Huletts Refineries Ltd. NEERI was responsible for all computer and systems software while Huletts was responsible for all instrumentation. The applications software was developed jointly by staff of both organizations. I was project leader of the project from its start in 1975 until its termination in 1978.

This case study is of interest because most of the FORTRAN programs used on this project have been translated into VIPER, permitting a direct comparison to be made between FORTRAN and VIPER. The comparison deals with four factors.

1. Memory space requirements.
2. Relative execution speeds.
3. Bulk storage requirements.
4. Readability of code and ease of implementation.

The first three comparisons are based on quantative data obtained from direct measurements while the last is a subjective, but no less important, assessment of the "useability" of the two systems.

The characteristics of the process and of the hardware and software used are tabulated in Appendix B, in addition to being summarised below:

6.1 FORTRAN IMPLEMENTATION

A Hewlett Packard 21MX computer was used, running initially under control of the RTE-2 executive with 32K of memory and later, (August 1977 onwards) under RTE-3 using 48K of memory. All the applications software was written in FORTRAN IV. The computer is interfaced to the plant instruments using a CAMAC interface. Detailed

process ...../6.2

process studies had to be performed concurrently with initial control work, the first control loop being placed on-line in January 1977 and the six other main control loops going on-line at approximately two month intervals as the process studies proceeded. The modular decomposition of the software was therefore essential to permit independent testing and debugging of new programs without disturbing existing control programs.

The software is organised as a series of 17 separate control and monitoring programs and approximately 45 supporting subroutines and programs. All the control programs and some of the service programs are listed in Table 6.1. The synchronization of the various modules is achieved using semaphones (called Resource Numbers in RTE). The only memory resident shared data is a blank COMMON area as RTE does not support labelled COMMON in a multiprogrammed environment. Various disc files are also used for shared data as well as for data base operations.

RTE 2 can address a maximum of 32K words of memory resulting in a single foreground area of 6K words in the configuration used in Durban: 14K for resident system and drivers; 10K for background (minimum for FORTRAN compiler); 1K for system buffering; 1K for COMMON . All the control programs were therefore swapped in and out of this single foreground area. This caused two problems; a high disc access rate and difficulties with the debugging of foreground programs, as described in the "Pathological Debugging Problem" of section 4.2.1. These problems and others, such as chronic base page overflow, led to the installation of RTE 3 in August 1977. Using the system with 48K of memory enabled three foreground memory partitions to be provided of 2,4 and 8 Kwords respectively. This reduced the disc swapping rate and permitted larger foreground programs, but did not otherwise materially affect the organization or structure of the software.

The source listings of the FORTRAN programs are provided in Appendix B.3.

6.2 VIPER IMPLEMENTATION

The FORTRAN programs were translated into VIPER directly, retaining the structure of the original programs except as noted below:

1. GOTO statements in the FORTRAN programs were avoided in all cases (the VIPER programs do not use any GOTO's) requiring a certain amount of logical reorganization to use VIPER's control structures.

2. In a few cases the programs were significantly reorganised to either take advantage of the modular properties of VIPER or to avoid particularly poor construction in the FORTRAN programs. These programs are marked with a (*) in Table 6.1.

3. As a result of the interactive facilities in VIPER a number of the FORTRAN programs are not required at all. Other functions such as CAMAC error reporting are included in the resident VIPER nucleus - some of these programs are listed in section 3 of Table 6.1.

The listings of the VIPER programs are given in Appendix B.2. Table 6.1 lists all the VIPER programs which have been written together with their size parameters. The program size information is summarised in Table 6.2 while the data areas which are used in the FORTRAN and VIPER versions are tabulated in Table 6.3.

6.3 COMPARISON BETWEEN FORTRAN AND VIPER PROGRAMS

The two different implementations are compared in size, Table 6.1 and in execution time, Table 6.4.

6.3.1 Size comparison

The sizes of the programs in the two systems can be compared in three classes:

1. Repetitive programs which execute either periodically (with a period of 5 to 30 seconds) or asynchronously in response to frequent external events.

2. Non-repetitive programs or programs which execute infrequently in response to external events.

3. Monitoring and service programs which are used to observe the performance of the control programs.

The size of the FORTRAN programs can be expressed in two ways. The one is the actual size of the program module (RTE-2 size) and the other the size of the smallest partition into which the program would fit in RTE-3 (expressed in pages, each page being 1K words in size). The RTE-2 size is quoted in order to asses how much space would be required if the programs were packed one against each other in a foreground resident partition. The RTE-3 size results from rounding the RTE-2 size up to the next highest page and adding one page for base page data and linking. From the figures tabulated in 6.1 it can be seen that the VIPER programs are in all cases considerably smaller than their FORTRAN counterparts. Furthermore in a 32K memory system, all the VIPER programs would fit into memory, enabling the system to operate without a bulk storage device. This is a significant and major advantage of VIPER over compiler oriented systems. Even if a number of additional programs were added, most of the repetitive programs would still fit into memory and only the less frequently used programs would have to reside on a bulk storage device. As disc units are quoted to have up to four times the failure rate of memories and CPU's (BHAT, 1976), avoiding the use of an electro-mechanical device for time critical tasks can make a marked contribution to the reliability of a system.

It is physically impossible to place the repetitive tasks in memory in RTE-2. Even if a subset of the critical tasks was selected which was only 6 to 8K in size, the system would be unworkable be= cause there would be no foreground partition in which to run the other tasks. As RTE-3 supports more than 32K of memory, a partition could be allocated to each task (or a group of tasks) if sufficient memory was available. This would require 60K words for the repetitive tasks which is 5 times more than VIPER requires. In addition to this 60K words, a foreground partition would still have to be provided plus

a ...../6.5

a background partition making a total of nearly 100K words in all. Even when using this amount of memory a disc storage unit is still required not only for swapping the non-repetitive tasks but also for supporting the language processing and file management facilities.

An important point to be noted is that this saving of space in VIPER is achieved without any particular attention having been paid to the storage and packing of the interpretive meta-codes. Using suitable meta-code structures HELPS (1974) and ADIX (1975) have shown that code compression factors of 0,5 to 0,3 can be achieved. BROWN P (1976a) has also discussed the use of compact codes and shown that the original source text can still be recreated from them. The aspect is commented on further in the concluding chapter, sections 7.2.2 and 7.2.3.

6.3.2 Speed comparison

6.3.2.1 FORTRAN measurements

The execution time of the FORTRAN programs was measured by running two low priority tasks, each of which measured the time which it spent computing. The one task was run in the background partition, while the other ran in a foreground partition. (Which is called a real-time partition in RTE-3.) The size and number of the partitions is shown in Table 6.4. The measuring programs have the lowest priorities.

If the measuring task running in the background partition (partition 4) is of a lower priority than the one in the foreground (partition 3), then the availability of partition 4 represents the time when the system was busy swapping and did not have a program to execute in any foreground partition. The availability of partition 3 represents the time when the system could have been processing additional real-time tasks. Items 1, 3, 4 and 5 of Table 6.4 illustrate measurements of this sort.

If the measuring task running in the background partition 4 is of higher priority than the one in the foreground, then the availability of the partition 4 is a measure of the availability of the CPU i.e.

it ...../6.6

it is the time that the CPU is not busy executing real-time tasks. The CPU is only switched from the background task to a real-time task when the swapping in operation is completed, and immediately returns to the background task when the foreground task is complete i.e. it does not have to wait to be swapped in nor does it have to wait for the real-time task to be swapped out. Item 2 shows this measurement.

To simulate the performance of RTE-2, which has only two partitions, a foreground and a background, two other small programs were run which generated operating system calls to lock a partition exclusively. These locking programs did not consume any overhead as they had the lowest priority.

The availability of the CPU can be determined to a first approximation (ignoring the effects of the measurement programs them= selves) by summing the availability of the individual partitions.

The measuring programs introduce, or are subject to, a number of errors. When measuring the availability of a foreground partition, for example, the measuring task also measures the time taken to swap itself in and out of memory. Even if the task does not have to be swapped, overhead is introduced by the additional scheduler context switches. The dispatcher must always switch back to the waiting measuring task when the control programs are not executing, instead of merely returning to an idle state. A more serious error is introduced by the resolution of the clock, which is 10 ms in RTE. Programs which complete executing in less than 10 ms will not be recorded by the measuring task. Even though this effect and the error introduced by the measuring program overheads act in opposite directions, the net affect is inpredictable. The results in Table 6.4(a) are therefore only approximate but are considered adequate to determine the general nature of the performance of the FORTRAN system and to compare its performance with that of VIPER. SPANG (1974) has commented on this difficulty of the performance evaluation tools themselves influencing the measurement results. The only solution is to use hardware performance evaluation aids, as is done in large systems, but this was considered

unnecessarily ...../6.7

unnecessarily complex for the system at hand where all that is required is an indication of the relative performance of two dissimilar systems.

The measurements were performed with all the control programs listed in Table 6.4(b) running, the results being tabulated in Table 6.4(a). The slight variations in the figures as different partitions are available, are not considered significant and it can be seen that the essential characteristics of the system are not changed by the use of additional partitions. The primary purpose of the additional memory space in RTE-3 was to reduce the disc access rate and to permit larger foreground programs to be used. Estimates of the average time that the CPU and real-time partitions are busy have been made from these figures and are noted at the end of Table 6.4(a).

6.3.2.2 VIPER measurements

The ease with which test data and programs could be generated in VIPER, permitted the individual execution times of all the programs to be measured. From these measurements, which are listed in Table 6.4(b), the total time that the CPU is busy computing can be determined from a knowledge of the relative frequency of execution of each program. This is known deterministically for all except the one program SERVOTIP, for which a statistical weighting factor can be calculated. These weighting factors are listed in the second column of Table 6.4(b).

The average time busy computing in each 60 second period is 7,92 seconds or 13,2%. As all these programs can be simultaneously resident in memory, there is no swapping overhead to be measured or taken into account. The computation time is therefore a direct measure of the overall availability.

6.3.2.3 Comparison

The results obtained from this case study are of interest for two reasons: firstly they indicate the gross performance capabilities of VIPER irrespective of any differences in the machines or in the measurement techniques used; and secondly they permit an estimate

to ...../6.8

to be made of the relative performance of interpretive versus compiled code.

Ignoring all differences between the HP 21MX and MIKROV computers, the results indicate that VIPER, running on a micro=programmed microprocessor emulator, is capable of substantially the same throughput of real-time tasks as a real-time executive which executes in-line compiled code with swapping. The HP RTE system could of course, also support concurrent tasks in the background partition, which could utilize the time when the foreground partitions are idle because swapping is in progress. As the most common tasks executed in this background area are editing, compiling and link loading, however, (none of which are required in an interactive system like VIPER), this argument is somewhat specious. Nevertheless, it is not claimed that VIPER is equivalent to a system like the HP RTE in computational power; only that given a set of real-time tasks, such as those encountered in the case study, VIPER has much the same performance and could be used in many applications where much larger and more complex operating systems had to be used previously.

It can be argued that the inefficient way in which the FORTRAN programs are organised, contributes to the good performance of VIPER relative to the RTE system. Frequently used programs like SCCS, or programs which take a relatively long time to complete like SCAD, could be placed resident in memory and other programs could be combined together into larger modules. These changes reduce the flexibility and modularity of the programs however, and it makes it either impossible or more difficult to perform on-line changes and upgrades. The execution time would have to be far more critical before retro=gressive changes of this type are justified.

The second aspect of the VIPER and FORTRAN measurements which is of interest, is an estimate of the ratio between the time to perform a given function in interpretive code, and the time to perform the same function in compiled code. The direct measured ratio made on the

programs ...../6.9

programs of the case study is 6,6, as noted in Table 6.4(b).

Extrapolating this ratio to obtain a direct indication of the difference between compiled and interpretive code is difficult because of a number of factors:

1. VIPER was running on a microprogrammed microprocessor emulator, whereas the FORTRAN programs were running on an HP 21MX.

2. The VIPER programs are functionally equivalent to the FORTRAN versions, but some of the VIPER programs are significantly simpler and execute less code as a result of their modular properties.

3. The RTE operating system in which the FORTRAN programs are running introduces an unknown overhead into the measurements.

4. The measurements on the FORTRAN programs are subject to un= certainty, particularly insofar as the CPU utilization is concerned as this could only be measured indirectly.

Taking these factors into account where possible, a ratio of about 6 to 1 in interpretive to compiled code execution times is estimated, with a possible variation between 5 to 1 and 8 to 1.

6.3.3 Bulk storage requirements

A particular advantage of interpretive systems which use an internal meta-code format is that only one copy of any program need be kept in the system. This contrasts with compiler oriented systems where three copies are usually retained: the source, the relocateable binary (output from compiler or assembler) and the absolute binary (memory image). The relocateable binary is required for loading purposes and also during the system generation, if a program is to be permanently linked into the system. The bulk storage requirements of the FORTRAN programs used in the case study are listed in Table 6.1. Taken together with the storage requirements of the absolute binary modules, the total bulk storage required for just the class 1 and 2 programs is 127 K words. This contrasts with the 15,3 K words required for all the VIPER PROGRAMS. The VIPER programs do not contain many

comments ...../6.10

comments (for reasons outlined in section 5.1.2), but even allowing 10 K words for comments, the space required for the VIPER programs is one fifth of that required for the FORTRAN programs. HOARE (1975) has commented on this desirability of reducing the bulk storage requirements by storing source programs in a more compact form and by eliminating additional copies of programs where possible.

In addition to the space required for the class 1 and 2 programs of Table 6.1, additional space is required for the monitoring and service programs (class 3 Table 6.1); for the several hundred library modules which are used by the linking loader; and for a few dozen files that are used for process communication functions. (Disc files used for logging process data are not included.) The total bulk storage requirements is therefore more like 500 K words. (If system generations are performed on the same system this requirement increases to 900 K words or more.)

The difference in the bulk storage requirements of the two systems has two important consequences:

1. Because the VIPER programs use far less space, smaller higher speed bulk storage devices can be used. Bubble or CCD memory devices in particular, would appear to be eminently suitable for use in an SVMM environment.

2. All on-line bulk storage devices should have some form of back-up facility. In the case of a system like RTE which uses a cartridge disc, the only feasible back-up medium is either magnetic tape or another disc unit, adding additional complexity and cost to the system. In the case of VIPER, cassette tape units have been used exclusively for off-line and back-up storage and a simple device such as this would be adequate for many applications. Floppy disc units would also be well suited for use in an SVMM system, provided a higher speed device such as bulk semiconductor RAM or CCD memory was available for the intermediate swapping operations.

A claim of this thesis is that Software Virtual Memory Management can use smaller, cheaper and higher speed bulk memory devices to achieve a similar or better performance than compiler oriented systems, without degrading the security of the system in any way. Furthermore, recent developments in bulk storage tech= nology can be readily incorporated into a system like VIPER.

6.3.4 Ease of use

The preceding three sections have dealt with quantative data obtained from measurements on the case study programs. More difficult to quantity, but just as important is the ease with which the system can be used. This is concerned with factors such as the debugging of programs, readability of code, documentation, safety and security, and ease with which programs can be written.

From my experience with the two systems over a period of two years, the following observations can be made:

1. The modular, structured code produced in VIPER is far easier to read and understand than the FORTRAN source.

2. The division of the global FORTRAN COMMON into separate named COMMON areas made a marked contribution to the safety of the system and permitted the data and program relationships to be visualised more clearly.

3. The VIPER programs were dramatically easier to debug. The simple undefined-variables checks, array-bounds checks and access checks were adequate to pin-point both coding and logic errors. Some of these checks even revealed errors in the original FORTRAN programs which had remained undetected for several months.

4. The VIPER programs were easy to test and commission because small test programs could easily be generated both to drive the programs, as well as to be driven by the program being tested i.e. respond to the stimuli issued by the program under test.

5. The programs were generally easy to write and the use of GOTO statements could be naturally avoided in most cases. The only akward feature in entering text, is the lack of a line editor. Many errors are of the single character type and a facility to edit a line without retyping all of it would be desirable; particularly the long lines occuring in multiple assignment statements. This editing facility has been added to a "relative" of PROSIC called ABAKUS (DU PLESSIS,1974) (ABAKUS was also derived from Varian BASIC) and could be added without difficulty to VIPER.

A final claim of this thesis is therefore that in VIPER, programs are easier to write, debug, read, test and document than they are in FORTRAN.

TABLE 6.1 HULETTS REFINERY SOFTWARE: SPACE REQUIREMENTS

VIPER			HP FORTRAN (RTE)					
Name	No Lines (1)	Size (Words) (1) (2)	Name	No Lines (1)	Size			
					Words (3) (RIE-2)	Pages (RTE-3)	Disc storage Source (4)	Disc storage Binary
							(In blocks of 64 words)	
1. Repetitive programme								
-			PACIR	17	323	2	8	2
SCANCS	29	640	SCCS	61	1 002	2	22	6
SCANADC	32	550	SCAD	60	798	2	34	6
ENGUNITS	60	1 155	ENGUN*	104	1 886	3	48	34
WATCH.DOG	37	593	WCHDG	45	1 201	3	50	7
SERVOTIP	40	760	SERVO*	121	5 328	7	74	17
SATFLOW	80	1 406	SAFCO	130	2 984	4	45	28
CLFLOW	48	899	CLFLO	88	2 633	4	46	10
REMELT	45	669	REMLT	60	1 466	3	27	10
LIMERATIO	53	1 020	CLIME	72	1 427	3	50	18
GASFLOWA	44	879	GASFA	62	1 717	3	46	9
GASFLOWB	44	879	GASFB	62	1 714	3	60	8
GASFLOWC	35	740	GASFC	64	1 592	3	34	7
FILTER.MONITOR	80	1 343	FILCY*	175	8 205	10	63	27
CDAC	5	128	-					
WCOUT	19	353	-					
MESSAGE	32	487	MESEG*	57	5 540	7	36	13
	683	12 501		1 178	32 276	60	766	199
2. Non-repetitive or infrequent programs								
STARTUP	16	340	STRUP*		4 666	6	52	11
SHUTDOWN	8	109	HANGO*		6 431	8	44	16
ENGLIMITS	15	365	-					
FILTERCOEF	12	265	-					
SERVOHOUR	16	257	-					
SERVO8HOUR	18	250	-					
FILTER.REPORT	72	999	RFLDT*		6 572	8	25	20
CLOOP	14	255	CLOOP*	34	2 138	4	32	6
	171	2 830			19 807	26	153	53
	683	12 501			32 276	60	766	199
	854	15 331			52 083	86	919	252
PRINT.MESSAGE	80	{						919
PRINT.PROG.NAME	11	{	DISC FILES				64 x	1 171
PRINT.CHAN.NAME	32	{					Words	74 944
3. Monitoring and service programs								
(Not required because of interactive facilites or included in operating system nucleus)			MONT		12 550	14		
			RCOMD	55	5 085	6		
			CAMEP	10	3 077	5		
			PRADC		3 328	5		
			WRDAC		3 547	5		
			LAMG2		3 462	5		
			HEAD		3 036	4		
					34 085	44		
					52 083	86		
					86 168	130		

NOTES

(1) Excluding comments.
(2) Including symbol table.
(3) Including non-reentrant library modules.
(4) Including comments.
* Functionaly equivalent but not comparable line-for-line.

TABLE 6.2 PROGRAM STATISTICS

A.	FORTRAN Programs
	1. Average length = 1 178/15 = 78,5 lines
	2. = 32 276/15 = 2 151 words
	3. = 60/15 = 4 pages
	4. Average words/line of code = 32 776/1 178 = 27,4 words
B.	VIPER Programs
	1. Average length = 854/24 = 35,6 lines
	2. Average length = 15 331/24 = 638 words
	3. Average words/line of code = 15 331/854 = 17,9 words
	4. Average length of descriptor table (direct measurement) = 178 words

TABLE 6.3 COMMON REQUIREMENTS

A.	FORTRAN			
	Global COMMON = 758 words			
	(See Case Study programs Appendix B.3 for description)			
B.	VIPER			
	SPECS	3		
	VOLTS	30		150 x 2 = 300
	ENG	30	Segment descriptor Overhead (Fig. 3.5(b))	6 x 15 = 90
	BITS	6		390
	ENGLIM	60		
	SERVOD	16		
	GASFLOW	5		
		150		

TABLE 6.4 HULETTS REFINERY SOFTWARE: SPEED

(a) FORTRAN PROGRAMS (HP RTE FORTRAN 92060-16092 Rev 1726)

Availability of partitions with all control programs listed in (b) running.

Comment		% Availability of Partition				% CPU Available
	Partition No.	1	2	3	4	
	Size K words	2	4	8	15	
1.	Simulates RTE 2, low priority BG	N	N	84,3	13,5	97,8
2.	Simulates RTE 2, high priority BG	N	N	0	98,4	98,4
3.	PACIR, SCCS and SCAD in partition 1	A	N	84,9	13,1	98,0
4.	Some programs in partition 2	N	A	85,2	12,7	97,9
5.	All partitions available	A	A	85,6	11,9	97,5

Notes: N - Partition not available (locked).

A - Partion available but actual time available not measured.

BG - Background.

All figures averaged over 5 minutes.

Average time CPU busy ≃ 2%.

Average time real-time partitions busy ≃ 15%.

TABLE 6.4(b)

VIPER PROGRAMS

Program	Execution Time millisecs	Number of executions/ minute	Computation Time/Minute Secs
SCANCS	75	10	0,75
SCANADC	1 410	2	2,82
ENGUNITS	790	2	1,58
WATCH.DOG	160	2	0,32
SERVOTIP	670	1,3*	0,89
SATFLOW	172	2	0,34
CLFLOW	105	2	0,21
REMELT	76	2	0,15
LIMERATIO	128	2	0,27
GASFLOWA	106	2	0,21
GASFLOWB	106	2	0,21
GASFLOWC	86	2	0,17
% Time busy in 60 sec sample time			7,92 secs 13,2 %

*Statistical weighting factor, all others deterministic.

RATIOS

1. $\dfrac{\text{Average time CPU busy in VIPER}}{\text{Average time CPU busy in RTE FORTRAN}} \simeq \dfrac{13,2}{2} = 6,6$

2. $\dfrac{\text{Average time CPU busy in VIPER}}{\text{Average time real-time partitions occupied in RTE}} \simeq \dfrac{13,2}{15} = 0,88$

CHAPTER 7

LIMITATIONS AND EXTENSIONS

7.1 LIMITATIONS

In addition to the particular ommissions from VIPER which were listed in section 2.7, there are three more fundamental limitations which affect real-time interactive systems using software virtual memory management.

7.1.1 Dynamic relocation

The software virtual memory management algorithms described in this thesis require that the segments of code be dynamically relocatably to any position in memory. To meet this stipulation with reasonable efficiency only relative address references can be used in the code, all other referencing being performed indirectly via specially constructed linking elements (descriptors). The use of interpretive code was proposed as the simplest method of meeting this requirement as appropriate meta-code structures can be devised which meet the relocation and relative addressing conditions.

To enable in-line (compiled) code to be used in a software virtual memory management system would require special order codes which would have to be provided by microprogramming if the actual instruction set was not suitable. (Certain machines do have codes which are relocatable e.g. Data General NOVA 2/3, provided certain coding restrictions are accepted.) If the same protection functions are required, however, either a time or space overhead must be incurred. The protection functions must either be provided by in-line code (requiring more space) or by out-of-line calls to subroutines, which is essentially what an interpreter does.

Two other problems which must be considered when using this machine code approach are:

1. Addressing of data items in shared data areas and parameter linking.

2. Decompilation of the machine code to recreate the source listing. This has been reported to have been done in one system (WILKINS, 1976) but no details of the algorithm have been published. De= compilation from machine code would also only appear possible on certain machines. (BROWN P., 1977)

The Varian 620i on which all the development work on VIPER was performed does not have a suitable instruction set for this purpose and is not microprogrammable, so this approach was not considered in any detail. With the microprogrammable MICROV now available these techniques are receiving reconsideration.

7.1.2 Swapping rate

The space allocation and dynamic linking operations in software virtual memory management are an order of magnitude slower than similar hard= ware virtual memory mapping devices. In many applications this does not significantly affect the performance of an SVMM system because most of the repetitive or critical tasks will be permanently resident in memory, but an SVMM system can clearly not support as high a swapping rate as a hardware memory management system.

Some alternative structures which may reduce the swapping overhead were discussed in section 3.4. These structures may permit a higher swapping rate to be tolerated with reasonable overhead, but the SVMM system will nevertheless generally still be significantly less efficient.

SVMM therefore cannot be said to compete with hardware virtual memory management; what it does achieve is to enable the advantages of virtual memory to be provided or small systems at low cost and without requiring special purpose hardware.

7.1.3 Performance limitations

The mapping operation which is performed on every reference to a

variable ...../7.3

variable together with the protection functions which are regarded as an intrinsic part of SVMM, limit the ultimate performance which can be attained in a system which uses SVMM. This phenomena was documented in Table 5.3 where it was shown that as the overhead associated with the interpreter process is reduced, the relative time spent performing the mapping and protection functions increases. The times shown in the last column of Table 5.3 for the "streamlined" version could possibly be reduced further by in-line code expansion in the interpreter (rather than using subroutine calls), but there is still a limit beyond which the mapping operation overhead will be dominant. This is clearly an intrinsic limitation of SVMM which can only be overcome by hardware memory management systems. As the results of the preceding two chapters have shown however, SVMM systems are still capable of excellent performance in the small processor domain, and can be improved further before this intrinsic mapping limit becomes significant.

7.2 EXTENSIONS

The concept of Software Virtual Memory Management has shown itself to be a powerful tool for constructing a flexible interactive software system. The interpretive mode of execution used contributes strongly to the attractive interactive features and it would be desirable to maintain this mode of execution while improving the performance of the system. There are eight possible ways in which the performance of a system like VIPER could be improved without sacrificing the interactive and protection facilities.

7.2.1 Floating point firmware

This simple hardware improvement was discussed in chapter 5 where it was estimated that it gives a 2 to 1 improvement in speed. A further advantage of floating point firmware or hardware is the memory space that is saved. Moving the basic functions add, subtract, multiply and divide, and conversion functions to and from integer and floating point, would save nearly 1 000 words of local memory space which would then be released for virtual memory operations. Placing additional routines such as trigonometic, log, exponential and square root functions etc. into firmware would save another 1 000 words besides improving the performance.

7.2.2 Polish notation

The advantage of using the Polish notation was discussed in section 5.1.2 and this is an extension which should be used in all interpretive systems. The disadvantage of more complex decompilation algorithms is offset by the simplification of the actual interpretive or evaluation section. The use of the Polish notation has two advantages: firstly the time to execute statements is considerably reduced, and secondly more compact representations of the internal code can be formulated. An example showing the difference between the infix and Polish forms was shown in Table 2.8. This compact representation would halve the size of the code portion of a segment.

7.2.3 Alternative procedure segment structures

If the size of the code portion of a segment were to be reduced by using the compact Polish form noted above, the symbol table partition of the segment would tend to become a major component of the overall segment size. As the ASCII representation of the symbol table elements is only required during interactive operations, the size of the table could be significantly reduced by maintaining separate seg= ments for the variable data values and for their ASCII names. This is analogous to the problem of space occupied by comments which was noted in section 5.1.2. They also should be kept in a separate segment so that if the local memory is full, all information which is super= fluous to the execution of segments can be swapped out of memory. Additional information which is not required in the normal execution of segments (or which can be eliminated by suitably restructuring the code) is the statement number, length and type.

These considerations lead to a proposal for an alternative segment structure which is shown in Fig. 7.1. The procedure segment is split up into four separate segments, one for the variable table + code, and one for each of the symbol table, statement numbers and comments. (The statement number and comment segments could possibly be combined.) As shown in Fig. 7.1, this structure is combined

with ...../7.5

with the use of a segment number identifier and segment directory, as was discussed in section 3.4. Some problems relating to access to shared data segments must still be solved using this structure, but these would not appear to be insurmountable.

The size of the remaining code + variable portion of the segment using this structure would be less than half of the space required by the segment using the current monolithic segment organization. This is a significant advantage in real-time applications, as the smaller the modules are, the bigger the "working-set" of real-time tasks can be. This permits larger and more complex tasks to be handled than would otherwise be possible. Although an arbitarily large set of tasks can theoretically be run in a virtual memory system, if a "working set" of modules cannot fit into memory, the high swapping rate and thrashing of modules to and from bulk store that will result, will seriously degrade the performance of the system. (DENNING 1974). In a real-time system the "working set" may be defined as the set of tasks (or modules within those tasks) which execute repetitively or frequently in response to external events. If all these tasks can fit into memory the system will be capable of achieving a significantly higher performance. This effect was demonstrated in the results of the case study.

7.2.4 Operating system kernel

One of the specific objectives of software virtual memory management was the avoidance of hardware memory mapping devices. On most current (or forseeable) mini and micro-computers this limits the local memory addressing space to 32K 16 bit words (64K bytes). In VIPER all the operating system code is kept permanently memory resident with only a few segments being used for system data storage operations.

This results in a maximum of 18 to 19K words being available for virtual memory operations. Furthermore the addition of new functions and drivers to the operating system will steadily decrease the memory available. Many of the modules which are now memory resident are used relatively infrequently and could reside on a bulk storage device most

of ...../7.6

of the time without noticably affecting the performance of the system. Modules in this category are the lexical and syntactical scanner, decompilation (listing) programs, directory manipulation routines and system documentation functions. By keeping those routines out of the resident operating system nucleus 3 000 to 4 000 words of memory could be saved, reducing the size of the resident code to 8K words or less if extensions 7.2.1 and 7.2.2 were also implemented.

These infrequently used modules could be swapped into memory into the fixed segment areas which were indicated in Fig. 3.1. The important point is that these areas could be allocated dynamically, and no area or partition need be permanently allocated for their use. This is in marked contrast with most minicomputer real-time executives where the memory is divided into fixed partitions which can only be changed at system generation time. As an example of this type of allocation consider the memory division employed in Hewlett Packard's RTE. A fixed background partition is provided which consumes 10 to 16K, but which is only used a small proportion of the time in a typical process control system. All the critical real-time tasks are forced to swap in and out of one (or a few) foreground partitions.

The resident code which remains after stripping off the in= frequently used functions can also be further subdivided into two or more levels. At the innermost level would be a small operating system kernel which implements the basic operating system functions such as interrupt handling and synchronization. At the next level, more sophisticated operating system functions are provided such as scheduling and memory management. The basic interpreter functions could be provided on a yet higher level together with the SVMM functions.

The use of a kernel has distinct advantages as far as the reliability and maintenance of the operating system is concerned. More than one level of kernel is in fact desirable in this respect, as a number of recent systems have shown that a modular system with appropriate layers of software built upon an innermost kernel is significantly more reliable and is easier to expand and maintain

(BAYER, .../7.7

(BAYER, 1975; CHALMERS, 1976; MARK, 1977; VOJNOVIC, 1977).

Further advantages noted by these authors when using a compact inner kernel, are firstly, that all the outer layers can be written in a high level language, enabling a measure of portability to be achieved, and secondly, that the kernel can be implemented in micro=code providing a very efficient realization of the essential and most frequently used operating system functions.

Incorporating these concepts into an implementation of VIPER would enable an efficient, compact and portable operating system to be constructed.

7.2.5 Multi-language

One of the limitations of VIPER as implemented in this thesis, is that it cannot support more than one language for on-line interactive operations. It should be desirable to extend the interactive and pro=tection facilities to enable them to be used in other more standard or conventional languages. As the information required for these operations is for the most part contained within the descriptor tables and not within the body of the code, it is theoretically possible to extend the facilities to other languages. The basic requirement would be for the same descriptor (symbol) table format to be used.

Two other practical requirements would also need to be met. The syntax scanner and decompilation routines for an additional language could not be kept memory resident and an essential requirement of a multi-language system would be the implementation of the modular kernel approach with the language processing modules being swapped in as needed. A second requirement would be that the internal meta-codes which were used would need to be language independent (otherwise two different interpreters would be required). The actual meta-codes would also have to be selected to have some of the general characteristics of machine code while retaining the properties required for the SVMM operations. ADIX (1975) and HELPS (1974) have shown that meta-codes with this dual general-plus-special purpose characteristic can be constructed for particular applications. Unsurmountable difficulties may however be encountered in attempting to use more complex languages such as PASCAL in the SVMM environment.

7.2.6 ...../7.8

7.2.6 "Throw-away" compiling (BROWN P. 1976; HAMMOND,1977)

"Throw-away" compiling was mentioned briefly in section 3.5.4. In this middle-path between interpretation and compilation, each statement of a procedure is dynamically compiled just before it is executed the first time. If each statement in a procedure is executed only once, throw away compiling is slower than inter= pretation, but if, as is frequently the case, the program spends a significant proportion of its time in one or more loops, then the compiled code which has accumulated for these loops will execute much faster. The term throw-away derives from the fact that when memory space is short or when any interactive operations take place, all the compiled code is thrown away and compilation is begun anew. An essential requirement for tolerable efficiency with this approach is the storing of the interpretive meta-code in Polish form to ensure that the code generation step can be performed quickly.

This technique is of interest to systems such as VIPER because of the repetitive nature of many tasks. It was noted in the case study and elsewhere that in smaller systems some of these tasks are likely to remain resident in memory. If they remain resident, however, then they could be executing in-line compiled code instead of inter= pretive meta-codes. This would enable the repetitive or time consuming tasks to execute faster and hence improve the performance of the system. The only disadvantage of this approach is that the compiled code generally takes more space, so that converting tasks from interpretive to in-line code will in general reduce the memory available for other tasks.

7.2.7 Microcoding

In addition to the microcoding of the floating point operations and possibly of an operating system kernel, some of the interpretive functions themselves can be micro-coded. GAINES (1976) has reported a 10 to 15 fold improvement in execution time of a BASIC-like system using less than a 1 000 words of microcode.

There ...../7.9

There are two approaches that can be used when using microcoded functions. The first is to retain the basic implementation of the inter= preter in Assembler but to place certain of the mapping and specialised search and move operations in microcode. This is essentially an extension of the concept of using floating point firmware.

The second approach is to use the microcode to implement a pseudo-machine which executes the interpretive meta-codes directly. A difficulty which arises from this approach is that the order codes and addressing structures required for the interpretive mode generally do not coincide with that of the host machine. To enable the full speed and space advantage of the interpretive code to be realised, architectural changes may therefore be necessary to enable the two different types of code to be executed on the same hardware. It is not simply a matter of providing a new set of functions in a control store (writable or otherwise) as it is the actual order codes themselves which are different.

It can be argued that if architectural changes are required, it may be more profitable to implement the virtual memory management functions in hardware and to return to a compiler oriented system. The advantage of retaining the interpretive mode of operation together with SVMM, however, is that no major operating system or language changes are required in order to enable a micro-coded implementation to be used. The advantage of portability would, in particular, be retained as the same meta-code could be executed on two different machines; in the one case via a normal interpreter and in the other by direct emulation in micro-code. In other words, the use of special hardware on one machine to obtain a particular speed advantage would not preclude the use of the language and operating system concepts on another machine with a different architecture. It is this BASIC-like portability that is an attractive advantage of SVMM, a portability which can be complemented by microcoding techniques.

7.2.8 Multicomputer operation

A further extension of VIPER which is being studied is the use of multiple processing elements. There are two aspects to this study, the first relating

...../7.10

relating to multi-processor systems and the second to multicomputer systems or computer networks. It has been pointed out by BORGERSON[37] that single-language systems such as BASIC and APL are particularly suitable for the implementation of multi-processor systems because it is possible to utilize one section of re-entrant code for the language processing which is operated on by multiple processors. The allocation of processors to tasks is a non-trivial problem, but the well-defined task partitioning that occurs in VIPER can help to reduce the magnitude of this problem.

The second aspect of multiple processor use occurs in multi-computer systems or computer networks. The properties of the SVMM-system permit the meta-code segments and data to be transmitted from one computer to another for execution on that machine. The processors in the system can differ, provided only that each is capable of evaluating the meta-codes by interpretation or micro-coding. In this environment, a task consisting of one or more segments can be executed on any element of the network without any modification or link-loading. This concept of 'packet-switching' of segments of tasks (as opposed to merely data) between elements of a multi-computer system is a unique property of SVMM which it is planned to use to advantage. To facilitate the movement of segments, it was desirable that all the information associated with a segment should be contained in a physically contiguous block, and this consideration influenced the segment structure that was chosen.

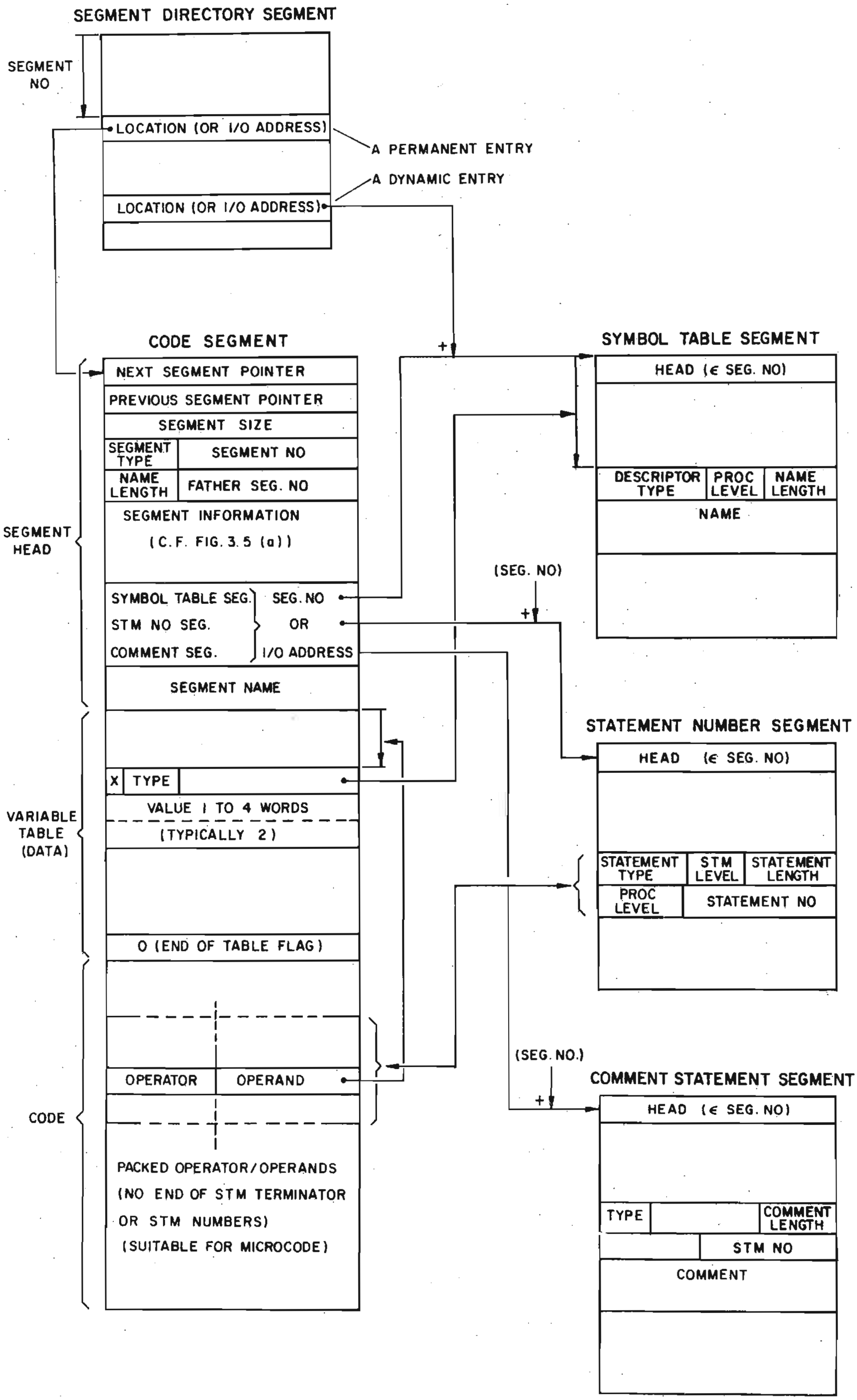


FIGURE 7.1 ALTERNATIVE PROCEDURE SEGMENT STRUCTURE

CHAPTER 8

CONCLUSION

Interactive real-time software systems, consisting of the amalgamation of a high level language and a simple operating system, are an important class of software which have been widely used in a variety of applications. It is claimed, however, that the structure and performance of this type of system needs to be enhanced to enable improved programming methods to be used and to enable more complex programming tasks to be undertaken by the application oriented user.

The goal of this thesis was therefore to demonstrate that the interactive facilities of such software systems could be extended and improved, using a structured language in a multiprogramming and multi-user environment, while retaining the ability to run on simple, small, minicomputer or microprocessor systems. An additional goal was to maintain the simplicity of operation and construction, while improving the protection facilities, as well as to demon= strate that good programming practices are possible on systems of this type.

In constructing a system to meet these goals, serious memory management problems had to be solved. This led to the development of the concept of "Software Virtual Memory Management" (SVMM); a memory management technique which extended the concept of hardware virtual memory management without requiring the use of hardware mapping devices. In addition to extending the effective memory space of the system, this memory management system facilitated the provision of a variety of protection functions.

In developing the operating system VIPER, which uses SVMM techniques, it is claimed that the above goals were attained, and that the following concepts were demonstrated:

1. The interactive facilities found in simple monoprogrammed systems can be extended and improved in multiprogramming systems. Both the interior and exterior (shared) data structures of a procedure can be examined while the procedure is executing, using normal program statements and commands. As far as I am aware, this is a

unique property of VIPER and has not been implemented on any other system.

2. Structured programming concepts can be simply implemented and the memory management algorithms can take advantage of the modular properties of structured programs.

3. The efficient way in which memory is used in SVMM improves the performance of interpretive systems by permitting many more programs to reside resident in memory. This reduces, or eliminates the need for swapping, resulting in the performance of the inter= pretive system being comparable to that of a system executing in-line code with swapping in typical applications.

4. The unification of the command and programming languages, and the use of the same language elements for debugging operations, simplifies the user interface. This facilitates the use of the system by application oriented users with minimal training in real-time operating system concepts. The SVMM structures also contribute to this simplicity by integrating the text manipulation and protection functions.

5. The SVMM structures permit protection facilities to be naturally incorporated at all levels in the system, including parameter passing, data segment access and the file-system-like protection of program modules. The integration of the protection functions into the language and operating system also simplifies these operations and encourages the use of the protection facilities by the application oriented user.

6. The documentation aids which can be provided in the interactive language contribute to the production of programs which are readable and maintainable. These include the structured programming indenting, the end-of-line comments and the system documentation aids.

In ...../8.3

In the implementation of SVMM in VIPER, the simplest structures and algorithms were employed which enabled these concepts to be demonstrated. As noted in chapter 3 improvements could quite likely be made to the memory allocation and scheduling algorithms. Alternative memory structures could also be investigated, as discussed in chapter 7. Despite this simplicity of construction, the performance of VIPER is considerably better than that of many simple real-time BASICs which are currently available, and systems using SVMM could be applied to applications where interpreters could not previously be used.

In the process control case study, for example, it was observed that VIPER had a performance which was comparable to that of a compiler oriented system executing in-line code with swapping. It is not claimed, however, that an interpretive system like VIPER competes with these compiler-orientel real-time executives in all applications. VIPER is a dedicated, high-level-language system, whereas these latter executives are general purpose multi-language systems. What is claimed is that in many applications the full facilities of these executives are not used. In these cases SVMM and an inter= preter can provide an attractive solution which simplifies the programming task and which facilitates the production of more reliable software.

VIPER was designed primarily as an interactive software tool for experimental process control work. A final claim of this thesis, however, is that the concept of Software Virtual Memory Management is of wider applicability. Business processing applications, for example, such as those described by GAINES (1976) and FULTON (1976) as well as distributed instrumentation systems (RAIMONDI, 1976; AGRAWAL, 1976; DIEHL, 1975; ANFALT, 1975; VON MEURS, 1977) could all use SVMM concepts to advantage. The numerous simple interpretive process control systems which have been reported (FOSTER, 1974; OTTO, 1974; LAURENCE, 1975; NELSON, 1976; GLADNEY, 1976; BERCHE, 1976) could also use the SVMM type structures to improve the program structure and interactive facilities, as even these simple systems suffer from shortcommings in one or other of these areas.

Furthermore, the extensions and improvements which can be implemented (as discussed in chapter 7) can be used to overcome some of the current limitations of the SVMM implementation in VIPER. This would facilitate the application of SVMM concepts to an even wider class of applications and could be used to eliminate the dependence on software interpreters. Software Virtual Memory Management is therefore a powerful technique for constructing real-time inter= active software systems on mini- and microcomputers.

CHAPTER 9

REFERENCES

ADIX M.S. and SCHULTZ H.A., "Interpretive execution of real time control applications", General Motors Research Laboratories, Warren, Mich. Research Publication GM-2014, November 1975.

AGRAWALA A.K. and BRYANT R.M., "Models of memory scheduling", SYMPOSIUM ON OPERATING SYSTEM PRINCIPLES, Nov. 19-21, 1975, pp 217-222.

AGRAWAL A.K. and BUTLER E.M., "Software considerations of a minicomputer based multi-instrument CAMAC automation system", IEEE TRANSACTIONS ON NUCLEAR SCIENCE, Vol. NS-23, pp 462-466, February 1976.

ANDREWS G.R. and MCGRAW J.R., "Language features for process interaction", ACM SIGPLAN NOTICES, Vol 12, No. 13, pp 114-127, March 1977.

ANFALT T, and JAGNER D., "A PDP 11 computer system for the multiterminal processing of several analytical instruments in an interpretive language", ANALYTICAL CHEMISTRY, Vol. 47, No. 4, pp 759-761, April 1975.

ANSI: American National Standard For Minimal BASIC, ANSI X3J2/76/01.

ARDEN B.W., "Interactive computing", PROC. OF IEEE, Vol. 63, No. 6, pp 836-843, June 1975(a).

ARDEN B.W. and BERENBAUM A.D., "A multi-microprocessor computer system archirecture, 5TH SYMP. ON OPERATING SYSTEMS PRINCIPLES, pp 114-21, November 19-21, 1975(b).

AIRD G.N., "Fredette's operating system interface language (FOSIL)" IFIP WORKING CONF. ON COMMAND LANGUAGES, Lund, Sweden. pp 267-280, July 1974.

BANIN R.A. and BURN R.D. JR, "Real-time data aquisition and process control in a distributed computing network", NATIONAL TELECOMMUNICATIONS CONF. NEW ORLEANS, Vol. 1, pp 24, 14-17, Dec. 1975.

BARNES J.G.P. and SAGE M.W., "Real time computing - now and in the future", IFAC WORLD CONGRESS, Paper 27.2, Boston 1975.

BARTH C.W., "Notes on the CASE statement", SOFTWARE PRACTICE AND EXPERIENCE, Vol. 4, pp 289-298, 1974.

BAYER D.L. and LYCKLAMA H., "MERT - A multi-environment real-time operating system", 5TH SYMP. ON OPERATING SYSTEM PRINCIPLES, pp 33-42, 19-21 Nov. 1975.

BERCHE S., HEBENSTREIT J. and NOYELLE Y., "LST: A highly adaptive real-time time-sharing system", 1st IFAC/IFIP SYMP. ON SOFTWARE FOR COMPUTER CONTROL, Tallin, USSR. pp 289-294, May 1976.

BIACHI G. and LEWIS A., "A pragmatic approach to high-level real-time language", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Roquencourt, pp 87-96, June 1976.

BHAT P.V., "Reliability analysis and backup for process computers", ISA CONF. ADVANCES IN INSTRUMENTATION, Vol. 31, pp 534/1-8, 1976.

BORGENSON B.R., "The viability of multimicroprocessor systems", COMPUTER, Vol. 9, No. 1, pp 26-31, January 1976.

BRINCH HANSEN P., "Operating system principles", PRENTICE HALL, 1973.

BRISTOL E.H., "A design study of a simple conversational language system", ASME, 76-WA/AUT-11, 1976.

BROWN GEORGE E., ECKHOUSE R.H. JR. and GOLDBERG R.P., "Operating system enhancement through microprocessing", 13TH IEEE COMPUTER SOCIETY INTERNATIONAL CONF. September 1976.

BROWN P.J., "Recreation of source code from reverse polish form", SOFTWARE - PRACTICE AND EXPERIENCE, Vol. 2, No. 3, pp 275-278, 1972.

BROWN P.J., "More on the recreation of source code from reverse polish", Papers in compiling, 6, Computing Laboratory, University of Kent at Canterbury, November 1976(a).

BROWN P.J., "Throw-away compiling - a middle path between interpretation and compilation", PROGRAM OPTIMISATION, Infotec, Maidenhead, pp 153-160, 1976(b).

BROWN P., Private communication. Department of Computer Science, University of Kent at Canterbury, June 1977.

BURSTALL R.M., COLLINS J.S. and POPPELSTONE R.J., "Programming in POP-2", Edinburgh University Press, 1971.

BUZEN J.B., "I/O subsystem architecture", Proc. of IEEE, Vol. 63, No. 6, pp 871-878, June 1975.

CAINE S.H. and KENT GORDON E., "PDL - A tool for software design", AFIPS National computer conf., vol. 44, Anaheim, pp 27-276, May 1975.

CARTWRIGHT L.M. and MAYBERRY J.C., "Distributed systems - a new approach to process control", ADVANCES IN INSTRUMENTATION, Vol. 30, pp 511: 1-8, 1976.

CARY T., "Structuring BASIC programs for managing overlays in a small computer system", ACM SIGPLAN NOTICES, Vol. 11, pp 88-93, April 1976.

CHALMERS A.F., "Operating system design for real-time control", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, pp 203-216, Roquencourt, 1976.

CHARLTON ...../9.3

CHARLTON C.C. and HIBBARD P.G., "A note on re-creating source code from the reverse polish form", SOFTWARE - PRACTICE AND EXPERIENCE, Vol. 3, No. 2, pp 151-154, 1973.

CHU Y. and CANNON E.R., "Interactive high-level language direct-execution microprocessor system", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, Vol. SE-2, No. 2, pp 126-134, June 1976.

CLAGGETT E., "Interpreters versus compilers for on-line microcomputers", CONTROL ENGINEERING, pp 24-27, August 1977.

CLYNICK P.R.R., "Developing real-time programs", COMPUTING WITH REAL-TIME SYSTEMS, PROC. OF 1ST EUROPEAN SEMINAR, Vol. 1, Transcripta, London 1972.

COSSERAT D.C., "A data model based on the capability protection mechanism", Revue Francaise D'automatique, INFORMATIQUE ET RECHERCHE OPERATIONNELLE, B-3, pp 63-78, September 1975.

CUFF R.N., "A conversational compiler for full PL/1", COMPUTER JOURNAL, Vol. 15, p 99, 1972.

CUNNINGHAM R.J. and PUGH C.G., "A language independent system to aid the development of structured programs", SOFTWARE - PRACTICE AND EXPERIENCE, Vol. 6, pp 487-503, 1976.

CURTIN R., HACK H. and SAUNDER J., "Analysis of a BASIC interpreter", Dept. of Computer Science, University of Cape Town, 1975.

DAGLESS E.L., "A multimicroprocessor: CYBA-M", paper, IFIP CONGRESS, Toronto, August 1977.

DAHL O.J., DIJKSTRA E.W. and HOARE C.A.R., "Structured programming", Academic Press, London 1972.

DAKIN R.J. and POOLE P.C., "A mixed code approach", COMPUTER JOURNAL, Vol. 16, No. 3, pp 219-222, 1973.

DANIERI J.J., "Principles of some debugging tools for large real-time systems and perspectives of evolution", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Roquencourt, pp 271-289, June 1976.

DASAI T., IHARDA K., SUNDHARA K. and NAKAMURA T., "High level process control language 'ESPIRIT' and its source level debugging system "Solda", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Eindhoven, June 1977.

DAWSON J.L., "Combining interpretive code with machine code", Computer Journal, Vol. 16, No. 3, pp 216-218, 1973.

DE BALBINE G., "Better manpower utilization using automatic restructuring", AFIPS NATIONAL COMPUTER CONF., vol. 44, Anaheim, pp 319-327, May 1975.

DEMILLO ...../9.4

DEMILLO R.A., EISENSTAT S.C. and LIPTON R.J., "Can structured programs be efficient", SIGPLAN - Notices, Vol. 11, pp 10-18, October 1976.

DENNING P.J., "Virtual memory", Computing surveys, Vol. 10, pp 153-189, September 1970.

DIEHL W. and SANDERS D., "Real-time basic used in a distributed network", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Boston pp 159-163, August 1975.

DIEHL W., "Software for industrial computer control - a review", 1ST IFAC/ IFIP SYMP. ON SOFTWARE FOR COMPUTER CONTROL, pp 279-285, Tallin, USSR, May 1976.

DIJKSTRA E.W., "The structure of 'THE' multiprogramming system", COMM. OF THE ACM, Vol. 11, No. 5, pp 341-346 May 1968(a).

DIJKSTRA E.W., "Co-operating sequential processes", In: programming languages, F. Genuys (Ed.), Academic Press, pp 43-112, 1968(b).

DIJKSTRA E.W., "The humble programmer", COMM. OF THE ACM. Vol. 15, No. 10, pp 859-866, October 1972.

DRIESTROWSKI A., "Session 1 general discussion", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Boston, pp 26 and 90.

DU PLESSIS J.J., "ABAKUS", Departement Elektrotegniese Ingenieurswese, Universiteit van Stellenbosch.

ECMA: Standard for minimal basic, 1976.

ELZER P., "Facilities for a real-time language", COMPUTING WITH REAL-TIME SYSTEMS, Proc. of 1st European seminar, Vol. 1, Transcripta. London 1972.

ELZER P. and ROESSLER R., "Real time languages and operating systems" 5TH IFAC/IFIP DIGITAL COMPUTER APPLICATIONS TO PROCESS CONTROL, The Hague, pp 1-12, 1977.

ESONE, "Real-time basic for CAMAC", ESONE COMMITTEE ESONE/RTB/12, April 1977.

FABRY R.S., "The case for capability based computers", 4TH SYMP. ON OPERATING PRINCIPLES, New York, October 1973.

FOSTER H.R. and CAPO J.O., "A real-time basic control system", ISA TRANSACTIONS, Vol. 12, pp 314-322, 1973.

FRAADE D.J. and GAST S., "A survey of computer networks and distributed control", 5TH IFAC/IFIP DIGITAL COMPUTER APPLICATIONS TO PROCESS CONTROL, The Hague, pp 13-29, 1977.

FULTON D.L. and THOMAS R.T., "Abacus/x - an incremental compiler for mini= computer use", ACM SIGPLAN NOTICES, Vol. 11, pp 119-126, April 1976.

FULTON D.L. and THOMAS R.T., "A minicomputer-based information system for a small business", COMPUTER, Vol. 9, pp 24-28, September 1976.

GAINES B.R., FACEY P.V. and SAMS J.B.S., "Minicomputers in security dealing", COMPUTER, Vol. 9, pp 6-15, September 1976.

GAINES B.R. and FACEY P.V., "Some experience in interactive system development and application", PROC. OF IEEE, Vol. 63, No. 6, pp 894-911, June 1975.

GALLIER P.W., "Surveying process digital computer control", IEEE INTERNATIONAL CONVENTION RECORD, Vol. 13, PT 6, pp 2-9, 1965.

GANS M., "Fast basic manual", Digital processes laboratory, Department of Electrical Engineering, University of Natal, 1975.

GERTLER J. and SEDLAK J., "Software for process control - a survey", AUTOMATICA, Vol. 1, pp 613-625, 1975.

GILB T. , "Documentation by computer", DATA MANAGEMENT, Vol. 13, p 17, March 1975.

GLADNEY H.M. and HOCHWELLER G., "Multiprogramming for real-time applications", 3RD IEEE SYMP. ON COMPUTER ARCHITECTURE, Clearwater, January 1976.

GLASS R.L., "Splinter - A PL/1 interpreter emphasising debugging capability", COMPUTER BULLETIN, pp 180-185, September 1968.

GOODENOUGH J.B., "Exception handling issued and a proposed notation", COMM. OF THE ACM, Vol. 18, No. 12, pp 683-696, 1975.

GORD E.P. and MAHON M.J., "Module interface description language", EUROPEAN COMPUTER CONFERENCE ON SOFTWARE SYSTEMS ENGINEERING, pp 1-14, September 1976.

GORDON-CLARK M.R., IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, ROQUENCOURT, pp 331-339, June 1976.

GORDON-CLARK M.R., "Session 1 general discussion", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Boston, pp 26 and 91, August 1975.

GOUWS J.H., "User's notes on the process BASIC interpreter", Internal report Ea-41, National Electrical Engineering Research Institute, Pretoria, June 1973.

GOULD I.H., "Interactive debugging system for a multiprogrammed minicomputer", EUROPEAN COMPUTER CONF. ON INTERACTIVE SYSTEMS, London, September 1975. pp 451-464.

GOYER P. and DU MASLE J., "Some basic notions for computer network command languages", IFIP WORKING CONF. ON COMMAND LANGUAGES, Lund, Sweden, July 1974, pp 327-333.

GRAM C. and HERTWECK F.R., "Command languages: design considerations and basic concepts", IFIP WORKING CONF. ON COMMAND LANGUAGES, Lund Sweden, July 1974, pp 43-69.

GRIEM P.D. JR., "On the principle of unique definition", AFIPS NATIONAL COMPUTER CONF. Vol. 44, Anaheim, pp 265-270, May 1975.

GUARDIA M.F. DE LA, and FIELD J.A., "A high level language oriented multi= processor", IEEE INTERNATIONAL CONF. ON PARALLEL PROCESSING, August 1976. pp 256-262.

GURSKI A.F., "Towards a high-level job control language", PROGRAMMING SYMPOSIUM, Paris, B. Robinet (Ed.), April 9-11, pp 130-140, Springer verlag, 1974.

HAASE V., "Facilities required in operating systems for real-time language programming", PROC. OF 1ST EUROPEAN SEMINAR, Transcripta, London 1972. pp 39-43.

HAASE V.M., "Evaluation of BASIC as a programming language for real-time systems", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Roquencourt, pp 331-339, June 1976.

HAMBLIN C.L., "Translation to and from polish notation", COMPUTER JOURNAL, Vol. 5, No. 3, pp 210-213, 1962.

HAMMOND J., "BASIC - an evaluation of processing methods and a study of some programs", Computing laboratory, University of Kent at Canterbury, 1977.

HEHER A.D., "PROSIC 2 user's guide", South African Council for Scientific and Industrial Research, Pretoria, Special report, Elek 69, April 1975.

HEHER A.D., "Some features of a real-time BASIC executive", SOFTWARE - PRACTICE AND EXPERIENCE, Vol. 6, pp 387-391, 1976(a).

HEHER A.D., "PROSIC: A real-time BASIC executive", SOUTH AFRICAN COUNCIL FOR AUTOMATION AND COMPUTATION SYMPOSIUM ON MINICOMPUTERS AND MICRO= PROCESSORS, Durban, pp 56-65, April 1976(b).

HEHER A.D., "A real-time interactive software system for process modelling, optimization and control", 5TH IFAC/IFIP DIGITAL COMPUTER APPLICATIONS TO PROCESS CONTROL, The Hague, pp 647-653, June 1977(a).

HEHER A.D., "Refinery computer control project: software organisation and CAMAC drivers", South African Council for Scientific and Industrial Research, Special report Elek 130, September 1977(b).

HEHER A.D., "Software virtual memory management in a real-time interactive software system", submitted to: IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, 1978.

HELPS K.A., "Compact interpreters - their implication on software and hardware design", REAL-TIME COMPUTER BASED SYSTEMS, AGARD conf. Proceeding No. 149, pp 12-1 to 12-9, 1974.

HENDERSON P. and SNOWDON R.A., "A tool for structured program development", IFIPS CONGRESS 1974. pp 204-207.

HEWLETT-PACKARD REAL TIME BASIC MEASUREMENT AND CONTROL SYSTEM, HP 9601B, Hewlett Packard Sunnyvale, California, 15 March 1974.

HILDEN J., "Guidelines for the design of interactive systems", COMPUTER JOURNAL, pp 144-150, May 1976.

HOARE C.A.R., "MONITORS: an operating system structuring system", COMMUNICATIONS OF THE ACM, Vol. 17, pp 549-557, 1974.

HOARE C.A.R., "Hints on programming language design", Real-time system implementation languages, U.S. Army Research and Development Group (Europe), Tech. Report ERO-2-75, Vol. 2, pp 505-534, April 1975(a).

HOARE C.A.R., "Data reliability", IEEE INTERNATIONAL CONFERENCE ON RELIABLE SOFTWARE", New York, pp 528-533, 1975(b).

HOLT R.C., "Structure of computer programs", PROC. OF IEEE, Vol. 63, No. 6, pp 879-893, June 1975.

HOOGENDOORN C.H. "The implementation of PHAST on MIKROV", South African Council for Scientific and Industrial Research, Report NIDR 78/21, Pretoria, March 1978.

HUNT K.L., "Real time operating systems", REAL-TIME COMPUTER BASED SYSTEMS (AGARD Conf.) Proceeding No. 149, pp 6-1 to 6-11, 1974.

ITOH D. and IZUTANI T., "FADEBUG -1. A New tool for program debugging", IEEE SYMP. ON COMPUTER SOFTWARE RELIABILITY, pp 38-43, 1973.

JONES A.K. and WOLF W.A., "Towards the design of secure systems", SOFTWARE - PRACTICE AND EXPERIENCE, Vol. 5, pp 321-336, 1975.

KERNIGHAN B.W. and PLAUGER P.J., "Programming style for programmers and language designers", IEEE Symp. on computer software reliability, pp 148-154, 1973.

KERNIGHAN B.W. and PLANGER P.J., "Software Tools", Addison Wesley, 1977.

KNUTH D.E., "The art of computer programming", Vol. 1, Addison Wesley, 1968.

KNUTH D.E., "Structured programming with GOTO statements", COMPUTING SURVEYS, Vol. 6, pp 261-301, 1974.

KNUTH D.E., "An empirical study of FORTRAN programs", SOFTWARE-PRACTICE AND EXPERIENCE, Vol. 1, pp 105-133, 1971.

KOPETZ H., "Error detection in real-time software", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Boston, pp 81-88, August 1975(a).

KOPETZ H., "Systematic error treatment in real-time software", IFAC WORLD CONGRESS, Paper 34.1, Boston, August 1975(b).

KOPETZ H. and CRICHTON E.R., "Ergonomics and security of real-time programming", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Roquencourt, pp 315-330, June 1976.

KYLSTRA F.J., "Real-time programming", 5TH IFAC/IFIP Digital computer applications to process control, The Hague, Netherlands, June 1977.

LARSON S.L.G., "Adapting BASIC for process control - meeting the needs of a conversational control system", ADVANCES IN INSTRUMENTATION, Vol. 29, pp 512/1-7, October 1974.

LAURANCE N., "Structured programming in a real-time environment", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Boston, pp 49-58, August 1975.

LEPATOUREL D.A., JOHNSON R.R., MARQUARDT D. and GURD D., "Application notes - A data aquisition system based on CAMAC and supported by BASIC and FORTRAN", CAMAC BULL. No. 8, pp 7-9, 1973.

LIENTZ B.P., "A comparative evaluation of versions of BASIC", COMM. OF THE ACM, Vol. 19, No. 4, pp 175-181, 1976.

LINN E.Y., SCHOEFFLER J.D. and ROSE C.W., "Distributed microcomputer data aquisition", INSTRUMENTATION TECHNOLOGY, Jan. 1975, pp 55-61.

LUCAS H.C. JR. and KAPLAN R.B., "A structured programming experiment" COMPUTER JOURNAL. Vol. 19, No. 2, pp 136-138, May 1976.

MADNICK S.E. and DONOVAN J.J., "Operating systems", McGraw Hill, 1974.

MAPLES M.D. and FISHER E.R., "High-level language applications at Lawrence Livermore Laboratory, Univ. of California, USA". MICROPROCESSORS, Vol. 1, No. 5, pp 312-316, June 1977.

MARK A., "Experience on the design and implementation of a structured real-time operating system", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Eindhoven, June 1977.

McMONIGALL J.P., "Automatic documentation of modular programs", NCC INTERFACE No. 16, pp 354-5, Dec. 1974.

MEISSNER L.P., "Proposed control structures for extended FORTRAN", SIGPLAN NOTICES, Jan. 1976. Vol. 11, No. 1, pp 16-21.

MUSSTOPF G., "Special application programming - A status report", 1ST IFAC/IFIP SYMP. ON SOFTWARE FOR COMPUTER CONTROL, Tallin, USSR, May 1976.

NAUR P. and RANDELL B., Software engineering: report on a conference sponsored by the NATO Science Committee, Garmisch, Germany, October 1968.

NEELY P.M., "A new programming discipline", SOFTWARE-PRACTICE AND EXPERIENCE, Vol. 6, pp 7-27, 1976.

NG T.S. and CANTONI A., "Run-time interaction with FORTRAN using mixed code", COMPUTER JOURNAL, Vol. 19, pp 91-92, 1976.

NELSON S.S. and NELSON B.E., "Factors considered in selecting BASIC-PLUS as an implementation language", SYMPOSIUM ON TRENDS AND APPLICATIONS 1976: Micro and mini systems, Gaithersburg, pp 89-92, May 1976.

NEWMAN N. and LANG T., "Documentation for computer users", SOFTWARE-PRACTICE AND EXPERIENCE, Vol. 6, pp 321-326, 1976.

NEWTON ...../9.9

NEWTON R.S., "An exercise in multi-processor operating-system design", REAL-TIME COMPUTER BASED SYSTEMS, AGARD CONF. PROC. No. 149, May 1974

OGDIN J.L., "The case against BASIC", DATAMATION, Sept. 1st, 1971. pp 34-41.

OSTERWEIL L.J. and FOSDICK L.D., "DAVE - a validation error detection and documentation system for FORTRAN programs", SOFTWARE-PRACTICE AND EXPERIENCE, Vol. 6, pp 473-486, 1976.

OTTO R.E. and BUTLER J.L., "Structured programming for process control", ADVANCES IN INSTRUMENTATION, Vol. 29, pp 513/1-7, October 1974.

PAGAN F.G., "On interpreter-oriented definitions of programming languages", COMPUTER JOURNAL, Vol. 19, No. 2, pp 151-155, May 1976.

PALME JACOB, "Interactive software for humans", MANAGEMENT DATAMATICS, Vol. 5, No. 4, pp 139-154, 1976.

PERSEUS COMPUTING AND AUTOMATION, MULTEX-BASIC manual WU-000168-01, July 1976.

PIERCE R.H., "Source language debugging on a small computer", COMPUTER JOURNAL, Vol. 17, No. 4, 1974.

POPEK G.J., HORNING J.J., LAMPSON B.W., MITCHELL J.G., LONDON R.L., "Notes on the design of EUCLID", ACM SIGPLAN NOTICES, Vol. 12, No. 3, pp 11-18, March 1977.

PURDUE, "Real-time extensions and implementations of real-time basic", International workshop on industrial computer systems, October 1975.

RADUE J.E. and MULLINS J.M., "Solving synchronization problems using semaphores", SOFTWARE-PRACTICE AND EXPERIENCE, Vol. 5, pp 51-64, 1975.

RAIMONDI D.L., GLADNEY H.M., HOCHWELLER G., MARTIN R.W. and SPENCER L.L., "Labs/7 - a distributed real-time operating system", IBM SYSTEMS JOURNAL, pp 81-101, No. 1, 1976.

RAMAMOORTHY C.V. and SUI-BUN F.HO, "Testing large software with automated software evaluation systems", IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, SE-1, pp 46-58, March 1975.

REIGEL E.W., FABER U. and FISHER D.A., "The interpreter - a microprogrammable building block system", IFIPS CONFERENCE PROCEEDINGS, Vol. 40, pp 705-723, 1972.

REITER A., "A resource-allocation scheme for multi-user on-line operation of a small computer", PROCEEDINGS AFIPS SJCC, Vol. 30, pp 1-8, 1967.

ROBSON J.M., "Worst case fragmentation of first fit and best fit storage allocation strategies", COMPUTER JOURNAL, Vol. 20, pp 242-244, 1977.

RYDER B.G., "The PFORT verifier", SOFTWARE-PRACTICE AND EXPERIENCE, Vol. 4, pp 359-377, 1974.

SCHICKER P., BAECHI W. and DUENKI A., "Job control in a heterogenous computer network", COMMUNICATIONS NETWORK, Online publishers, Uxbridge, Middlesex, England. Sept. 1975. pp 537-547.

SCHOEFFLER J.D., "Realtime operating systems for distributed process control systems", IFAC WORLD CONGRESS. Paper 27.3, Boston 1975.

SCHOEFFLER J.D. and KEYES M.A. IV, "Hierarchical language processing" 1ST IFAC/IFIP SYMP. ON SOFTWARE FOR COMPUTER CONTROL, pp 263-272, Tallin, USSR, May 1976.

SCHROTT G., "Common elementary synchronization functions for an operating system kernel", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Roquencourt, pp 239-256, June 1976.

SCHULTZ J.T., "Real-time multi-user BASIC", HEWLETT-PACKARD JOURNAL, Vol. 27, No. 5, 1976.

SLOMAN M.S., PENNY B.K. and MARSLAND T.A., "A communication protocol for distributed microprocessors", IEEE CONF. ON DISTRIBUTED COMPUTER CONTROL SYSTEMS, Birmingham, September 1977.

SNOWDON R.A., "PEARL - an interactive system for the preparation and validation of structural programs", Univ. of Newcastle-upon-Tyne, Computing Laboratory, Tech. Report Series 28.

SCOWEN R.S. and WICHMANN B.A., "The definition of comments in programming languages", SOFTWARE - PRACTICE AND EXPERIENCE, Vol. 4, pp 181-188, 1974.

SMEDEMA K. (Ed.), "Session 7 general discussion, IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Eindhoven, June 1977.

SPANG H.A. III, "The structure and comparison of three real-time operating systems for process control", AUTOMATICA, Vol. 8, pp 49-64, 1972.

SPANG H.A. III, "Measurement and improvement of memory allocation in a process computer", 4TH IFAC/IFIP SYMP. ON DIGITAL COMPUTER APPLICATIONS TO PROCESS CONTROL, pp 236-247, Zurich, March 1974.

STOY J.E. and STRACHEY C., "OS6 - an experimental operating system for a small computer. Part 1: General principles and structure", COMPUTER JOURNAL, Vol. 15, No. 2, pp 117-124, May 1972.

TANENBAUM A.S., "A survey of operating systems", INFORMASIE JAARGANG, Vol. 18, No. 12, Amsterdam, Dec. 1976.

TAYLOR J.R., "A comparison of two real-time programming languages", AERE, U.K., PAPERS ON REAL-TIME PROGRAMMING (3) 1971.

TEALE D.W., "Dual computers - an overlooked process control solution", ADVANCES IN INSTRUMENTATION, Vol. 30, 517:1-4, 1976.

TOBIAS R.L., "Common extensions to the BASIC programming language", IBM TECHNICAL DISCLOSURE BULLETIN, Vol. 17, No. 12, pp 3508-12, 1975.

VAN AARDT J.L., "Microprocessor emulation of a Varian minicomputer", South African Council for Scientific and Industrial Research, Sept. 1977. Report NIDR 77/03.

VAN MEURS J. and CARDOZO E.L., "Interfacing the user", SOFTWARE-PRACTICE AND EXPERIENCE, Vol. 7, pp 85-93, 1977.

VOJNOVIC D., "Kernel real-time system", IFAC/IFIP WORKSHOP ON REAL-TIME PROGRAMMING, Eindhoven, June 1977.

WAGNER F. and WODA H., "Process BASIC, KFV-PDV", Karlsruhe, Report 53, 1975.

WETTSTEIN H., "The implementation of synchronizing operations in various environments", SOFTWARE-PRACTICE AND EXPERIENCE, Vol. 7, pp 115-126, 1977.

WILKES M.E., "Software engineering and structured programming", IEEE TRANS. ON SOFTWARE ENGINEERING, Vol. SE-2, No. 4, pp 274-276, 1976.

WILCOX T.R., DAVIS A.M. and TINDALL M.H., "The design and implementation of a table driven, interactive diagnostic programming system", COMM. OF THE ACM, Vol. 19, No. 11, 1976.

WILKINS A.G., "Swepspeed-1 user guide", Central Electricity Generating Board, London, SSD/SW/75/N16, January 1976.

WILLIAMS T.J., "Trends in the development of process control computer systems", Journal of quality technology, Vol. 8, No. 2, April 1976.

WIRTH N. "Program development by step-wise refinement", COMM. OF THE ACM, Vol. 14, pp 221-227, 1971.

WIRTH N., "Toward a discipline of real-time programming", COMM. OF THE ACM, Vol. 20, pp 577-583, 1977.

WULF W., COHEN E., CORWIN W., JONES A., LEVIN R., PIERSON C. and POLLACK F., "Hydra: the kernel of a multiprocessor operating system", COMM. OF THE ACM, Vol. 17, No. 6, pp 337-345, June 1974.

ZEH A., "Reliability of process control computer languages", 1ST IFAC/IFIP SYMPOSIUM ON SOFTWARE FOR COMPUTER CONTROL, SOCOCO-76 pp 245-251, May 1976.

ZIMA H., "A programming language for real-time systems", COMPUTING WITH REAL TIME SYSTEMS, Proc. of 1st European Seminar, Vol. 1, Transcripta, London 1972.

APPENDIX A

VIPER

A.1 BNF Description of VIPER.

A.2 VIPER commands.

A.3 Hardware and software development systems.

APPENDIX A1

BNF DESCRIPTION OF VIPER

```
segment ::= <procedure> | <subroutine>
procedure ::= PROC| PROCEDURE [ <name>]
              <statements>
              [ START
              <statements>]
               END| <goto>

subroutine ::= SUB| SUBROUTINE[ <name>[ (<formal param list>)]]
              <statements>
              [ START
               <statements>]
               RETURN| <goto>
formal param list ::= <variable>[,<formal param list>]
command ::= <proc stm>
statements ::= <line no>{<proc stm>| <control stm>}
               [ statements]
proc stm ::=  <assign>| <print>| <unary if>| <rem>| <goto>|  <input >|
             <common>| <dim>| <op stm>| <call>
control stm ::= {<if>| <for>| <while>| <case>| <error>| < region>}
assign ::= LET <assignment list>
assignment list ::= <assignment>{;<assignment list>}
assignment ::=  <assignment head list>= <expr>
assignment head list ::={<variable>| <system assign>}[ =<assignment head list>]
system assign ::= {PRIORITY| PASSWORD| ACCESS}[ ( name) ]

call ::= CALL <sub name>[ (<expr list>)]
expr list ::= <expr>[ ,<expr list>]
print ::= PRINT[ <lu spec>][ print list]
print list ::={<expr >| "<string>"| TAB(<expr>) }[ { ,| ; }<print list>]
lu spec ::= (<expr>)
input ::= INPUT [ <lu spec>] <variable list>
variable list ::= <variable>[ ,<variable list>]

rem ::= REM [ <string>]

goto ::= GOTO<line no>
```

common ::= COMMON<com name>[<variable list>]

dim ::= DIM<dim list>

dim list ::= <array variable>[,<dim list>]

op stm ::= <lock free> | <list>| <save >| <get>
<run>| <wait>| <log on>| <log off>| <name ops>

name ops ::= {RESET| STATUS| MONITOR| DEBUG| CHANGE|
TRACE ON| TRACE OFF| SCRATCH| DELETE|
GO}[<proc name>]

lock free ::= LOCK| FREE<com name>

list :: = LIST[<lu spec>][<proc name>][,<line no>[,<line no>]]

save ::= SAVE[lu spec][<proc name>| <com name>]

get ::= GET[<proc name>| <com name>]|[,<io address>]

RUN ::= RUN[<proc name>][<time spec>]

time spec ::= {{EVERY| IN}<expr>{SECS| MINS| HOURS}}|
{AT<expr>:<expr>[:<expr>] }| <time spec>

wait ::= WAIT<expr>{SECS| MINS| HOURS}

log on ::= LOGON<pass word>[,<lun>[,<priority>[,<access>]]]

lun :: = <number> (logical unit no)

priority :: = <number> (Maximum priority of password)

access :: = <octal constant> (Access states allowed to the password)

log off ::= LOGOFF[<password>[,<lun>]

proc name :: = <name>

com name :: = <name>

line no :: = <integer>

io address :: = <integer>

number :: = <integer>| <octal constant>

octal constant :: = <integer>B

unary ...../A1.3

```
unary if ::= IF<expr><proc stm>

if ::= IF< expr>
       THEN[ <proc stm>]
       [ statements]
       [ ELSE[ <proc stm>]
       [ statements] ]
       ENDIF

while ::= DOWHILE<expr>
          <statements>
          ENDDO

for ::= FOR<variable>=<expr> TO <expr>[ STEP<expr>]
        <statements>
        NEXT<variable>

case ::= <case list>
         ENDCASE<variable>
case list ::= CASE<variable><rel op><expr>
              <statements>
              <case list>

error ::= ERROR
          <statements>
          ERET

region ::= REGION<name>
           <statements>
           END REGION<name>

expr ::= <conj>| <conj> OR <expr>
conj ::= <boolian op>| boolian op> AND <conj>
boolian op ::= <arith expr>| <arith expr><rel op><boolian op>
rel op ::= >| <| >=| <=| #| =
arith expr ::= <term>| <term><pm op><arith expr>
pm op ::= +| —
term ::= <factor>| <factor><md op><term>
md op ::= *| /
factor ::= <primary>| <un op><primary>
```

un op ...../A1.4

```
un op ::= +| -| NOT
primary ::=  <operand>| <operand>**<primary>
operand::= <variable>| <decimal no>| <system function>| ( <expr>)
system function ::= <trig func>| <arith func>| <format func>
                  | <access function>| <bit function>
trig func ::= {SIN| COS| TAN| ATN}(<expr>)
arith func ::= {EXP| LOG| SQR| RND} ( <expr>)
format func ::= {FLT| FIX| INT| SGN}( <expr>)
bit func :: = {SHIFT| XOR| BIT}(<expr>,<expr>)
access func :: = {PRIORITY| PASSWORD| ACCESS[(<name>)
                | READA| WRITEA| READA+WRITEA
variable ::= <dim variable>| <simple variable>
dim variable :: = <name>(<expr>[ ,<expr>] )
simple variable ::= <name>
name ::= <letter><letter digit>                          (max length = 16)
letter digit :: = <letter>| <digit><letter digit>
letter :: =  A| B| C ... | Z|
digit  :: =  0| 1| 2 ... | 9
```

APPENDIX A.2

VIPER COMMANDS

This appendix describes the commands which are available in VIPER. All the commands can also be used as program statements, although some, such as LOGON, CHANGE, DEBUG, etc. are seldom used in this mode. The syntax of the statements is the same in both cases, only the presence or absence of a statement number differentiating between the two modes.

The BNF description of the command syntax was given in Appendix A.1. In this appendix the syntax is repeated for ease of reference, followed by a semantic description and examples in some cases.

LOGON<password>[,<lun>[,<priority>[,<access>]

- <password> - new password can only be specified if command is issued by Master password holder; if password is known, identifies user to system.
- <lun> - accept further input from device specified by logical unit number (lun). Current terminal remains active until LOGOFF. If not specified remain on current terminal.
- <priority> - can only be specified by Master; determines maximum priority which can be specified by this password holder.
- <access> - can only be specified by Master; determines states in which user can operate (a user can be excluded from CHANGE or DEBUG)

Examples:

- LOGON MASTER - Logon with master password (any name, up to 16 characters, specified at system generation).
- LOGON USER1, 2, 50, 77B - Establish USER1 on logical unit 2, maximum priority of 50, all states permissible.
- LOGON USER2, 3, 90, 17B - USER3 not permitted to enter DEBUG or CHANGE modes
- LOGON USER4 - Change to previously specified User4 password on the same terminal.

LOGOFF[<password>[,<lun>]]

Terminate input from a terminal. No further input accepted until correct LOGON entered. Password and logical unit number can only be specified by Master; used to logoff a particular user from the system: <lun> = 0 deletes the specified password, user cannot LOGON again.

Examples:

LOGOFF	-	Terminate current session; disables terminal until correct LOGON entered
LOGOFF USER1, 2	-	Terminate USER1 or unit 2 (Master only)
LOGOFF USER2, 0	-	Delete password USER2 (Master only)

PROCEDURE <name>

Create a new procedure with specified name. If issued as a command, name must be specified and must be unique.

SUBROUTINE <name>[(<formal param list>)]

As procedure, except parameter list can be specified when used as a program statement. Parameter list ignored when issued as a command. (The difference between procedures and subroutines is arbitary and was adopted largely for ease of transition of FORTRAN oriented programmers. A single type, procedure, would be sufficient.)

CHANGE [<proc name>]

Move to CHANGE mode, if permitted by password attributes, on the specified procedure (or subroutine). If name not specified, shift mode on current segment. Permits any changes to be made to procedure.

DEBUG [<proc name>]

As CHANGE, but in DEBUG mode existing statements cannot be changed or deleted and only PRINT and LET statements can be added. Statements added under DEBUG can be deleted, however.

MONITOR [<proc name>]

Permit state of procedure to be monitored, but allow no changes or additions.

LIST ...../A2.3

LIST [<proc name>][,<line no>][,<line no>]

List a procedure or any portion of it. Current procedure assumed if name ommitted.

Examples:

LIST - List all of current procedure
LIST PROCA - List all of procedure PROCA
LIST, 100, 200 - List from statement 100 to 200 of current
LIST PROCB, 300 - List statement 300 only of PROCB

RUN [<proc name>][<time spec>]

Examples:

RUN - Execute current
RUN PROCA - Execute PROCA
RUN PROCB EVERY 10 SECS - Cyclic execution
RUN EVERY 10 SECS IN 2 MINS - Cyclic after delay
RUN PROCD AT 10:20 - At time of day
RUN EVERY 1 HOURS AT CURRENT.HOUR+1:0:0
- Every hour on the hour
RUN WEEKLY EVERY 24*7 HOURS
- Run once a week
RUN SHUTDOWN IN 2*24 HOURS AT 04:00:30
- Shutdown at 04h00.30 in 2 days time

WAIT <expr> SECS| MINS| HOURS

Wait designated period before resuming execution.

Examples:

WAIT 2 SECS
WAIT 2*X MINS

SAVE [(<lun>)][<name>]

Save a procedure or common data file on the external device specified by logical unit lun. (In VIPER, (lun) always defaulted to a single bulk storage device, compucorder or Disc). Name optional, current saved if not specified.

Examples:

SAVE - Save current on default bulk storage device
SAVE PROCA - Save specified procedure
SAVE COMX - Save current values in data area COMX

GET ...../A2.4

GET [(<lun>)][<name>][,<io address>]

Obtain a copy of a procedure from a specified (or default) bulk storage device. Restore named file (procedure or common) or obtain file from a particular physical address on the device. (Used for Compucorder where no off-line directory exists)

Examples:

GET	-	Restore current with text as at last SAVE
GET PROCA	-	Restore specified procedure
GET, 90	-	Obtain a procedure from address 90 of compucorder (legality of address is carefully checked with code words on the magnetic tape).

RESET [<proc name>]

Clear all entries on scheduler lists; release externals; delete any unused descriptors on symbol table. Name optional. Password holder only.

SCRATCH [<proc name>]

Clear symbol and statement pools but do not delete segment. (Releases all externals first)

DELETE [<proc name>]

Delete segment, does reset first then deletion. If current procedure deleted, move terminal control back up to father, or Master if no father exists and logoff if father or Master password does not match current.

STATUS [<name>]

Display the status of a procedure or common area. Procedure status indicates lists on which procedure resides, and scheduler parameters. Common status indicates state of sempaphore and size information.

TRACEON [<proc name>]
TRACEOFF [<proc name>]

Turn statement execution count trace on and off. Count is examined by using LIST with trace still on. TRACEON, TRACEOFF and RESET resets count to zero. Procedure name optional.

LOCK ...../A2.5

LOCK <com name>

Lock the semaphore associated with the specified common data area. Procedures executing further LOCK or REGION statements suspend pending a FREE or END REGION.

FREE <com name>

Unlock (release) semaphore. If any procedures are suspended waiting on this semaphore, the one which has been waiting longest will be released to execute.

STOP [<proc name>]

Suspend execution of procedure, saving suspension point and displaying message on console device:

STOPPED IN LINE XXX OF <proc name>

If name ommitted, stop procedure which is currently associated with input device. A "stopped" procedure can only be restarted with a GO or GOTO.

END [<proc name>]

Terminate execution immediately, does not save termination point. Also used as normal program termination statement.

TURNOFF [<proc name>]

Remove from time list, permitting procedure to complete current execution, i.e. inhibit repetitive execution.

GO [<proc name>]

Restart a procedure after a STOP. Continues executing from suspension point.

GOTO <line no>

Restart execution after a STOP at a specified line number.

ACCESS (<name>)=0| READA| WRITEA| READA+WRITEA

Set access attributes of a data element. This can be either a shared data segment name (common name); the name of either a simple or subscripted variable within a common area; a local array variable; or a formal parameter.

Examples/ ...../A2.6

Examples:

```
ACCESS(COM1)  =  READA+WRITEA          - Read and write access
ACCESS (ARRAY) = READA                 - Read only
ACCESS (SOMENAME)  =  0                - No access
```

PRIORITY [(<proc name>)] = <value>

Assign a priority to a procedure, in the range 0 to 127. 0 is highest priority, 127 lowest. The password attributes may prohibit setting a priority below a specified value (see LOGON).

Examples:

```
PRIORITY (PROC1) = 50
PRIORITY = 40 - Set priority of current procedure associated with
                input device.
```

PASSWORD [(<name1>)] = PASSWORD [(<name2>)]

Change the password associated with procedure or data area <name1> to that associated with <name 2>. Only the Master can use this command to change passwords.

Examples:

```
PASSWORD (PROC2) = PASSWORD - PROC2 password = current
PASSWORD (PROC2) = PASSWORD (PROC3)
```

The ACCESS, PRIORITY and PASSWORD functions can also be used in expressions to determine the value of the attribute.

Examples:

```
PRINT ACCESS (COM1)
IF ACCESS (ARRAY) = READA PRINT "ARRAY READ ONLY"
IF ACCESS (SOMENAME) = 0 CALL NO.ACCESS.FIX
LET PR1 = PRIORITY (PROC1)
IF PASSWORD (PROC2) = PASSWORD (PROC3) PRINT "SAME PASSWORDS"
```

APPENDIX A.3

HARDWARE AND SOFTWARE DEVELOPMENT SYSTEMS

VIPER is written in VARIAN Assembler code which is cross-assembled on a CDC CYBER 174. The cross assembler program is written in FORTRAN and was originally run on an IBM 370/158. Additions and changes to the VIPER source are performed with the CDC KRONOS text editor on a (remote) CRT terminal. As down loading facilities from the CDC directly into the Varian had not been implemented at the time that the development of VIPER was taking place, the binary output of the cross-assembler is dumped on paper tape for loading into the Varian. (As there is in fact no paper tape punch unit on the CYBER, output is via an intermediate 9 track magnetic tape, for punching on an off-line unit.)

All the development work on VIPER was performed on a Varian 620i com= puter with 16K words of core memory. This computer is equipped with a paper tape reader, magnetic tape cassette unit, cartridge disc and CRT and TTY terminals in addition to process input-output units and a CAMAC System Crate interface. In April 1978 the construction of a microprogrammable emulator of the Varian was completed and further development of VIPER and the programs of the case study was performed on this machine. The emulator, called the MIKROV, uses INTEL 3000 bit slices and was based on a design by J. VAN AARDT (1977) of NIDR, CSIR. This machine was operated with 24K of RAM initially which was later upgraded to 28K. The remaining 4K of the 32K memory space is allocated for PROM memory. Only 2K of this space has been used for a resident debug aid plus paper tape and cassette load/dump utilities.

On the 620i the cartridge disc unit was used as a swapping device while a CAMAC bulk memory unit (which was constructed specifically for VIPER use) was used on the MIKROV. Magnetic tape cassette units were used for program storage on both machines. The CAMAC bulk memory module was built using 16K dynamic RAM memory chips and was designed and layed-out for a capacity of 64K words in a single Camac module, but only 16K words were used for the case study as the module was operated with only one quarter of the chips inserted. No battery back up was provided for this module as a high-speed AC mains switch over unit was used at the Huletts Refinery which switched to an alternative AC supply if the primary supply failed.

A P P E N D I X   B

CASE STUDY

HULETTS REFINERY COMPUTER CONTROL PROJECT

PROGRAM INDEX

APPENDIX B.1

PROCESS DESCRIPTION

The process under control was the front end of the Huletts Sugar Refinery at Rossborough, Durban. Fig. B.1 is a schematic diagram of this section of the refinery. The control functions consisted of three flow control loops and four quality control loops to control pH and reagent addition. A number of monitoring functions were also performed.

The software is organised in two classes, the first being the timing and scanning programs and the second the control and monitoring programs, as shown in Fig. B.2. The synchronization of the programs is performed with semaphores and communication amongst the programs is performed via a single global COMMON area.

The computer was interfaced to the process using a CAMAC system, as shown in Fig. B.3. This diagram shows a dual computer configuration. This use of dual computers was investigated briefly but due to the rapid and continual development of programs that took place during the period when this thesis was in progress, the dual computer configuation was never used for control work. All the results reported in this thesis were obtained on the single HP 21MX running under control of the HP RTE (Real-time Executive). RTE 2 was used initially with 32K of memory but this was later upgraded to RTE 3 with 48K of memory.

The programs depicted in Fig. B.2 and listed in Appendix B.3 were all independent modules which could be separately compiled and executed. This facilitated the testing and on-line expansion of the system as new functions were added.

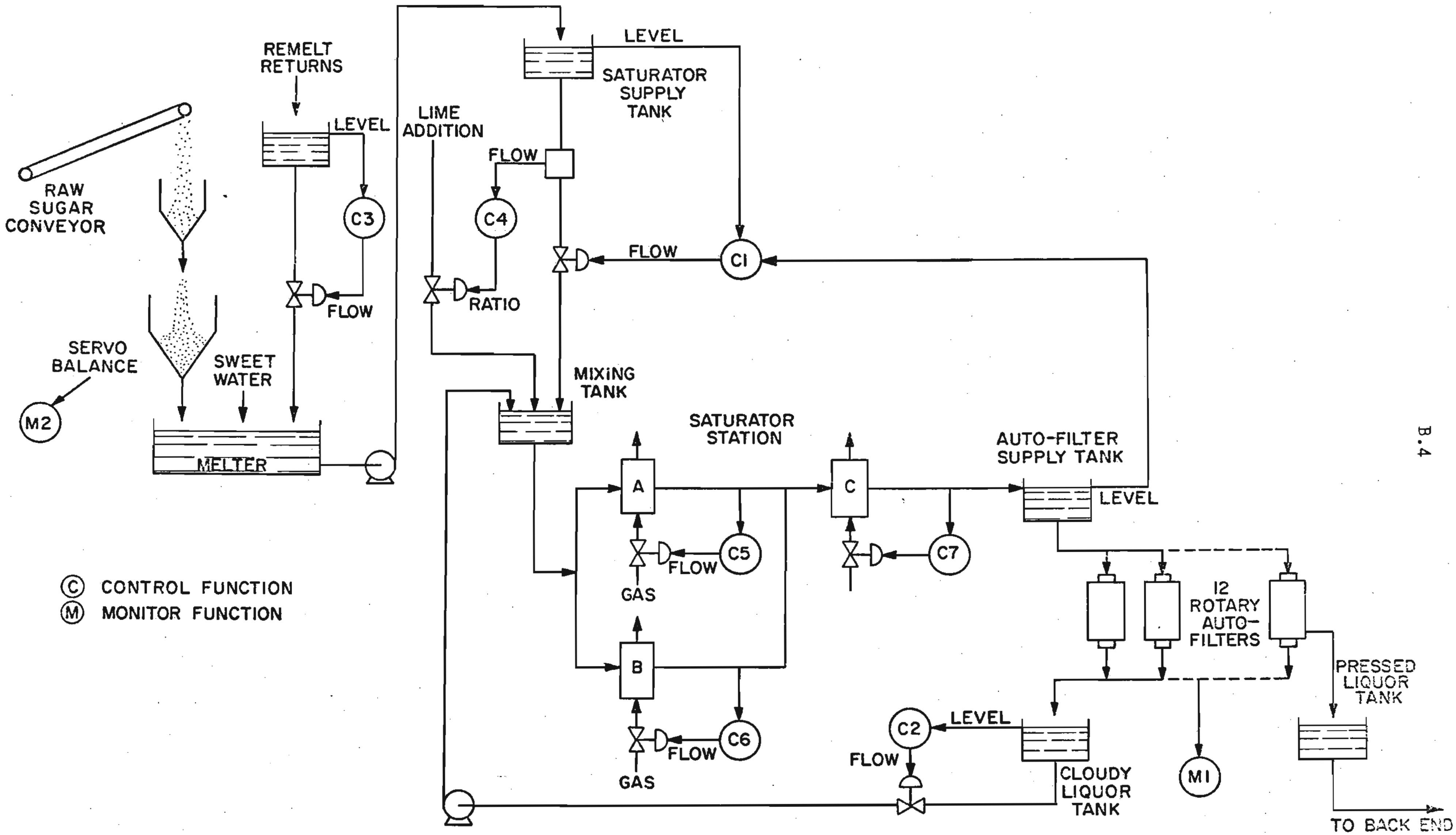
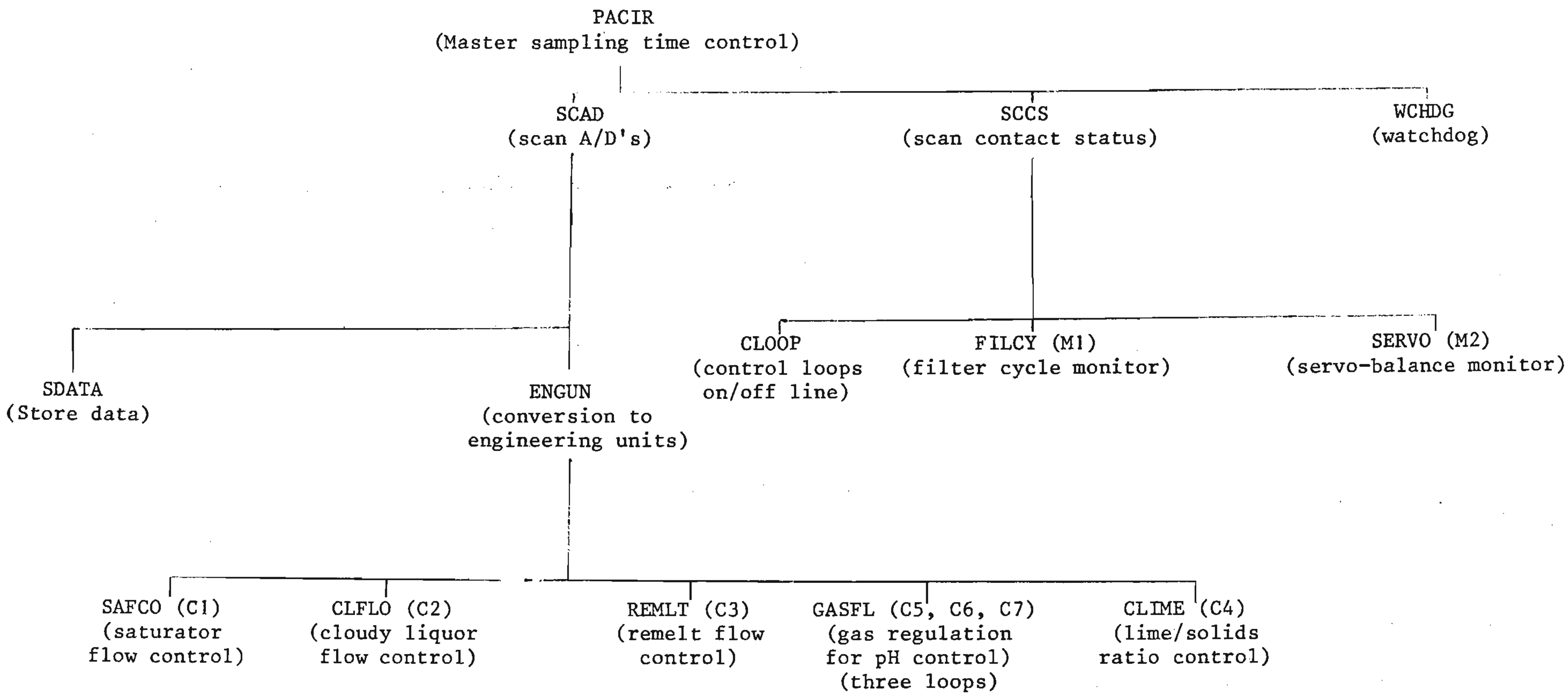


FIGURE B.1

THE HULETTS SUGAR REFINERY COMPUTER CONTROL PROJECT



The numbers C1 - C7 and M1, M2, correspond to the elements marked in Figure B.1.

FIGURE B.2 SCHEMATIC OF FACTORY SOFTWARE

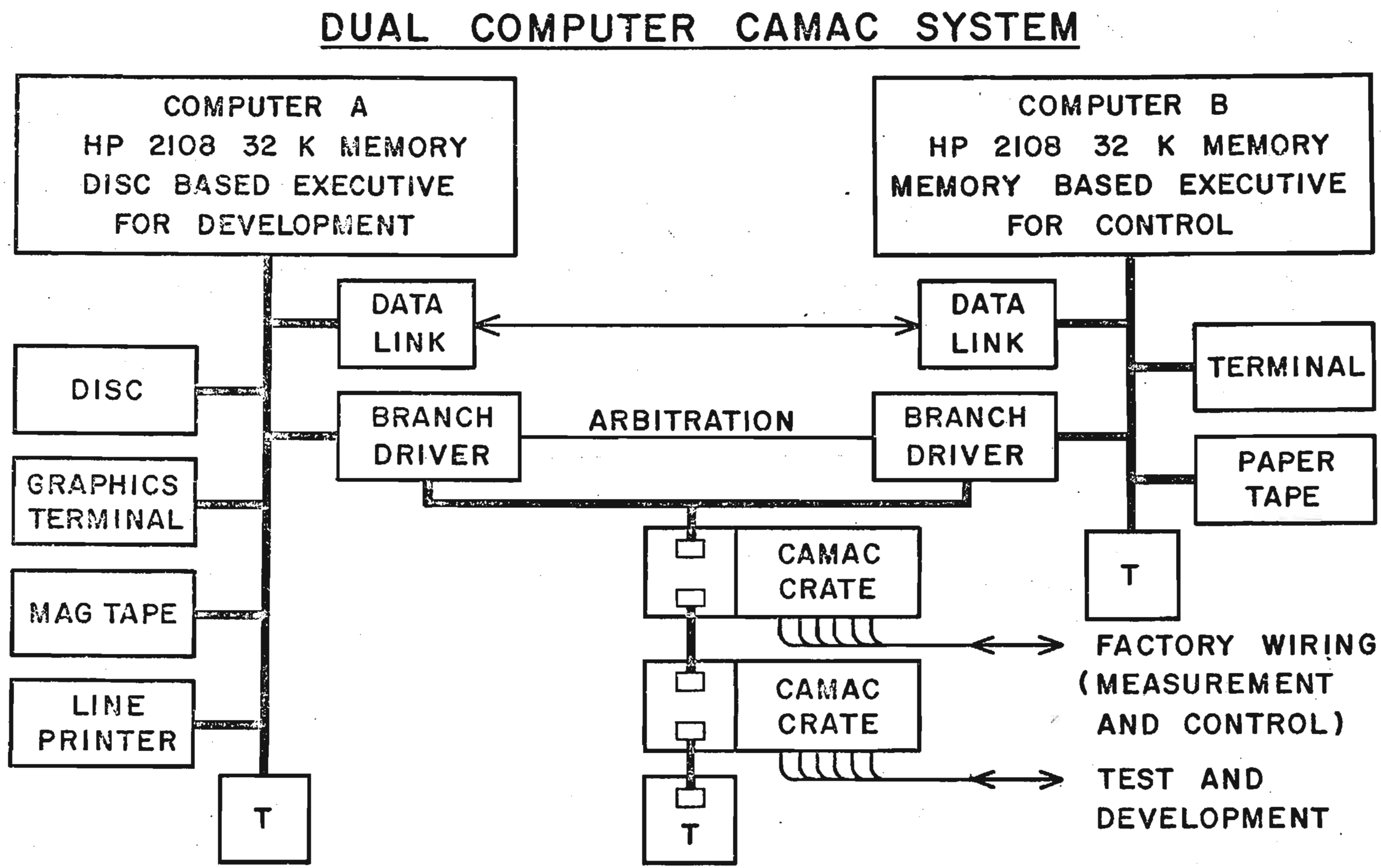

DUAL COMPUTER CAMAC SYSTEM
COMPUTER A
HP 2108 32 K MEMORY
DISC BASED EXECUTIVE
FOR DEVELOPMENT
COMPUTER B
HP 2108 32 K MEMORY
MEMORY BASED EXECUTIVE
FOR CONTROL
DATA LINK
DATA LINK
DISC
TERMINAL
BRANCH DRIVER
ARBITRATION
BRANCH DRIVER
GRAPHICS TERMINAL
PAPER TAPE
CAMAC CRATE
MAG TAPE
T
FACTORY WIRING
(MEASUREMENT
AND CONTROL)
CAMAC CRATE
LINE PRINTER
TEST AND
DEVELOPMENT
T
T

FIGURE B.3

APPENDIX B.2

VIPER PROGRAMS

Notes:

1. The VIPER programs have relatively few comments. This was because of the small memory size of the Varian 620i on which VIPER was running at the time the programs were written. (The MIKROV with its larger memory, was only used later.) For expanded descriptions of any of the programs see the FORTRAN listings. See also section 4.5.1.2.

2. The numbers on the right hand side of the listings of some of the programs are statement execution counts - as described in section 4.3.2.1 p. 4.22. (A bug in VIPER resulted in some of the counts being incorrect. This has been fixed, but the listings were not updated.)

```
VIPER  REV  A7  12/04/78  08:10:29.7  18/04/78

#    1 PROCEDURE SCANCS                                 SCAN CONTACT INPUT SWITCHES
     2  REM  9-11-77
    50  COMMON BITS,CIN(4),SCOP(2)
    60  LET ACCESS((CIN)=READA+WRITEA
    80  LET SRVO1=SRVO2=CNTO=FILO1=FILO2=FILO3=D=0
   100  CALL DECLR(CSW1,1,12,0)
   110  CALL DECLR(CSW2,1,12,1)
   120  CALL DECLR(CSW3,1,13,0)
   130  CALL DECLR(CSW4,1,13,1)
   140  CALL CAMAC(18,CSW1,D,Q)                       WRITE LAM MASK TO ALL CHANNELS
   150  CALL CAMAC(18,CSW2,D,Q)
   160  CALL CAMAC(18,CSW3,D,Q)
   170  CALL CAMAC(18,CSW4,D,Q)
   300 START SCANCS
   310 CALL CAMAC(0,CSW1,FILN1,Q)                     READ CONTACT STATUS SWITCHES
   320 CALL CAMAC(0,CSW2,FILN2,Q)
   330 CALL CAMAC(0,CSW3,CIN3,Q)
   340 CALL CAMAC(0,CSW4,CIN4,Q)
   350 LET CIN(1)=FILN1 ; CIN(2)=FILN2 ; CIN(3)=CIN3 ; CIN(4)=CIN4
   400 LET FILN3=CIN3 AND 15 ; CNTN=SHIFT(CIN4,-1)
   420 LET SRVN1=BIT(15,CIN3) ; SRVN2=BIT(1,CIN4)
   500 LET FILD=XOR(FILO1,FILN1)+XOR(FILO2,FILN2)+XOR(FILO3,FILN3)
   510 LET SRVD=XOR(SRVO1,SRVN1)+XOR(SRVO2,SRVN2)
   520 LET CNTD=XOR(CNTO,CNTN)
   600 IF FILD RUN FILTER.MONITOR
   610 IF SRVD RUN SERVOTIP
   620 IF CNTD RUN CLOOP
   700 LET FILO1=FILN1 ; FILO2=FILN2 ; FILO3=FILN3
   710 LET SRVO1=SRVN1 ; SRVO2=SRVN2 ; CNTO=CNTN
   720 END SCANCS
```

```
VIPER  REV  A7  12/04/78  10:42:46.0  18/04/78

*     1 PROCEDURE SCANADC                                         SCAN A TO D CHANNELS
    2    REM  231277BDR
   50    COMMON SPECS,NADC,CDUM(2)
   60    COMMON VOLTS,V(NADC)
   70    LET ACCESS((VOLTS)=READA+WRITEA
   80    CALL DECLR(MUX1A,1,6,0)
   90    CALL DECLR(MUX1B,1,6,1)
  100    CALL DECLR(MUX2A,1,7,0)
  110    CALL DECLR(MUX2B,1,7,1)
  120    CALL DECLR(ADC,1,8,0)
  300 START SCANADC
  305 LET J=0 ; JINC=17 ; MUX1=MUX2=MUX1A
  310 FOR I=1 TO NADC
  320    CALL CAMAC(26,MUX1,D,0)                                            RESET
  330    CALL CAMAC(16,MUX2,J,0)                               WRITE OUT CHANNEL NO
  340    CALL CAMAC(2,ADC,D,0)                                     START CONVERSION
  350    LET J=J+JINC            COMPUTE NEXT CHAN WHILE WAITING FOR COMPLETION
  360    IF I=17 LET J=0 ; JINC=4352 ; MUX1=MUX1B
  370    IF I=33 LET J=0 ; JINC=17 ; MUX1=MUX2=MUX2B
  390    IF I=49 LET J=0 ; JINC=4352 ; MUX2=MUX2A
  530    LET Q=0
  540    DOWHILE Q=0                              WAIT FOR CONVERSION TO COMPLETE
  550      CALL CAMAC(0,ADC,D,Q)
  560    ENDDO
  570    LET V(I)=(D-32)/(3273.5)
  580 NEXT I  310
  600 RUN ENGUNITS
  610 END SCANADC
```

```
VIPER  REV  A7  12/04/78  12:42:59.6  18/04/78

#     1 PROCEDURE ENGUNITS
      2   REM  010278BDR
     50   COMMON SPECS,NADC,ES,DELT
     60   COMMON VOLTS,V(NADC)
     70   COMMON ENG,E(ES)
     80   COMMON ENGLIM,EL(ES,2)
     90   LET ACCESS((V)=ACCESS((E)=READA+WRITEA
    100   CALL ENGLIMITS
    300 START ENGUNITS
    310 LET LLIM=0
    320 FOR I=1 TO NADC
    330   IF I>19 LET LLIM=2
    340   IF V(I)<LLIM
    350     THEN  CALL MESSAGE(1,I)
    360       LET V(I)=LLIM
    370   ENDIF
    380   IF V(I)>10
    390     THEN  CALL MESSAGE(1,I)
    400       LET V(I)=10
    410   ENDIF
    420 NEXT I  320
    500 FOR I=11 TO 18
    510   LET E(I)=V(I)*10
    520 NEXT I  500
    530 LET E15=E(15)
    540 IF E15<60 OR E15>90 LET E(15)=80
    600 LET SGPB=1.23+.13*V(3)
    610 IF SGPB<1.2449 OR SGPB>1.3889
    620   THEN  CALL MESSAGE(2,3)
    630     LET SGPB=1.29999
    640 ENDIF
    650 LET E(1)=100*(.3744*V(1)/SGPB)/2.26099
    700 LET E(2)=100*(.5573*V(2)/SGPB)/3.35299
    710 LET E(5)=100*(.4645*V(5)/SGPB)/3.871
    720 LET E(27)=100*(.4047*(V(27)-2)/SGPB)/2.165
    730 LET E(28)=100*(.2903*(V(28)-2)/1.276)/1.82
    800 LET E(3)=100*(SGPB-1.1+2.20000E-03*E15)/(.2695*SGPB+2.29000E-03*E15)
    810 LET E(10)=63+7*(V(10)-1)/4
    820 LET E(7)=(V(7)-1)/4
    830 LET E(8)=(V(8)-1)/4
    900 LET E(6)=11.76*SQR(V(6)/10)
    910 LET E(9)=(V(9)-.836)/.937
    920 LET E(23)=153.5*(V(23)-2)/8
    930 LET E(24)=5436.6*SQR((V(24)-2)/8)
    940 LET E(25)=5436.6*SQR((V(25)-2)/8)
    950 LET E(26)=2718.3*SQR((V(26)-2)/8)
   1000 LET E(20)=7+.625*(V(20)-2)
   1010 LET E(21)=7+.625*(V(21)-2)
   1020 LET E(22)=7+.625*(V(22)-2)
   1030 LET E(30)=(V(30)-2)*2.5
   1100 FOR I=1 TO ES
   1110   IF E(I)<EL(I,1) OR E(I)>EL(I,2)  CALL MESSAGE(2,I)
   1120 NEXT I 1100
   1200 RUN SATFLOW
   1210 RUN CLFLOW
   1220 RUN REMELT
   1230 RUN LIMERATIO
   1240 RUN GASFLOWA
   1250 RUN GASFLOWB
   1260 RUN GASFLOWC
   1270 END ENGUNITS
```

```
VIPER  REV  A7  12/04/78  14:32:33.0  18/04/78

#     1 PROCEDURE WATCH.DOG
     2   REM  170178BDR
    40   LET NLOOPS=7 ; MAXNO=2 ; LU=1
    50   COMMON BITS,CIN(4),SCOP(2)
    60   LET ACCESS((SCOP)=READA+WRITEA
    70   CALL DECLR(LAMG,1,23,0)
    80   LET SCOP(1)=0 ; SCOP(2)=0
    90   DIM FLAG((NLOOPS)
   100   FOR I=1 TO NLOOPS
   110     LET FLAG(I)=MAXNO
   120   NEXT I  100
   130   WAIT 1 MINS
   300 START WATCH.DOG
   310 CALL CAMAC(16,LAMG,0,Q)
   320 CALL CAMAC(0,LAMG,D,Q)
   330 IF D#0 PRINT (LU)"LAMG ERROR,0,D="D
   340 CALL CAMAC(16,LAMG,32767,Q)
   350 CALL CAMAC(0,LAMG,D,Q)
   360 IF D#32767 PRINT (LU)"LAMG ERROR,32767,D="D
   500 FOR J=1 TO NLOOPS
   510   LET STATJ=BIT(J,SCOP(1))
   520   IF FLAG(J)>100 LET FLAG(J)=100
   530   LET CNT=FLAG(J)-MAXNO
   600   IF CNT=0 OR STATJ=1
   605     THEN  LET CHAN=J-1
   610       CALL WCOUT(CHAN,STATJ)
   620       LET MASK=SHIFT(1,J)
   630       IF STATJ
   640         THEN  LET SCOP(2)=SCOP(2) OR MASK
   650           CALL MESSAGE(3,J)
   660         ELSE  LET SCOP(2)=SCOP(2) AND  NOT MASK
   670       ENDIF
   700       LET FLAG(J)=(FLAG(J)+1)*(1-STATJ)
   705   ENDIF
   710 NEXT J  500
   720 LET SCOP(1)=0
   730 END WATCH.DOG
```

```
VIPER  REV  A7  3/04/78   02:31:25.0  01/01/00

#     1 PROCEDURE SERVOTIP
      2   REM  310178BDR
     50   COMMON BITS,CIN(4),SCOP(2)
     60   COMMON SERVOD,PROD,PROD1,PROD2,MASSRATE,MASS.HOUR,DUM(11)
     70   DIM TMASS(2,90)
     80   LET ACCESS(SERVOD)=READA+WRITEA
    200   CALL DECLR(SCAL0,1,16,1)
    210   CALL DECLR(SCAL1,1,16,1)
    220   CALL CAMAC(9,SCAL0,D,Q)
    230   CALL CAMAC(9,SCAL1,D,Q)
    240   LET SRVO1=SRVO2=PROD=PROD1=PROD2=MASS.HOUR=0
    250   CALL TINT(DELS,TPREV)
    300 START SERVOTIP
    310 REGION SERVOD
    320   LET SRVN1=BIT(16,CIN(3)) ; SRVN2=BIT(1,CIN(4))
    330   IF SRVO1=0 AND SRVN1=1
    340     THEN  CALL CAMAC(2,SCAL0,MOT1,Q)
    350       LET TMASSO=MOT1/500 ; PROD1=PROD1+TMASSO
    360   ENDIF
    370   IF SRVO2=0 AND SRVN2=1
    380     THEN  CALL CAMAC(2,SCAL1,MOT2,Q)
    390       LET TMASSO=MOT2/500 ; PROD2=PROD2+TMASSO
    400   ENDIF
    410   LET PROD=PROD+TMASSO ; SRVO1=SRVN1 ; SRVO2=SRVN2
    420   LET MASS.HOUR=MASS.HOUR+TMASSO
    430   CALL TINT(DELS,TPREV)
    440   LET DELH=DELS/3600
    450   FOR I=90 TO 2 STEP -1
    460     LET J=I-1 ; TMASS(1,I)=TMASS(1,J) ; TMASS(2,I)=TMASS(2,J)+DELH
    470   NEXT I  450
    480   LET TMASS(1,1)=TMASSO ; TMASS(2,1)=0
    490   LET K=1 ; MASSRATE=0
    500   DOWHILE TMASS(2,K)<=1 AND K<=90
    510     LET MASSRATE=MASSRATE+TMASS(1,K) ; K=K+1
    520   ENDDO
    530   LET MASSRATE=MASSRATE/TMASS(2,K-1)
    540 ENDREGION SERVOD
    550 END SERVOTIP
```

```
VIPER  REV  A7  12/04/78  14:06:30.2  19/04/78

#     1 PROCEDURE SATFLOW                                                          1
    2   REM  230178BDR                                                             1
   50   COMMON SPECS,NADC,ES,DELT                                                  1
   60   COMMON ENG,E(ES)                                                           1
   70   COMMON VOLTS,V(NADC)                                                       1
   80   COMMON BITS,CIN(4),SCOP(2)                                                 1
   90   LET ACCESS((SCOP)=READA+WRITEA                                             1
  100   LET FMAF=153.5 ; HAFM=3.35299 ; HSSM=2.26099 ; HPLM=3.871                  1
  110   LET AAFST=6.59 ; ASST=11.3999                                              1
  120   LET VAFST=AAFST*HAFM ; VSST=ASST*HSSM                                      1
  130   LET HAFSP=.5 ; HSSSP=.3 ; VPLR=5                                           1
  140   LET GPA=.2 ; GIA=36 ; GPS=2 ; GIS=50                                       1
  150   LET W=1.50000E-03 ; DAMP=.7                                                1
  200   LET HSSN1=E(1)/100 ; HAFF1=HAFF2=RAF=E(2)/100                              1
  210   LET SOLIDS=DLSSV=0 ; ALPHA=.2                                              1
  220   LET NUMPT=1000*E(7) ; FLOW=E(23) ; BRIX=V(3) ; BRIX2=E(3)                  1
  230   CALL FILTERCOEF(W,DAMP,DELT,CB,CC,CD,CE)                                  12
  240   LET GIAV=1/(60*GIA) ; GISV=1/(60*GIS)                                      1
  300 START SATFLOW                                                              100
  310 LET HAFN=E(2)/100 ; HPLN=E(5)/100                                          100
  320 LET HSSN=E(1)*ALPHA/100+(1-ALPHA)*HSSN1                                    100
  330 IF HSSN<5.00000E-02 CALL MESSAGE(7,1)                                      100
  340 IF HSSN>.95 CALL MESSAGE(7,2)                                              100
  350 IF HAFN<5.00000E-02 CALL MESSAGE(7,3)                                      100
  360 IF HAFN>.95 CALL MESSAGE(7,4)                                              100
  400 LET HAFF=CB*HAFF1-CC*HAFF2+CD*HAFN+CE*RAF                                  100
  410 LET HFDOT=(HAFF-HAFF1)/DELT ; DELAF=HAFF-HAFF1                             100
  420 IF ABS(DELAF)>.1                                                           100
  430   THEN  CALL MESSAGE(7,9)                                                    0
  440     LET HFDOT=.1*SGN(DELAF/DELT)                                             0
  441 ENDIF                                                                      100
  445 LET HSDOT=(HSSN-HSSN1)/DELT ; DELSN=HSSN-HSSN1                             100
  450 IF ABS(DELSN)>.1                                                           100
  452   THEN  CALL MESSAGE(7,9)                                                    0
  454     LET HSDOT=.1*SGN(DELSN)/DELT                                             0
  456 ENDIF                                                                      100
  460 LET EAFT=HAFF-HAFSP ; ESST=HSSN-HSSSP                                      100
  470 LET HAFF2=HAFF1 ; HAFF1=HAFF ; RAF=HAFN ; HSSN1=HSSN                       100
  480 LET GPISST=GPS*(HSDOT+GISV*ESST)                                           100
  500 LET GAIN=0                                                                 100
  510 IF HPLN<.5 AND HAFN<.5 LET GAIN=1.00000E-03                                100
  520 LET GPIAST=-GPA*(HFDOT+GIAV*EAFT-GAIN*(.5-HPLN))                           100
  530 IF HSSN>HSSSP LET DLSF=GPIAST                                              100
  540 IF HSSN<HSSSP AND GPIAST>0 LET DLSF=GPISST                                 100
  550 IF HSSN<HSSSP AND GPIAST<0 LET DLSF=GPISST+GPIAST                          100
  560 LET DELN=DLSF*DELT ; DLSSV=DELN+DLSSV                                      100
  600 IF ABS(DLSSV)>1.00000E-03                                                  100
  610   THEN  LET NUMP=INT(DLSSV*1000)                                           100
  620     LET DLSSV=DLSSV-NUMP/1000                                              100
  630   ELSE  LET NUMP=0                                                           0
  640 ENDIF                                                                      100
  650 LET NPOS=E(7)*1000 ; DIF=NPOS-NUMPT                                        100
  660 IF ABS(DIF)>25                                                             100
  670   THEN  CALL MESSAGE(7,7)                                                   16
  680     LET NUMPT=NPOS                                                           4
  690 ENDIF                                                                      100
```

```
 700 IF ABS(NUMP)>100                                    100
 710   THEN  CALL MESSAGE(7,8)                              0
 720     LET NUMP=100                                       0
 730 ENDIF                                                100
 740 IF NUMP+NUMPT<0                                      100
 750   THEN  CALL MESSAGE(7,5)                              0
 760     LET NUMP=-NUMPT                                    0
 770 ENDIF                                                100
 780 IF NUMP+NUMPT>1000                                   100
 790   THEN  CALL MESSAGE(7,6)                              0
 800     LET NUMP=1000-NUMPT                                0
 810 ENDIF                                                100
 820 LET NUMPT=NUMPT+NUMP                                 100
1000 IF NUMP#0 CALL CAMAC(16,IPUL,NUMP,0)                 500
1100 LET FLOW=FLOW*(1-ALPHA)+ALPHA*E(23)                  100
1110 LET BRIX=BRIX*(1-ALPHA)+ALPHA*V(3)                   100
1120 LET SGPB=1.23+1.30000E-02*BRIX                       100
1125 LET BRIX2=BRIX2*(1-ALPHA)+ALPHA*E(3)                 100
1130 LET RATES=FLOW*SGPB*BRIX2/100                        100
1140 LET DSOLID=RATES*DELT/3600                           100
1150 LET SOLIDS=SOLIDS+DSOLID                             100
1160 LET SCOP(1)=SCOP(1) OR 1                             100
1170 END SATFLOW                                          -16
```

```
VIPER  REV  A7  12/04/78  17:05:57.6  19/04/78

#     1 PROCEDURE CLFLOW                                                  1
    2   REM  230178BDR                                                    1
   50   COMMON SPECS,NADC,ES,DELT                                         1
   60   COMMON ENG,E(ES)                                                  1
   70   COMMON BITS,CIN(4),SCOP(2)                                        1
   80   LET ACCESS((SCOP)=READA+WRITEA                                    1
  100   LET FMCL=10 ; HAFM=3.35299 ; HCLM=2.165                           1
  105   LET ACLT=4.67 ; VCLT=ACLT*HCLM                                    1
  110   LET VPLR=5 ; GPC=2 ; GIC=60 ; W=1.50000E-03 ; DAMP=.7             1
  120   LET HAFF1=HAFF2=RAF=E(2)/HAFM                                     1
  130   LET HCLF1=HCLF2=RCL=E(27)/HCLM                                    1
  140   LET DLLRV=0 ; GICV=1/(60*GIC)                                     1
  150   CALL FILTERCOEF(W,DAMP,DELT,CB,CC,CD,CE)                         12
  300 START CLFLOW                                                      200
  310 IF BIT(4,CIN(4))                                                  200
  320   THEN  LET HAFN=E(2)/100                                         200
  330     LET HAFF=CB*HAFF1-CC*HAFF2+CD*HAFN+CE*RAF                     200
  340     LET HFDOT=(HAFF-HAFF1)/DELT                                   200
  350     LET HAFF2=HAFF1 ; HAFF1=HAFF ; RAF=HAFN                       200
  360     LET HCLN=E(27)/100                                            200
  370     IF HCLN<5.00000E-02 CALL MESSAGE(8,1)                         200
  380     IF HCLN>.95 CALL MESSAGE(8,2)                                 200
  390     LET HCLF=CB*HCLF1-CC*HCLF2+CD*HCLN+CE*RCL                     200
  400     LET HCDOT=(HCLF-HCLF1)/DELT ; DELCF=HCLF-HCLF1                200
  410     IF ABS(DELCF)>.1                                              200
  420       THEN  CALL MESSAGE(8,6)                                       4
  430         LET HCDOT=.1*SGN(DELCF)/DELT                                1
  440     ENDIF                                                         200
  450     LET HCLF2=HCLF ; HCLF1=HCLF ; RCL=HCLN                        200
  500     LET HFDOT=0 ; HAFF=.3                                         200
  510     LET DLLR=GPC*((HCDOT-HFDOT)+GICV*(HCLF-HAFF))                 200
  520     LET DLLRV=DLLR*10 ; VPLR=VPLR+DLLRV*DELT                      200
  530     IF ABS(DLLRV)>1                                               200
  540       THEN  CALL MESSAGE(8,3)                                       0
  550         LET DLLRV=1                                                 0
  560     ENDIF                                                         200
  600     IF VPLR<0                                                     200
  610       THEN  CALL MESSAGE(8,4)                                       0
  620         LET VPLR=.1                                                 0
  630     ENDIF                                                         200
  640     IF VPLR>10                                                    200
  650       THEN                                                        195
  660         LET VPLR=9.89999                                          195
  670     ENDIF                                                         200
  680 ENDIF                                                             200
  700 CALL CDAC(0,VPLR)                                                2000
  710 LET SCOP(1)=SCOP(1) OR 2                                          200
  720 END CLFLOW                                                        -16
```

```
VIPER  REV  A7  12/04/78  15:26:24.8  19/04/78

*     1 PROCEDURE REMELT                                                    1
   2    REM  010278BDR                                                      1
  50    COMMON SPECS,NADC,ES,DELT                                           1
  60    COMMON ENG,E(ES)                                                    1
  70    COMMON BITS,CIN(4),SCOP(2)                                          1
  80    LET ACCESS((SCOP)=READA+WRITEA                                      1
  90    CALL DECLR(PULS,1,14,1)                                             1
 100    LET HRMM=1.82 ; AREA=10.03 ; HRMNSP=.25 ; DELN=0                    1
 110    LET ALPHA=.2 ; GPR=1 ; GIR=50 ; GIRV=1/(60*GIR)                     1
 120    LET HRMNS=E(28)/100 ; NUMPT=1000*E(8)                               1
 300 START REMELT                                                         300
 310 IF BIT(5,CIN(4))                                                     300
 320    THEN  LET HRMN=E(28)/100                                          300
 330      IF HRMN>.95 CALL MESSAGE(9,1)                                   300
 340      IF HRMN<5.00000E-02 CALL MESSAGE(9,2)                           300
 350      LET HRNDOT=ALPHA*(HRMN-HRMNS)                                   300
 360      LET HRMNS=ALPHA*HRMN+(1-ALPHA)*HRMNS                            300
 370      LET ERR=HRMNS=HRMNSP                                            300
 380      LET DELFSP=GPR*HRNDOT*GIRV*ERR*DELT                             300
 390      LET DELN=DELFSP+DELN                                            300
 400      IF ABS(DELN)<1.00000E-03                                        300
 410        THEN  LET NUMP=0                                              297
 420        ELSE  LET NUMP=INT(DELN*1000)                                   3
 430          LET DELN=DELN-NUMP/1000                                       3
 440      ENDIF                                                           300
 450      LET NPOS=E(8)*1000 ; DIFF=NPOS-NUMPT                            300
 460      IF ABS(DIFF)>25                                                 300
 470        THEN  CALL MESSAGE(9,3)                                         0
 480          LET NUMPT=NPOS                                                0
 490      ENDIF                                                           300
 500      IF ABS(NUMP)>100                                                300
 510        THEN  CALL MESSAGE(9,4)                                         0
 520          LET NUMP=100                                                  0
 530      ENDIF                                                           300
 540      IF NUMP+NUMPT<0                                                 300
 550        THEN  CALL MESSAGE(9,5)                                         0
 560          LET NUMP=-NUMPT                                               0
 570      ENDIF                                                           300
 580      IF NUMP+NUMPT>=1000                                             300
 590        THEN  CALL MESSAGE(9,6)                                         0
 600          LET NUMP=1000-NUMPT                                           0
 610      ENDIF                                                           300
 620      LET NUMPT=NUMPT+NUMP                                            300
 700      CALL CAMAC(16,PULS,NUMP,0)                                     1500
 710 ENDIF                                                                300
 720 LET SCOP(1)=SCOP(1) OR 4                                             300
 730 END REMELT                                                           300
                                                                          -16
```

```
VIPER  REV  A7  12/04/78  20:02:27.0  19/04/78

*    1 PROCEDURE LIMERATIO                                                 1
   2    REM  230178BDR                                                     1
  22    REM                                                                1
  50    COMMON SPECS,NADC,ES,DELT                                          1
  60    COMMON VOLTS,V(NADC)                                               1
  70    COMMON ENG,E(ES)                                                   1
  80    COMMON BITS,CIN(4),SCOP(2)                                         1
  90    COMMON GASFLOW,GASAMAX,GASBMAX,GASCMAX,PHCSP,IZC                   1
 100    LET ACCESS((SCOP)=READA+WRITEA                                     1
 200    LET ZR=0 ; CCAO=10.314 ; GOR=2 ; ALPHA=.2 ; VOLTO=V(9)             1
 210    LET BRIX=E(3) ; FLOW=E(23) ; SADV=V(3) ; PHC=E(22)                 1
 220    IF BIT(7,SCOP(2)) LET PHCSP=E(22)                                  1
 230    LET ESF=1*DELT/(60*45)                                             1
 300 START LIMERATIO                                                     103
 310 LET BRIX=BRIX*(1-ALPHA)+ALPHA*E(3)                                  103
 320 LET FLOW=FLOW*(1-ALPHA)+ALPHA*E(23)                                 103
 330 LET SADV=SADV*(1-ALPHA)+ALPHA*V(23)                                 103
 340 LET SGPB=1.23+1.30000E-02*SADV ; SFR=FLOW*SGPB ; SLIDS=SFR*BRIX/100
                                                                         103
 400 LET LOOPSTAT=BIT(6,CIN(4))                                          103
 410 IF LOOPSTAT=0 LET FLIM=E(9)/1.183 ; FRCS=FLIM*CCAO/SLIDS            103
 420 IF ABS(VOLTO-V(9))<.1 AND LOOPSTAT=1                                103
 430   THEN  LET NOGO=(SCOP(2) AND 112)+(CIN(4) AND 448)                 103
 440     IF NOGO=560                                                     103
 450       THEN  LET IZ=0                                                102
 460         LET PHC=E(22)*ALPHA+(1-ALPHA)*PHC                           102
 470         LET ER=PHC-PHCSP                                            102
 500         LET ZA=E(24)/GASAMAX ; ZB=E(25)/GASBMAX ; ZC=E(26)/GASCMAX
                                                                          86
 510         IF ZA>.97 AND ZB>.97 LET IZ=1                               102
 520         IF ZA<.1 AND ZB<.97 AND ZC<.97                              102
 530           THEN                                                        0
 540             IF ER<0 LET IZ=1                                          0
 550             IF ER>0 AND ZC<.1                                         0
 560               THEN  CALL MESSAGE(10,1)                                0
 570                 END LIMERATIO                                         0
 580               ELSE                                                    0
 590                 IF ER>0 AND ZC>.1 LET IZ=-1                           0
 600             ENDIF                                                     0
 610         ENDIF                                                       102
 650         LET ZR=(1-ESF)*ZR+ESF*IZ                                    102
 660         LET FCR=FCRS*(1-GOR*ZR*ER)                                  102
 670         LET FLIM=FCR*SLIDS/CCAO                                     102
 680         LET SPEED=1.183*FLIM ; VOLTO=.937*SPEED+.836                102
 690       ELSE  CALL MESSAGE(10,2)                                        4
 700         LET ZR=0 ; FLIM=E(9)/1.183 ; FCRS=FLIM*CCAO/SLIDS             1
 710     ENDIF                                                           103
 720   ELSE  LET VOLTO=V(9) ; ZR=0                                         0
 730 ENDIF                                                               103
 800 CALL CDAC(2,VOLTO)                                                 1030
 810 LET SCOP(1)=SCOP(1) OR 8                                            103
 820 END LIMERATIO                                                       -16
```

```
VIPER  REV  A7  12/04/78  19:05:28.3  19/04/78

*    1 PROCEDURE GASFLOWA                                                    1
     2   REM  010278BDR                                                      1
    50   COMMON SPECS,NADC,ES,DELT                                           1
    60   COMMON ENG,E(ES)                                                    1
    70   COMMON BITS,CIN(4),SCOP(2)                                          1
    80   COMMON GASFLOW,GASAMX,GASMBMX,GASCMX,PHCSP,IZC                      1
    90   LET ACCESS(SCOP)=READA+WRITEA                                       1
    91   LET ACCESS(GASFLOW)=READA+WRITEA                                    1
   100   IF DELT<6 CALL MESSAGE(11,3)                                        1
   110   LET GIRS=30 ; GPS=.25 ; GINDEP=3.12500E-02 ; GOA=1                  1
   120   LET PHA=PHACD=PHASP=E(20) ; PHC=E(22)                               1
   130   LET VPA=.55 ; VLIM=.65 ; GASA=.5                                    1
   140   LET GASAMX=2720 ; ALPHA=.2                                          1
   150   LET GIRF=GINDEP*DELT ; GIF=1/(60*GIRF) ; GIS=1/(60*GIRS)            1
   300 START GASFLOWA                                                      200
   310 IF BIT(7,CIN(4))                                                    200
   320   THEN  LET EAPDOT=ALPHA*(E(20)-PHA)                                200
   330     LET PHA=E(20)*ALPHA+(1-ALPHA)*PHA                               200
   340     LET PHC=E(22)*ALPHA+(1-ALPHA)*PHC                               200
   350     LET ERPHC=PHC-PHCSP                                             200
   360     LET PHAC=PHASP-GOA*IZC*ERPHC                                    200
   370     LET SPPDOT=PHAC-PHACD ; PHACD=PHAC                              200
   380     LET ERAPH=PHA-PHAC                                              200
   390     LET DELFA=GPS*(EAPDOT-SPPDOT)+GIS*ERAPH*DELT                    200
   400     LET GASA=GASA+DELFA                                             200
   410     IF GASA*5436.6>GASAMX LET GASA=GASAMX/5436.6                    200
   420     IF GASA<0 LET GASA=1.00000E-02                                  200
   430     LET FLOWA=E(24)/5436.6 ; ERAF=FLOWA-GASA                        200
   440     LET DELVA=GIF*DELT*ERAF ; VPA=VPA-DELVA                         200
   500     IF VPA>VLIM LET VPA=VLIM                                        200
   510     IF FLOWA<GASAMX/5436.6                                          200
   520       THEN                                                          200
   530         IF VPA<0                                                    200
   540           THEN  CALL MESSAGE(11,2)                                    0
   550             LET VPA=0                                                 0
   560         ENDIF                                                       200
   570       ELSE  CALL MESSAGE(11,1)                                        0
   580     ENDIF                                                           200
   700 ENDIF                                                               200
   710 LET VPAD=10*(1-VPA)                                                 200
   720 IF VPAD<10*(1-VLIM) LET VPAD=10*(1-VLIM)                            200
   730 CALL CDAC(3,VPAD)                                                  2000
   740 LET SCOP(1)=SCOP(1) OR 16                                           200
   750 END GASFLOWA                                                        -16
```

```
LIST GASFLOWB


VIPER  REV  A7  12/04/78  15:50:54.3  21/04/78

#     1 PROCEDURE GASFLOWB
    2   REM  010278BDR
   50   COMMON SPECS,NADC,ES,DELT
   60   COMMON ENG,E(ES)
   70   COMMON BITS,CIN(4),SCOP(2)
   80   COMMON GASFLOW,GASAMX,GASBMX,GASCMX,PHCSP,IZC
   90   LET ACCESS((SCOP)=ACCESS((GASFLOW)=READA+WRITEA
  100   IF DELT<6 CALL MESSAGE(12,3)
  110   LET GIRS=30 ; GPS=0.25 ; GINDEP=3.12500E-02 ; GOB=1
  120   LET PHB=PHBCO=PHBSP=E(21) ; PHC=E(22)
  130   LET VPB=.55 ; VLIM=.65 ; GASB=.5
  140   LET GASBMX=2720 ; ALPHA=.2
  150   LET GIRF=GINDEP*DELT ; GIF=1/(60*GIRF) ; GIS=1/(60*GIRS)
  300 START GASFLOWB
  310 IF BIT(8,CIN(4))
  320   THEN  LET EPBDOT=ALPHA*E(21)-PHB
  330     LET PHB=E(21)*ALPHA+(1-ALPHA)*PHB
  340     LET PHC=E(22)*ALPHA+(1-ALPHA)*PHC
  350     LET ERPHC=PHC-PHCSP
  360     LET PHBC=PHBSP-GOB*IZC*ERPHC
  370     LET SPPDOT=PHBC-PHBCO ; PHBCO=PHBC
  380     LET ERBPH=PHB-PHBC
  390     LET DELPB=GPS*(EBPDOT-SPPDOT)+GIS*ERBPV*DELT
  400     LET GASB=GASB+DELFB
  410     IF GASB*5436.6>GASBMX LET GASB=GASBMX/5436.6
  420     IF GASB<0 LET GASB=1.00000E-02
  430     LET FLOWB=E(25)/5436.6 ; ERBF=FLOWB-GASB
  440     LET DELVB=GIF*DELT*ERBF ; VPB=VPB-DELVB
  500     IF VPB>VLIM LET VPB=VLIM
  510     IF FLOWB<GASBMX/5436.6
  520       THEN
  530         IF VPB<0
  540           THEN  CALL MESSAGE(12,2)
  550             LET VPA=0
  560         ENDIF
  570       ELSE  CALL MESSAGE(12,1)
  580     ENDIF
  700 ENDIF
  710 LET VPBO=10*(1-VPB)
  720 IF VPBO<10*(1-VLIM) LET VPAO=10*(1-VLIM)
  730 CALL CDAC(4,VPAO)
  740 LET SCOP(1)=SCOP(1) OR 32
  750 END GASFLOWB
```

```
VIPER  REV  A7  12/04/78  18:37:19.9  19/04/78

#    1 PROCEDURE GASFLOWC                                                  1
     2   REM  010278BDR                                                    1
    50   COMMON SPECS,NADC,ES,DELT                                         1
    60   COMMON ENG,E(ES)                                                  1
    70   COMMON BITS,CIN(4),SCOP(2)                                        1
    80   COMMON GASFLOW,GASAMX,GASBMX,GASCMX,PHCSP,IZC                     1
    90   LET ACCESS(GASFLOW)=READA+WRITEA                                  1
    91   LET ACCESS(SCOP)=READA+WRITEA                                     1
   100   LET VPC=.55 ; VLIM=.65 ; GASC=.5 ; GASCMX=1360                    1
   110   LET ALPHA=.2 ; IZC=0                                              1
   120   LET PHCSP=PHC=E(22)                                               1
   130   LET GIRS=30 ; GPS=.5 ; GINDEP=2.41700E-02                         1
   140   LET GIS=1/(60*GIRS) ; GIRF=GINDEP*DELT ; GIF=1/(60*GIRF)          1
   300 START GASFLOWC                                                    200
   310 LET ECDOT=ALPHA*(E(22)-PHC)                                       200
   320 LET PHC=E(22)*ALPHA+(1-ALPHA)*PHC ; ERC=PHC-PHCSP                 200
   330 LET DELFC=GPS*ECDOT+GIS*ERC*DELT ; GASC=GASC+DELFC                200
   340 IF GASC*2718.3>GASCMX LET GASC=GASCMX/2718.3                      200
   350 IF GASC<0 LET GASC=1.00000E-02                                    200
   360 LET FLOWC=E(26)/2718.3 ; ERFC=FLOWC-GASC                          200
   400 LET DELVC=GIF*ERFC*DELT ; VPC=VPC-DELVC                           200
   410 IF VPC>VLIM LET VPC=VLIM                                          200
   420 IF FLOWC>.96*GASCMX/2718.3                                        200
   430   THEN  LET IZC=1                                                   0
   440     CALL MESSAGE(13,1)                                              0
   450   ELSE  LET IZC=0                                                 200
   460     IF VPC<0                                                      200
   470       THEN  LET VPC=0                                               0
   480         CALL MESSAGE(13,2)                                          0
   490     ENDIF                                                         200
   500 ENDIF                                                             200
   600 LET VPCO=10*(1-VPC)                                               200
   610 IF VPCO<10*(1-VLIM) LET VPCO=10*(1-VLIM)                          200
   620 CALL CDAC(5,VPCO)                                                2000
   630 LET SCOP(1)=SCOP(1) OR 64                                         200
   650 END GASFLOWC                                                      -16
```

```
LIST FILTER.MONITOR

VIPER  REV  A7  12/04/78  16:32:11.9  21/04/78

#     1 PROCEDURE FILTER.MONITOR
    2   REM  230178BDR
   10   LET MAX=4
   50   COMMON SPECS,NADC,ES,DELT
   60   COMMON BITS,CIN(4),SCOP(2)
   70   COMMON FILTER,FILDAT(MAX*4,12),CYCLE(12)
   75   COMMON ENG,E(ES)
   80   DIM FSTIM((2,12)
   90   LET DELP=200 ; FILAR=117 ; IFILS=OSMAN=OSPRS=OSVAL=0
  100   FOR K=1 TO 12
  110     LET FSTIM(2,K)=-1 ; CYCLE(K)=1 ; FSTIM(1,K)=0
  120     FOR J=1 TO MAX*4
  130       LET FILDAT(J,K)=-1
  140     NEXT J  120
  150   NEXT K  100
  160   CALL TINT(TSTART,TPREV)
  300 START FILTER.MONITOR
  310 REGION FILTER
  320   CALL TINT(TSART,TPREV)
  330   CALL TIME(Y,MON,D,H,MIN,S)
  340   LET TNEW=H+(MIN+S/60)/60
  350   LET NSMAN=CIN(1)
  360   LET NSPRS=SHIFT(CIN(1),-12) OR SHIFT(CIN(2),4)
  370   LET NSVAL=SHIFT(CIN(2),-8) OR SHIFT(CIN(3),8)
  380   LET MAN.ONOF=OSMAN AND  NOT NSMAN
  390   LET MAN.OFON=NSMAN AND  NOT OSMAN
  400   LET PRS.OFON=NSPRS AND  NOT OSPRS
  410   LET VAL.OFON=NSVAL AND  NOT OSVAL
  500   LET B=E(3) ; Z1=111-B ; Z2=111+E(14)
  510   LET Z=-1.23399*B/Z1+246.527/Z2+659.543*B/(Z1*Z2)
  520   LET AMU=EXP(Z-2.25699)
  600   LET NUMB=12
  610   FOR I=1 TO 12
  620     LET NUMB=NUMB-BIT(I,NSMAN)
  630   NEXT I  610
  640   LET FLOW=E(23)/NUMB
  650   FOR J=1 TO 12
  660     LET FILPR=FLOW*FLOW*AMU
  670     LET FSTIM(1,J)=FSTIM(1,J)+FILPR*TSTART
  680   NEXT J  650
```

```
1000   FOR K=1 TO 12
1010     IF CYCLE(K)<MAX*4
1020       THEN
1100         IF BIT(K,MAN.ONOF)
1110           THEN  LET FSTIM(1,K)=0 ; FSTIM(2,K)=TNEW
1120             IF IFLS#0
1130               THEN  LET STRTM=TNEW-FSTIM(2,IFLS)
1140                 IF STRTM<0 LET STRTM=STRTM+24
1150                 LET KNT=CYCLE(K)*4-3
1160                 LET FILDAT(KNT,K)=STRTM
1170             ENDIF
1180             LET IFILS=K
1190         ENDIF
1200         IF BIT(K,PRS.OFON)
1220           THEN  LET KNT=CYCLE(K)*4
1230             IF FILDAT(KNT,K)=-1
1240               THEN  LET FILDAT(KNT,K)=FILBY
1250                 PRINT "FILTERABILITY FOR FILTER "K"=FILBY"
1260             ENDIF
1270         ENDIF
1300         IF BIT(K,VAL.OFON) AND FSTIM(2,K)>=0
1310           THEN  LET VPOP=TNEW-FSTIM(2,K)
1320             IF VPOP<0 LET VPOP=24
1330             LET KNT=CYCLE(K)*4-Z
1340             IF FILDAT(KNT,K)=-1 LET FILDAT(KNT,K)=VPOP
1350         ENDIF
1400         IF BIT(K,MAN.OFON) AND FSTIM(2,K)>=0
1410           THEN  LET CPOP=TNEW-FSTIM(2,K)
1420             IF CPOP<0 LET CPOP=CPOP+24
1430             LET KNT=CYCLE(K)*4-1
1440             LET FILDAT(KNT,K)=CPOP
1450         ENDIF
1460     ENDIF
1470   NEXT K 1000
1500   LET OSMAN=NSMAN ; OSPRS=NSPRS ; OSVAL=NSVAL
1510 ENDREGION FILTER
1520 END FILTER.MONITOR
```

```
VIPER  REV  A7  12/04/78  20:32:04.4  18/04/78

#     1 SUBROUTINE WCOUT(CHAN,STAT)
   10   LET C=1 ; DUM=0
  300 START WCOUT
  310 LET IC=ICHN ; N=10
  320 IF IC>31 LET N=11 ; IC=IC-32
  330 IF STAT
  340   THEN  LET F=20
  350      IF IC>16 LET F=22 ; IC=IC-16
  360   ELSE  LET F=12
  370      IF IC>16 LET F=14 ; IC=IC-16
  380 ENDIF
  400 CALL DECLR(COUT,5,N,IC)
  410 CALL CAMAC(F,COUT,DUM,Q)
  420 IF F=20 OR F=12
  430   THEN  CALL CAMAC(27,COUT,DUM,Q)
  440   ELSE  CALL CAMAC(28,COUT,DUM,Q)
  450 ENDIF
  460 IF Q#STAT PRINT "CONTACT OUT ERROR,N,A="N,IC
  470 RETURN

LIST CDAC

VIPER  REV  A7  12/04/78  20:33:36.8  18/04/78

#     1 SUBROUTINE CDAC(CHAN,VOLTS)
   10 CALL DECLR(DAC,1,1,CHAN)
   20 LET I=VOLTS*25.5
   30 CALL CAMAC(16,DAC,I,Q)
   50 RETURN
```

```
VIPER  REV  A7  12/04/78  16:45:40.4  21/04/78

*     1 SUBROUTINE MESSAGE(MESN,CHAN)
   10   LET MAX=10 ; MESMAX=13
   50   DIM PM((MAX,2),REPT((MESMAX)
   60   LET CPM=0 ; MINDAY=60*24
   70   FOR I=1 TO MESMAX
   80     LET REPT(I)=0
   90   NEXT I   70
  100 START MESSAGE
  110 LET MESNC=100*MESN+CHAN
  120 CALL TIME(Y,M,D,H,MIN,S)
  130 LET TNEW=H*60+MIN
  140 LET TOLDEST=-MINDAY ; IOLD=0
  150 FOR I=1 TO CPM
  160   IF PM(I,1)=MESNC
  170     THEN  LET TDIF=TNEW-PM(I,2)/100
  180       IF TDIF<0 LET TDIF=TDIF+MINDAY
  190       IF TDIF>REPT(MESN)
  200         THEN  LET NREPS=PM(I,2)-100*INT(PM(I,2)/100)
  210           LET PM(I,2)=TNEW*100
  220           CALL PRINT.MESSAGE(MESNC,TDIF,NREPS)
  230         ELSE  LET PM(I,2)=PM(I,2)+1
  240       ENDIF
  250       RETURN
  260   ENDIF
  270   LET TCUR=PM(I,2)/100
  280   IF TNEW<TCUR LET TCUR=TCUR-MINDAY
  290   IF TOLDEST<TCUR LET TOLDEST=TCUR ; IOLD=I
  300 NEXT I  150
  310 CALL PRINT.MESSAGE(MESNC,0,0)
  320 IF CPM<MAX LET CPM=CPM+1 ; IOLD=CPM
  330 LET PM(IOLD,1)=MESNC ; PM(IOLD,2)=TNEW*100
  340 RETURN
```

```
#


LIST STARTUP


VIPER  REV  A7  3/04/78   13:46:34.8  06/04/78

#     1 PROCEDURE STARTUP
    50 COMMON SPECS,NADC,ES,DELT
    60 LET ACCESS(SPECS)=READA+WRITEA
    70 LET NADC=30 ; ES=30
    80 LET DELT=30 ; DELTCS=5
    90 CALL TIME(Y,M,D,HOUR,MIN,S)
   100 RUN SCANCS EVERY DELTCS SECS
   110 RUN SCANADC EVERY DELT SECS
   120 RUN WATCH.DOG EVERY DELT SECS
   130 RUN SERVOHOUR EVERY 1 HOURS  AT HOUR+1:0
   140 LET NEXTSHIFT=8*INT(HOUR/8)+6
   150 RUN SERVO8HOUR EVERY 8 HOURS  AT NEXTSHIFT:0
   160 RUN FILTER.REPORT EVERY 8 HOURS  AT NEXTSHIFT:0
   170 PRINT "HULETTS FACTORY SOFTWARE STARTED UP AT" ;
   180 CALL PTAD
   190 END STARTUP


LIST SHUTDOWN


VIPER  REV  A7  3/04/78   14:02:42.4  06/04/78

#     1 PROCEDURE SHUTDOWN
    10 TURNOFF SCANCS
    20 TURNOFF SCANADC
    30 TURNOFF WATCH.DOG
    40 TURNOFF SERVOHOUR
    50 TURNOFF SERVO8HOUR
    60 TURNOFF FILTER.REPORT
    70 END SHUTDOWN
```

```
VIPER  REV  A7  12/04/78  10:39:45.7  19/04/78

#    1 SUBROUTINE FILTERCOEF(W,DAMP,DELT,CB,CC,CD,CE)
  100 LET W0=SQR(1-DAMP*DAMP)*W
  110 LET A=W*DAMP
  120 LET EAT=EXP(-A*DELT)
  130 IF W0<1.00000E-06
  140   THEN  LET THETA=1.5708 ; CA=EAT
  150   ELSE  LET THETA=ATN(-A/W0)
  160      LET CA=EAT*COSW0*DELT+THETA)/COSTHETA)
  170 ENDIF
  200 LET CB=2*EAT*COSW0*DELT)
  210 LET CC=EAT*EAT ; CD=1+CA-CB ; CE=CC-CA
  220 RETURN
```

```
VIPER  REV  A7  12/04/78  14:26:53.1  18/04/78

#    1 SUBROUTINE ENGLIMITS
   50 COMMON SPECS,MADC,ES,DELT
   60 COMMON ENGLIM,EL(ES,2)
   70 LET ACCESS((ENGLIM)=WRITEA
  200 FOR I=1 TO ES
  210   LET EL(I,1)=-1.00000E-38 ; EL(I,2)=1.00000E+38
  220 NEXT I  200
  225 RETURN
  230 LET EL(3,1)=60 ; EL(3,2)=80
  240 LET EL(7,1)=0 ; EL(7,2)=1
  250 LET EL(10,1)=63 ; EL(10,2)=70
  260 LET EL(18,1)=75 ; EL(18,2)=90
  270 LET EL(20,1)=8.59999 ; EL(20,2)=9.3
  280 LET EL(21,1)=8.59999 ; EL(21,2)=9.3
  290 LET EL(22,1)=7.79999 ; EL(22,2)=8.59999
  300 RETURN
```

```
LIST SERVOHOUR


VIPER  REV  A7  3/04/78   12:33:00.4  06/04/78

#     1 PROCEDURE SERVOHOUR
   50    COMMON SERVOD,DUM1(4),MASS.HOUR,MASS8H(8),DUM2(3)
   60    LET ACCESS(SERVOD)=READA+WRITEA
   70    FOR I=1 TO 8
   80      LET MASS8H(I)=0
   90    NEXT I   70
  100 START SERVOHOUR
  110 REGION SERVOD
  120    FOR I=8 TO 2 STEP -1
  130      LET MASS8H(I)=MASS8H(I-1)
  140    NEXT I  120
  150    LET MASS8H(1)=MASS.HOUR
  160    PRINT "SOLIDS MELT RATE="MASS.HOUR" TONS/HOUR"
  170    LET MASS.HOUR=0
  180 ENDREGION SERVOD
  200 END SERVOHOUR



LIST SERVO8HOUR


VIPER  REV  A7  3/04/78   12:41:28.8  06/04/78

#     1 PROCEDURE SERVO8HOUR
   50    COMMON SERVOD,DUM(5),MASS.8H(8),MASS.SHIFT(3)
   60    LET ACCESS(SERVOD)=READA+WRITEA
   70    FOR I=1 TO 3
   80      LET MASS.SHIFT(I)=0
   90    NEXT I   70
  100    LET SHIFTN=1
  110 START SERVO8HOUR
  120 REGION SERVOD
  130    LET MASS=0
  140    FOR I=1 TO 8
  150      LET MASS=MASS+MASS.8H(I)
  160    NEXT I  140
  170    LET MASS.SHIFT(SHIFTN)=MASS
  180    LET SHIFTN=SHIFTN+1
  190    IF SHIFTN>3 LET SHIFTN=1
  200 ENDREGION SERVOD
  210 END SERVO8HOUR
```

```
VIPER  REV  A7  12/04/78  16:26:25.4  21/04/78

#     1 PROCEDURE FILTER.REPORT
   10   LET MAX=4 ; M=4
   50   COMMON FILTER,FILDAT(MAX*4,12),CYCLE(12)
   60 START FILTER.REPORT
   70 DIM AV((12,4),TAV((M)
   80 REGION FILTER
   90   PRINT "FILTER DATA FOR SHIFT ENDING AT" ;
  100   CALL PTAD
  110   FOR L=1 TO 72
  120     PRINT "X" ;
  130   NEXT L  110
  140   PRINT
  150   FOR I=1 TO 4
  160     FOR J=1 TO 12
  170       LET SUM=0
  180       FOR K=0 TO CYCLE(J)-1
  190         LET SUM=SUM+FILDAT(4*K+I,J)
  200       NEXT K  180
  210       LET AV(J,I)=SUM/CYCLE(J)
  220     NEXT J  160
  230   NEXT I  150
  240   FOR M=1 TO 4
  250     LET TAV(M)=0
  260     FOR N=1 TO 12
  270       LET TAV(M)=TAV(M)+AV(N,M)
  280     NEXT N  260
  290   NEXT M  240
  300   PRINT TAB(2) ;
  310   FOR L=1 TO 12
  320     PRINT L,
  325   NEXT L  310
  330   PRINT
  340   PRINT
  400   FOR I=0 TO MAX-1
  410     FOR K=1 TO 4
  420       FOR J=1 TO 12
  430         LET X=INT(FILDAT(4*I+K,J)+.5)
  440         IF X=-1 PRINT "        " ;
  450         IF X#-1 PRINT X ;
  460       NEXT J  420
  470     NEXT K  410
  472     PRINT
  474   NEXT I  400
  480   FOR I=1 TO 12
  483     LET CYCLE(I)=1
  486     FOR J=1 TO MAX*4
  489       LET FILDAT(J,I)=-1
  492     NEXT J  486
  495   NEXT I  480
  500 ENDREGION FILTER
  505 PRINT
```

```
510 PRINT
520 PRINT "AVERAGES FOR EACH FILTER"
530 PRINT
540 PRINT "FILTER NO" ; TAB(15) ; AV.ST.INT. ; TAB(30) ;
550 PRINT "AV. C. P. PER" ; TAB(45) ; "AV. CYC. TIME" ; TAB(60) ; "AV. FILTBY"
560 PRINT
570 FOR I=1 TO 12
580   PRINT I,AV(I,1),AV(I,2),AV(I,3),AV(I,4)
590 NEXT I  570
592 PRINT
594 PRINT
600 PRINT "OVER-ALL AVERAGES"
610 PRINT
620 PRINT "AVERAGE START INTERVAL=" ; TAV(1)
630 PRINT "AVERAGE TIME TO VALVE FULL OPEN=" ; TAV(2)
640 PRINT "AVERAGE FILTER CYCLE TIME=" ; TAV(3)
650 PRINT "AVERAGE FILTERABILITY=" ; TAV(4)
730 DIM AV((0),TAV((0)
740 END FILTER.REPORT
```

```
VIPER  REV  A7  12/04/78  15:47:29.7  21/04/78

#    1 PROCEDURE CLOOP
     2   REM  9-11-77
    50   COMMON BITS,CIN(4),SCOP(2)
    60   LET CO=0
   300 START CLOOP
   310 LET C=SHIFT(CIN(4),-1) ; DIF=XOR(C,CO) ; CO=C
   320 LET MESN=4+BIT(1,C)                                    MESN=4 OR 5
   340 IF BIT(1,DIF) CALL MESSAGE(MESN,0)
   350 FOR J=1 TO NLOOP
   360   IF BIT(J,SCOP(2)) AND BIT(J,DIF)
   362     THEN  LET MESN=4+BIT(J,C)
   364       CALL MESAGE(MESN,J)
   366   ENDIF
   370 NEXT J  350
   380 SAVE BITS
   400 END CLOOP

CLOOP
#
```

```
VIPER  REV  A7  12/04/78  19:52:58.2  19/04/78

*     1 SUBROUTINE PRINT.MESSAGE(MESNC,TDIF,NREPS)
   50   COMMON SPECS,NADC,ES,DELT
   60   COMMON ENG,E(ES)
   70   COMMON VOLTS,V(NADC)
   80   COMMON ENGLIM,EL(ES,2)
   90   LET LU=1
  300 START PRINT.MESSAGE
  310 LET MESN=INT(MESNC/100) ; J=MESNC-100*MESN
  400 CASE MESN=1
  410   PRINT (LU)"ADC"J" (" ;
  420   CALL PRINT.CHAN.NAM(J,LU)
  430   PRINT (LU)") OUT OF RANGE,="V(J)" VOLTS" ;
  500 CASE MESN=2
  510   PRINT (LU)"ENG"J" : " ;
  520   CALL PRINT.CHAN.NAM(J,LU)
  530   PRINT (LU)" OUT OF RANGE,VALUE"E(J)" LIMITS ARE"EL(J,1) ; EL(J,2)
  600 CASE MESN=3
  610   CALL PRINT.PROG.NAM(J,LU)
  620   PRINT (LU)"IS NOW ON LINE" ;
  700 CASE MESN=4
  710   CALL PRINT.PROG.NAM(J,LU)
  720   PRINT (LU)"HAS GONE OFF-LINE" ;
  800 CASE MESN=5
  810   CALL PRINT.PROG.NAM(J,LU)
  820   PRINT (LU)"CONTROL ROOM SWITCH SET TO LOCAL MODE" ;
  900 CASE MESN=6
  910   CALL PRINT.PROG.NAM(J,LU)
  920   PRINT (LU)"CONTROL ROOM SWITCH SET TO COMPUTER MODE" ;
 1000 CASE MESN=7
 1010   PRINT (LU)"SATFLOW"J" :" ;
 1020   IF J=1 PRINT (LU)"SAT SUPPLY TANK LOW:"E(1)" %FULL" ;
 1030   IF J=2 PRINT (LU)"SAT SUPPLY TANK HIGH:"E(1)" %FULL" ;
 1040   IF J=3 PRINT (LU)"AUTOFILTER SUPPLY TANK LOW:"E(2)" %FULL" ;
 1050   IF J=4 PRINT (LU)"AUTOFILTER SUPPLY TANK HIGH:"E(2)" %FULL" ;
 1060   IF J=6 PRINT (LU)"SAT FLOW FULL OPEN" ;
 1070   IF J=7 PRINT (LU)"CALCULATED VALVE POS DIFFERS FROM ACTUAL" ;
 1080   IF J=8 PRINT (LU)"SAT FLOW VALVE MOVING TOO FAST(>10%)" ;
 1090   IF J=9 PRINT (LU)"CHECK VALUES OF ERROR DERIVATIVES"
 1100 CASE MESN=8
 1110   PRINT (LU)"CLFLOW"J" :" ;
 1120   IF J=1 PRINT (LU)"CLOUDY LIQUOR TANK LOW:"E(27)"%FULL" ;
 1130   IF J=2 PRINT (LU)"CLOUDY LIQUOR TANK HIGH:"E(27)"%FULL" ;
 1140   IF J=3 PRINT (LU)"LIQUOR RETURNS VALVE POS CHANGE>10%" ;
 1150   IF J=4 PRINT (LU)"LIQUOR RETURNS VALVE CLOSED" ;
 1160   IF J=5 PRINT (LU)"LIQUOR RETURNS VALVE FULL OPEN" ;
 1170   IF J=6 PRINT (LU)"CHECK CLT LEVEL DERIVATIVE" ;
 1200 CASE MESN=9
 1210   PRINT (LU)"REMELT"J" :" ;
 1220   IF J=1 PRINT (LU)"REMELT TANK HIGH:"E(28)"%FULL" ;
 1230   IF J=2 PRINT (LU)"REMELT TANK LOW:"E(28)"%FULL" ;
 1240   IF J=3 PRINT (LU)"CALCULATED FLOW SETPOINT AND FEEDBACK DIFFER" ;
 1250   IF J=4 PRINT (LU)"CALCULATED FLOW SETPOINT CHANGE>10%" ;
 1255   IF J=5 PRINT (LU)"VALVE CLOSED" ;
 1260   IF J=6 PRINT (LU)"VALVE FULL OPEN" ;
```

```
1300 CASE MESN=10
1310   PRINT (LU)"LIMERATIO"J" :" ;
1320   IF J=1 PRINT (LU)"PHC>PHCSP AND CGAS<10%" ;
1330   IF J=2 PRINT (LU)"GAS CONTROL OFF-NO LIME CONTROL" ;
1400 CASE MESN=11
1410   PRINT (LU)"GASFLOWA"J" :" ;
1420   IF J=1 PRINT (LU)"A SAT OUT OF GAS;FLOW="E(24)"CFM" ;
1430   IF J=2 PRINT (LU)"A SAT GAS SUPPLY VALVE CLOSED" ;
1440   IF J=3 PRINT (LU)"DELT TOO SMALL" ;
1500 CASE MESN=12
1510   PRINT (LU)"GASFLOWB"J" :" ;
1520   IF J=1 PRINT (LU)"B SAT OUT OF GAS;FLOW="E(25)"CFM" ;
1530   IF J=2 PRINT (LU)"B SAT GAS SUPPLY VALVE CLOSED" ;
1540   IF J=3 PRINT (LU)"DELT TOO SMALL" ;
1600 CASE MESN=13
1610   PRINT (LU)"GASFLOWC"J" :" ;
1620   IF J=1 PRINT (LU)"C SAT OUT OF GAS;FLOW="E(26)"CFM" ;
1630   IF J=2 PRINT (LU)"C SAT GAS SUPPLY VALVE CLOSED" ;
1640   IF J=3 PRINT (LU)"DELT TOO SMALL" ;
1700 CASE MESN=MESN
1710   PRINT (LU)"MESSAGE:MESN="MESN",*UNKNOWN MESSAGE*"
1800 ENDCASE MESN
1810 CALL PTAD
1820 IF NREPS#0 PRINT (LU)"    ("NREPS"OCCURENCES IN LAST"TDIF" MINS)"
1830 RETURN
```

```
VIPER  REV  A7  12/04/78  20:20:57.8  18/04/78

#     1 SUBROUTINE PRINT.PROG.NAM(J,LU)
   100 IF J<0 OR J>7 PRINT (LU)"**UNKNOWN NAME**" ;
   110 IF J=0 PRINT (LU)" MASTER " ;
   120 IF J=1 PRINT (LU)" SATFLOW " ;
   130 IF J=2 PRINT (LU)" ELFLOW " ;
   140 IF J=3 PRINT (LU)" REMELT " ;
   150 IF J=4 PRINT (LU)" LIMERATIO " ;
   160 IF J=5 PRINT (LU)" GASFLOWA " ;
   170 IF J=6 PRINT (LU)" GASFLOWB " ;
   180 IF J=7 PRINT (LU)" GASFLOWC " ;
   200 RETURN

LIST PRINT.CHAN.NAM

VIPER  REV  A7  12/04/78  20:22:40.1  18/04/78

#     1 SUBROUTINE PRINT.CHAN.NAM(J,LU)
   100 IF J<1 OR J>30 PRINT (LU)"*UNKNOWN CHANNEL*" ;
   110 IF J=1 PRINT (LU)"SAT SUPPLY TANK LEVEL" ;
   120 IF J=2 PRINT (LU)"AUTU-FILTER SUPPLY TANK LEVEL" ;
   130 IF J=3 PRINT (LU)"POLISHING BRIX" ;
   140 IF J=4 PRINT (LU)"BROWN LIQUOR BRIX" ;
   150 IF J=5 PRINT (LU)"PRESSED LIQUOR TANK LEVEL" ;
   160 IF J=6 PRINT (LU)"      " ;
   170 IF J=7 PRINT (LU)"130K FEEDBACK" ;
   180 IF J=8 PRINT (LU)"PUMPED FILTER SUPPLY PRESSURE" ;
   190 IF J=9 PRINT (LU)"%CO2" ;
   200 IF J=10 PRINT (LU)"      " ;
   210 IF J=11 PRINT (LU)"A SAT TEMP" ;
   220 IF J=12 PRINT (LU)"B SAT TEMP" ;
   230 IF J=13 PRINT (LU)"C SAT TEMP" ;
   240 IF J=14 PRINT (LU)"AFS EXIT TEMP" ;
   250 IF J=15 PRINT (LU)"SAT TANK TEMP" ;
   260 IF J=16 PRINT (LU)"FINE LIQUOR TEMP" ;
   270 IF J=17 PRINT (LU)"REMELT TEMP" ;
   280 IF J=18 PRINT (LU)"SWEET WATER TEMP" ;
   290 IF J=19 PRINT (LU)"      " ;
   300 IF J=20 PRINT (LU)"A SAT PH" ;
   310 IF J=21 PRINT (LU)"B SAT PH" ;
   320 IF J=22 PRINT (LU)"C SAT PH" ;
   330 IF J=23 PRINT (LU)"REMELT FLOW" ;
   340 IF J=24 PRINT (LU)"A GAS FLOW" ;
   350 IF J=25 PRINT (LU)"B GAS FLOW" ;
   360 IF J=26 PRINT (LU)"C GAS FLOW" ;
   370 IF J=27 PRINT (LU)"CLOUDY LIQUOR RETURNS TANK LEVEL" ;
   380 IF J=28 PRINT (LU)"REMELT TANK LEVEL" ;
   400 RETURN
```

PAGE 0001 FTN. 8:23 AM SUN., 8 AUG., 1976

```
FTN4,L,T
      PROGRAM PACIR(1,10),03107??? 231277BDR
C-----------------------------------------------------------------------
C  PACIR - PACE THE MASTER SAMPLING RATE
C
C-----------------------------------------------------------------------
C
C  PACIR CLEARS RESOURCE NUMBERS 1 TO 3  EACH TIME IT RUNS
C  THESE ARE USED TO PACE THE SCAN PROGRAMS
C     IRN(1) - SCAD(SCAN A-TO-D'S)
C     IRN(2) - SCCS(SCAN CONTACT STATUS)
C     IRN(3) - WCHDG(WATCH-DOG)
C  THE RESOURCE NUMBERS ARE ALLOCATED BY STRUP
C
C PACER SUSPENDS ITSELF IF ISMUL(1) IS NEGATIVE. ISMUL(1) IS SET ON AND
C OFF BY HANGO WHICH IS EITHER RUN DIRECTLY OR SCHEDULED BY STRUP
C-----------------------------------------------------------------------
C
C                    ------ COMMON  ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C   ENG    - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C   ADCV   - A/D VOLTAGES (UPDATED BY SCAD)
C   CDACV  - D/A VOLTAGES (UPDATED BY CDAC)
C
C   SAFCOD- SATURATOR FLOW CONTROL DATA
C   CLFLOD- CLOUDY LIQUOR FLOW DATA
C   REMLTD- REMELT CONTROL DATA
C   CLIMED- CONTROL LIME DATA
C   GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C   GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C   GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C   FILCYD- FILTER CYCLE MONITER DATA
C   SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C   ISAMT  - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C   ISMUL  - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C   IRN    - RESOURCE NUMBERS
C   ICIN   - CONTACT STATUS IN (UPDATED BY SCCS)
C   ICOUT  - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C   ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C   ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C   ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C-----------------------------------------------------------------------
C
C
      CALL SWITF(15)
      IF(ISMUL(1).LT.0) CALL EXEC(6,0,2)
```

```
PAGE 0002  PACIR  8:23 AM  SUN.,  8  AUG., 1976

C                           SUSPEND AND REMOVE FROM TIME LIST
C
C    CLEAR RESOURCE NUMBERS 1 TO 3 WHEN DUE AS INDICATED BY ISMUL 2 TO 4
      DO 10 IRNI=1,3
        ISMUL(IRNI+17) = ISMUL(IRNI+17) + 1
        IF(ISMUL(IRNI+17).LT.ISMUL(IRNI+1))GOTO 10
        CALL RNRQ(4,IRN(IRNI),ISTAT)
        ISMUL(IRNI+17) = 0
   10 CONTINUE
      END


FTN4 COMPILER: HP92060-16092 REV. 1726

**  NO WARNINGS **  NO ERRORS **   PROGRAM = 00081       COMMON = 00758
```

```
PAGE 0001  FTN.   8:26 AM  SUN.,  8  AUG., 1976

FTN4,L,T
      PROGRAM SCCS,1,20
C-----------------------------------------------------------------------
C
C       SCCS - SCAN CONTACT STATUS
C       VERSION : 9-11-1977
C-----------------------------------------------------------------------
C
C CLEARS RESOURCE NOS 7 & 8.
C CLOOP IS CALLED BY EXEC CALL IN ORDER TO BE ABLE TO PARSE PARAMETERS
C TO IT AND TO ALLOW IT TO BE DORMANT WHEN NOT REQUIRED.
C SCCS LOCKS ON RESOURCE NO. 2 WHICH IS CLEARED BY PACIR AND THEN CHECKS
C ITS OWN RUN FREQUENCY AGAINST ISMUL(3).
C I.E.  SCCS RUNS EVERY .....(ISAMT*ISMUL(3))   SECONDS
C
C
C                ICIN   BIT-SIGNIFICANCE :
C
C             BITS 1 TO 16 IN ICIN(1) - FILCY
C             BITS 1 TO  8 IN ICIN(2) - FILCY
C             BITS 2 TO 16 IN ICIN(4) - CLOOP
C             BIT 16 IN ICIN(3) - SERVO
C             BIT 1  IN ICIN(4) - SERVO
C
C-----------------------------------------------------------------------
C
C                    ------ COMMON ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C   ENG   - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C   ADCV  - A/D VOLTAGES (UPDATED BY SCAD)
C   CDACV - D/A VOLTAGES (UPDATED BY CDAC)
C
C   SAFCOD- SATURATOR FLOW CONTROL DATA
C   CLFLOD- CLOUDY LIQUOR FLOW DATA
C   REMLTD- REMELT CONTROL DATA
C   CLIMED- CONTROL LIME DATA
C   GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C   GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C   GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C   FILCYD- FILTER CYCLE MONITER DATA
C   SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C   ISAMT  - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C   ISMUL  - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C   IRN    - RESOURCE NUMBERS
C   ICIN   - CONTACT STATUS IN (UPDATED BY SCCS)
C   ICOUT  - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C   ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
```

PAGE 0002 SCCS 8:26 AM SUN., 8 AUG., 1976

```
C     ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C     ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C-----------------------------------------------------------------------
C
      INTEGER CLOOP(3)
C
      DATA CLOOP/2HCL,2HOO,1HP/
C
C
C
C       INITIALISE ALL BITS TO ZERO.
C
      DO 10 I=1,4
      ICOUT(I)=0
   10 CONTINUE
      IFILO1=0
      IFILO2=0
      IFILO3=0
      ISCOP(3) = 0
      ISRVO1=0
      ISRVO2=0
      ICNTO =0
      JCNTO =0
      IRUN=0
C
      CALL DECLR(ICSW1,1,12,0)
      CALL DECLR(ICSW2,1,12,1)
      CALL DECLR(ICSW3,1,13,0)
      CALL DECLR(ICSW4,1,13,1)
C
C       WRITE LAM MASK FOR ALL 64 CHANNELS:
C
   40 CALL CAMAC(18,ICSW1,IDUM,IQ)
      CALL CAMAC(18,ICSW2,IDUM,IQ)
      CALL CAMAC(18,ICSW3,IDUM,IQ)
      CALL CAMAC(18,ICSW4,IDUM,IQ)
C
C WAIT ON RESOURCE NUMBER
C
      CALL RNRQ(2,IRN(2),ISTAT)
C                    GLOBAL SET TO PERMIT PACIR TO CLEAR
C
      IRUN=IRUN + 1
      IF(IRUN.LT.ISMUL(3))GOTO 40
C                    CHECK RUN FREQUENCY FOR SCCS
      IRUN=0
C
      CALL SWITF(13)
C
C               READ STATUS OF 64 CONTACTS:
C
      CALL CAMAC(0,ICSW1,ICIN(1),IQ)
      CALL CAMAC(0,ICSW2,ICIN(2),IQ)
      CALL CAMAC(0,ICSW3,ICIN(3),IQ)
```

PAGE 0003 SCCS 8:26 AM SUN., 8 AUG., 1976

```
      CALL CAMAC(0,ICSW4,ICIN(4),IQ)
C
C           MASK OFF SPECIFIC PORTIONS:
C
C                          FOR FILCY :
      IFILN1=ICIN(1)
      IFILN2=ICIN(2)
      IFILN3 = IAND(ICIN(3),000017B)
C
C                          FOR CLOOP :
      ICNTN = IAND(ICIN(4),177776B)
      JCNTN = ISHFT(ICNTN,-1)
C
C                          FOR SERVO :
      ISRVN1 = IAND(ICIN(3),100000B)
      ISRVN2 = IAND(ICIN(4),000001B)
C
C      LOOK FOR CHANGES IN STATUS & RELEASE APPROPRIATE RESOURCE NO.
C
      IFILD=IXOR(IFILO1,IFILN1)+IXOR(IFILO2,IFILN2)+IXOR(IFILO3,IFILN3)
      ISRVD=IXOR(ISRVO1,ISRVN1)+IXOR(ISRVO2,ISRVN2)
      ICNTD=IXOR(ICNTO,ICNTN)
C
C
      IF(IFILD.NE.0) CALL RNRQ(4,IRN(7),ISTAT)
C                    CLEAR FILCY TO RUN
      IF(ISRVD.NE.0)CALL RNRQ(4,IRN(8),ISTAT)
C                    CLEAR SERVO TO RUN
      IF(ICNTD.NE.0)CALL EXEC(24,CLOOP,JCNTO,JCNTN)
C                 QUEUE SCHEDULE WITHOUT WAIT
C
C           UPDATE OLD STATUS WORDS:
C
      IFILO1=IFILN1
      IFILO2=IFILN2
      IFILO3=IFILN3
      ISRVO1=ISRVN1
      ISRVO2=ISRVN2
      ICNTO=ICNTN
      JCNTO =JCNTN
      ISCOP(3)=JCNTN
C
      GOTO 40
      END
```

FTN4 COMPILER: HP92060-16092 REV. 1726

** NO WARNINGS ** NO ERRORS ** PROGRAM = 00332 COMMON = 00758

PAGE 0001 FTN. 8:24 AM SUN., 8 AUG., 1976

```
FTN4,L,T
      PROGRAM SCAD1(1,20),03107??? 231277BDR
C
C-----------------------------------------------------------------------
C
C     SCAD1 - SCAN A TO D CONVERTOR
C
C-----------------------------------------------------------------------
C
C    THIS IS A PROGRAM FOR SCANNING 64 ADC CHANNELS. EACH CHANNEL
C    IS ADDRESSED INDIVIDUALLY. THE VALUES ARE CONVERTED  TO'
C    VOLTS AND STORED IN THE COMMON ARRAY ADCV(I).
C
C                 N2=MUX2 STATION NUMBER
C                 N1=MUX1 STATION NUMBER
C                 N =ADC  STATION NUMBER
C                 IC= CRATE NUMBER
C
C
C SCAD1 RUNS AFTER A TIME INTERVAL DETERMINED BY THE FREQUENCY OF PACIR
C AND A MULTIPLE THEREOF (I.E. ISMUL(2)). PACIR CLEARS SCAD1 EACH TIME
C IT RUNS AND SCAD1 CHECKS ITS OWN RUN FREQUENCY. SCAD1 ALSO REGULATES
C THE RUN FREQUENCIES OF SDATA AND ENGUN BASED ON THEIR SEPERATE AND
C INDIVIDUAL MULTIPLES (I.E.ISMUL(5) & ISMUL(6) RESP.) OF SCAD1'S RUN
C INTERVAL.
C I.E.   ENGUN RUNS EVERY .....(ISAMT*ISMUL(2)*ISMUL(6))  SECONDS
C        SDATA RUNS EVERY .....(ISAMT*ISMUL(2)*ISMUL(5))  SECONDS
C        SCAD1 RUNS EVERY .....(ISAMT*ISMUL(2))           SECONDS
C
C
C I.E. PACIR SETS THE SAMPLING TIME OF ALL PROGRAMS.
C AT THE END OF SCAN, SCAD1 CLEARS THE FOLLOWING RESOURCE NUMBERS :-
C     IRN(4) - EVERY SCAN  (SPARE)
C     IRN(5) - EVERY ISMUL5 SCANS
C     IRN(6) - EVERY ISMUL6 SCANS
C THIS RELEASES THE WAITING CONTROL PROGRAMS (VIZ. SDATA & ENGUN)
C
C-----------------------------------------------------------------------
C
C                     ------ COMMON ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C   ENG   - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C   ADCV  - A/D VOLTAGES (UPDATED BY SCAD)
C   CDACV - D/A VOLTAGES (UPDATED BY CDAC)
C
C   SAFCOD- SATURATOR FLOW CONTROL DATA
C   CLFLOD- CLOUDY LIQUOR FLOW DATA
C   REMLTD- REMELT CONTROL DATA
```

```
PAGE 0002  SCAD1  8:24 AM  SUN.,  8  AUG., 1976

C     CLIMED- CONTROL LIME DATA
C     GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C     GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C     GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C     FILCYD- FILTER CYCLE MONITER DATA
C     SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C     ISAMT - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C     ISMUL - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C     IRN   - RESOURCE NUMBERS
C     ICIN  - CONTACT STATUS IN (UPDATED BY SCCS)
C     ICOUT - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C     ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C     ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C     ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C-----------------------------------------------------------------------
C
        N2=6
        N1=7
        N=8
        IC=1
C
C                                   CAMAC DECLARATIONS
        CALL DECLR(MUX1A,IC,N2,0)
        CALL DECLR(MUX1B,IC,N2,1)
        CALL DECLR(MUX2A,IC,N1,0)
        CALL DECLR(MUX2B,IC,N1,1)
        CALL DECLR(NADC,IC,N,0)
C
C
C
C   MAIN DATA SAMPLING LOOP
C
C   WAIT FOR PACER TO CLEAR RESOURCE NUMBER 1 (IRN1)
C
C
  10    CALL RNRQ(2,IRN(1),ISTAT)
C                       GLOBAL SET SO THAT PACER CAN CLEAR IT
        CALL SWITF(14)
C
        DO 1000 I=1,64
C
C
         IF(I.GT.16)GOTO 200
C                              ELSE   MUX1A
           J=17*(I-1)
          MUX1=MUX1A
           MUX2=MUX1A
           GOTO 800
C
   200   IF(I.GT.32)GOTO 400
C                              ELSE   MUX1B
           J=4352*(I-17)
           MUX1=MUX1B
```

PAGE 0003 SCAD1 8:24 AM SUN., 8 AUG., 1976

```
               MUX2=MUX1A
               GOTO 800
C
   400  IF(I.GT.48)GOTO 600
C                            ELSE  MUX2A
              J=17*(I-33)
              MUX1=MUX2A
              MUX2=MUX2A
              GOTO 800
C
C                                  MUX2B
   600        J=4352*(I-49)
               MUX1=MUX2B
               MUX2=MUX2A
C
C SET UP MULTIPLEXOR CHANNEL
C
   800        CALL CAMD1(26,MUX1,IDUM,IQ,IERR)
                IF(IERR.GE.1)CALL CAMER(IERR,26,MUX2B)
              CALL CAMD1(16,MUX2,J,IQ,IERR)
                 IF(IERR.GE.1)CALL CAMER(IERR,16,MUX2A)
C
        CALL CAMD1(2,NADC,IDUM,IQ,IERR)
C                    START CONVERSION
          IF(IERR.GE.1)CALL CAMER(IERR,2,NADC)
C
       CALL EXEC(12,0,1,0,-2)
C                    WAIT FOR CONVERSION TO COMPLETE
C                    INCREASED FROM 10 TO 20 MS 24-11-76 BY A.D.HEHER
C                    TO AVOID INTERMITTANT CONVERSION ERRORS
        CALL CAMD1(0,NADC,ID,IQ,IERR)
C                    READ DATA
          IF(IERR.GE.1)CALL CAMER(IERR,0,NADC)
C                               CONVERT TO VOLTS
        ADCV(I)=(ID-32)/3273.5
C
  1000 CONTINUE
C
       ISMUL(22)=ISMUL(22)+1
       ISMUL(21)=ISMUL(21)+1
       CALL RNRQ(4,IRN(4),ISTAT)
C           DUMMY RESOURCE NUMBER
       IF(ISMUL(21).LT.ISMUL(5)) GOTO 2000
         ISMUL(21)=0
         CALL RNRQ(4,IRN(5),ISTAT)
C           RELEASES SDATA
 2000  IF(ISMUL(22).LT.ISMUL(6)) GOTO 2020
         ISMUL(22)=0
         CALL RNRQ(4,IRN(6),ISTAT)
C           RELEASES ENGUN
 2020  CONTINUE
C                    CLEAR - RELEASES CONTROL PROGRAMS
       GOTO 10
       END
```

PAGE 0004 SCAD1 8:24 AM SUN., 8 AUG., 1976

FTN4 COMPILER: HP92060-16092 REV. 1726

** NO WARNINGS ** NO ERRORS ** PROGRAM = 00330 COMMON = 00758

```
PAGE 0001  FTN.   8:27 AM  SUN.,  8  AUG., 1976

FTN4,L,T
      PROGRAM ENGUN(2,30),180178BDR 230178BDR 010278BDR
C**************************************************************************
C     THIS PROGRAM CALCULATES THE ENGINEERING UNITS OF THE FACTORY
C     DATA STORED AS VOLTS IN THE ADCV ARRAY.
C     THE PROGRAM'S RESOURCE NUMBER IS RELEASED BY "SCAD".
C****************************************************************************
C
C
C   ***** OUT OF SPECIFICATION ALARM MESSAGES *****
C
C       1 - "A" SATURATOR TEMPERATURE ABNORMAL
C       2 - "B" SATURATOR TEMPERATURE ABNORMAL
C       3 - "C" SATURATOR TEMPERATURE ABNORMAL
C       4 - AUTOFILTER SUPPLY TANK TEMPERATURE ABNORMAL
C       5 - SATURATOR SUPPLY TANK TEMPERATURE OUT OF RANGE
C       6 - FINE LIQUOR TEMPERATURE ABNORMAL
C       7 - REMELT LIQUOR TEMPERATURE ABNORMAL
C       8 - SWEET WATER TEMPERATURE OUT OF RANGE
C
C       9 - POLISHING BRIX MEASUREMENT OUT OF RANGE
C      10 - BROWN LIQUOR BRIX MEASUREMENT OUT OF RANGE
C
C      11 - SATURATOR SUPPLY TANK LEVEL OUT OF RANGE
C      12 - AUTOFILTER SUPPLY TANK LEVEL OUT OF RANGE
C      13 - PRESSED LIQUOR TANK LEVEL OUT OF RANGE
C      14 - CLOUDY LIQUOR TANK LEVEL OUT OF RANGE
C      15 - RECOVERY REMELT TANK LEVEL OUT OF RANGE
C
C      16 - SATURATOR FLOW CONTROLLER FEED-BACK SIGNAL OUT OF RANGE
C      17 - REMELT FLOW CONTROLLER FEED-BACK SIGNAL OUT OF RANGE
C      18 - MAGFLOW SIGNAL OUT OF RANGE
C      19 - REMELT RETURN FLOW OUT OF RANGE
C      20 - CLOUDY LIQUOR RETURN FLOW OUT OF RANGE
C      21 - "A" SATURATOR GAS FLOW OUT OF RANGE
C      22 - "B" SATURATOR GAS FLOW OUT OF RANGE
C      23 - "C" SATURATOR GAS FLOW OUT OF RANGE
C      24 - LIME WHEEL SPEED ( FLOW ) OUT OF RANGE
C
C      25 - "A" SATURATOR PH OUT OF RANGE
C      26 - "B" SATURATOR PH OUT OF RANGE
C      27 - "C" SATURATOR PH OUT OF RANGE
C
C      28 - GAS CO2 CONCENTRATION LESS THAN  8%
C      29 - GAS CO2 CONCENTRATION LESS THAN 10%
C
C
C
C
C
C---------------------------------------------------------------------------
C
C                 ------ COMMON ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
```

PAGE 0002 ENGUN 8:27 AM SUN., 8 AUG., 1976

```
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C   ENG    - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C   ADCV   - A/D VOLTAGES (UPDATED BY SCAD)
C   CDACV  - D/A VOLTAGES (UPDATED BY CDAC)
C
C   SAFCOD- SATURATOR FLOW CONTROL DATA
C   CLFLOD- CLOUDY LIQUOR FLOW DATA
C   REMLTD- REMELT CONTROL DATA
C   CLIMED- CONTROL LIME DATA
C   GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C   GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C   GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C   FILCYD- FILTER CYCLE MONITER DATA
C   SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C   ISAMT  - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C   ISMUL  - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C   IRN    - RESOURCE NUMBERS
C   ICIN   - CONTACT STATUS IN (UPDATED BY SCCS)
C   ICOUT  - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C   ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C   ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C   ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C----------------------------------------------------------------------
C
C
      CALL WAIT(1,3,IERR)
C                       ONE MINUTE WAIT TO SUPPRESS ERROR MESSAGES AT
C                       START-UP DURING TERMINAL ENABLE.
  100 CALL RNRQ(2,IRN(6),IDUM)
C                         LOCK RESOURCE NUMBER
      CALL SWITF(12)
      CALL ENGUS(ADCV,ENG)
      DO 150 I = 11,20
        CALL RNRQ(4,IRN(I),IDUM)
C                         RELEASE OF RESOURCE NUMBERS
  150 CONTINUE
      GOTO 100
      END
```

FTN4 COMPILER: HP92060-16092 REV. 1726

** NO WARNINGS ** NO ERRORS ** PROGRAM = 00054 COMMON = 00758

PAGE 0004 FTN. 8:27 AM SUN., 8 AUG., 1976

```
      SUBROUTINE ENGUS(ADCV,ENG),030178BDR 160178BDR 230178BDR
C
C*************************************************************************
C     THIS SUBROUTINE CALCULATES THE ENGINEERING UNITS OF THE FACTORY
C     DATA STORED AS VOLTS IN THE ADCV ARRAY.
C
C*************************************************************************
C
      DIMENSION ADCV(64),ENG(64)
      IREP = 60
C
C         ****TEMPERATURES****
      DO 200 I=11,18
        ENG(I)=ADCV(I)*10.
        J=I-10
        IF((ADCV(I).LT.0.).OR.(ADCV(I).GT.10.))CALL ERMES(J,
     1    IFIX(100.*ADCV(I)),IREP)
C                  INSTRUMENT FAILURE CHECK
  200 CONTINUE
C          **** SPECIFIC TEMPERATURE RANGE CHECK-OUT ****
      IF((ENG(15).LT.60.).OR.(ENG(15).GT.90.))CALL ERMES(5,
     1  IFIX(ENG(15)),IREP)
      IF((ENG(15).LT.60.).OR.(ENG(15).GT.90.))ENG(15) = 80.
C        DEFAULT VALUE OF SATURATOR TEMP. FOR REPORTING
CC    IF((ENG(18).LT.75.).OR.(ENG(18).GT.90.))CALL ERMES(8,
CC   1  IFIX(ENG(18)),IREP)
C
C         **** SPECIFIC GRAVITY AT POLISHING BRIXER ****
      SGPB=1.23+0.013*ADCV(3)
      IF((SGPB.GT.1.2449).AND.(SGPB.LT.1.3889)) GOTO 300
C         MINIMUM SG SET AT  60 DEG. BRIX AND  90 DEG. CELSIUS.
C         MAXIMUM SG SET AT  80 DEG. BRIX AND  60 DEG. CELSIUS.
      CALL ERMES(9,IFIX(100.*SGPB),IREP)
      SGPB = 1.30
C         DEFAULT VALUE AT  68 DEG. BRIX AND  80 DEG. CELSIUS.
C
C        ****** TANK LEVELS *****
  300 IF((ADCV(1).LT.0.).OR.(ADCV(1).GT.10.))CALL ERMES(11,
     1  IFIX(100.*ADCV(1)),IREP)
      IF(ADCV(1).LT.0.)ADCV(1)=0.
      IF(ADCV(1).GT.10.)ADCV(1)=10.
      ENG(1)=0.3744*ADCV(1)/SGPB
      ENG(1)=100.*ENG(1)/2.261
C                      LEVEL AS % FULL
      IF((ADCV(2).LT.0.).OR.(ADCV(2).GT.10.))CALL ERMES(12,
     1  IFIX(100.*ADCV(2)),IREP)
      IF(ADCV(2).LT.0.)ADCV(2)=0.
      IF(ADCV(2).GT.10.)ADCV(2)=10.
      ENG(2)=0.5573*ADCV(2)/SGPB
      ENG(2)=100.*ENG(2)/3.353
C                      LEVEL AS % FULL
      IF((ADCV(5).LT.0.).OR.(ADCV(5).GT.10.))CALL ERMES(13,
     1  IFIX(100.*ADCV(5)),IREP)
      IF(ADCV(5).LT.0.)ADCV(5)=0.
      IF(ADCV(5).GT.10.)ADCV(5)=10.
```

PAGE 0005 ENGUS 8:27 AM SUN., 8 AUG., 1976

```
      ENG(5)=0.4645*ADCV(5)/SGPB
      ENG(5)=100.*ENG(5)/3.871
C                     LEVEL AS % FULL
CC    IF((ADCV(27).LT.2.).OR.(ADCV(27).GT.10.))CALL ERMES(14,
CC   1  IFIX(100.*ADCV(27)),IREP)
      IF(ADCV(27).LT.2.)ADCV(27)=2.
      IF(ADCV(27).GT.10.)ADCV(27)=10.
      ENG(27)=0.4047*(ADCV(27)-2.)/SGPB
      ENG(27)=100.*ENG(27)/2.165
C                     LEVEL AS % FULL
CC    IF((ADCV(28).LT.2.).OR.(ADCV(28).GT.10.))CALL ERMES(15,
CC   1  IFIX(100.*ADCV(28)),IREP)
      IF(ADCV(28).LT.2.)ADCV(28) = 2.0
      IF(ADCV(28).GT.10.)ADCV(28) = 10.
      ENG(28) = 0.2903*(ADCV(28)-2.)/1.276
      ENG(28)=100.*ENG(28)/1.82
C                     LEVEL AS % FULL
C
C             ****** BRIXES ****
      TEMP=ENG(15)
      ENG(3)=100.*(SGPB-1.1+.0022*TEMP)/(.2695*SGPB+.00229*TEMP)
C     THIS FORMULA IS INCORRECT.
      IF((ENG(3).LT.60.).OR.(ENG(3).GT.80.))CALL ERMES(9,
     1  IFIX(100.*ENG(3)),IREP)
      ENG(10) = 63. + 7./4.*(ADCV(10)-1.)
CC    IF((ENG(10).LT.63.).OR.(ENG(10).GT.70.))CALL ERMES(10,
CC   1  IFIX(100.*ENG(10)),IREP)
C
C     ******  130K FEED-BACK SIGNALS *****
      ENG(7)=(ADCV(7)-1.)/4.
      IF((ADCV(7).LT.1.).OR.(ADCV(7).GT.5.))CALL ERMES(16,
     1  IFIX(100.*ADCV(7)),IREP)
      ENG(8)=(ADCV(8)-1.)/4.
      IF((ADCV(8).LT.1.).OR.(ADCV(8).GT.5.))CALL ERMES(17,
     1  IFIX(100.*ADCV(8)),IREP)
C
C        ****** FLOW *****
      IF(ADCV(6).GT.0.)GOTO 301
      ENG(6) = 0.
      GOTO 302
  301 ENG(6)=11.76*SQRT(ADCV(6)/10.)
C                     REMELT FLOW RATE IN CU.M/HR
C     IF((ADCV(6).LT.2.).OR.(ADCV(6).GT.10.))CALL ERMES(19,
C    1  IFIX(100.*((ADCV(6)-2.)/8.)),IREP)
  302 ENG(9) = (ADCV(9)-0.836)/0.937
C                     LIME WHEEL SPEED  (0-10 RPM)
      IF ((ADCV(9).LT.1.).OR.(ADCV(9).GT.10.))CALL ERMES(24,
     1   IFIX(ENG(9)),IREP)
      DO 310 I=24,26
        A = 5436.6
C                     FLOW IN CU.M/HR FOR A&B SATS.
        IF(I.EQ.26)A=2718.3
C                     FLOW IN CU.M/HR FOR C SAT
        ARG = (ADCV(I)-2.)/8.
        IF(ARG.LE.0.)GOTO 305
```

PAGE 0006 ENGUS 8:27 AM SUN., 8 AUG., 1976

```
            ENG(I) = A*SQRT(ARG)
      305   IF(ADCV(I).LE.2.)CALL ERMES(I-3,IFIX(ADCV(I)),IREP)
            IF(ADCV(I).GE.10.)CALL ERMES(I-3,IFIX(ADCV(I)),IREP)
      310 CONTINUE
          ENG(23) = (ADCV(23)-2.)/8.*153.5
C                         MAGFLOW METER, CU.M./H
          IF((ADCV(23).LT.2.).OR.(ADCV(23).GT.10.))CALL ERMES(17,
         1  IFIX(100.*(ADCV(23)-2.)/8.),IREP)
C
C
C          ***** PH'S *****
C
C        PH SETPOINT(A&B)=9.2 : MAX=9.7 : MIN=9.0  AT 20 DEG. C
C        PH SETPOINT(C)  =8.2  : MAX=8.7 : MIN=8.0  AT 20 DEG. C
C        FACTORY VALUES =( LAB VALUES - 0.4 )   ASSUMED HERE.
C
          ENG(20)=7.+(ADCV(20)-2.)*.625
CC        IF((ENG(20).LT.8.6).OR.(ENG(20).GT. 9.3))CALL ERMES(25,
CC       1  IFIX(100.*ENG(20)),IREP)
          ENG(21)=7.+(ADCV(21)-2.)*.625
CC        IF((ENG(21).LT.8.6).OR.(ENG(21).GT.9.3))CALL ERMES(26,
CC       1  IFIX(100.*ENG(21)),IREP)
          ENG(22)=7.+(ADCV(22)-2.)*.625
CC        IF((ENG(22).LT.7.8).OR.(ENG(22).GT.8.6))CALL ERMES(27,
CC       1  IFIX(100.*ENG(22)),IREP)
C
C          ***** GAS CO2 CONCENTRATION *****
C
          ENG(30) = (ADCV(30)-2.)*2.5
          IF(ENG(30).LT.8.0)CALL ERMES(28,IFIX(100.*ENG(30)),IREP)
CC        IF(ENG(30).LT.10.)CALL ERMES(29,IFIX(100.*ENG(30)),IREP)
          RETURN
          END
```

FTN4 COMPILER: HP92060-16092 REV. 1726

** NO WARNINGS ** NO ERRORS ** PROGRAM = 01365 COMMON = 00000

PAGE 0001 FTN. 9:18 AM MON., 20 FEB., 1978

```
FTN4,L,T
      PROGRAM  WCHDG(2,20),091177BDR 170178BDR
C*************************************************************************
C     WCHDG - WATCH-DOG.
C
C
C     THIS PROGRAM CHECKS THE OPERATION OF ALL CONTROL PROGRAMS.
C     IF ANY OF THEM STOP RUNNING IT CAUSES THE CORRESPONDING
C     CONTROL LOOP TO BE SWITCHED TO MANUAL. IT ALSO CHECKS FOR
C     COMPUTER FAILURE AND USES THE WATCH-DOG TIMER. IF PACIR
C     STOPS RUNNING, ALL CONTROL LOOPS ARE SWITCHED TO MANUAL USING
C     THE MASTER SWITCH.
C
C     EACH BIT IN THE WORDS ISCOP1 AND ISCOP2 SIGNIFIES THE STATUS
C     OF A PROGRAM. WHEN A CONTROL PROGRAM RUNS IT SETS A BIT ALLO-
C     CATED TO IT, TO THE VALUE 1. WCHDG CHECKS TO SEE THAT THE BITS IN
C     ISCOP1 HAVE BEEN SET TO 1. IF SO ,IT SETS THEM BACK TO ZERO.
C     IF NOT, A COUNTER IS USED TO TIME OUT THAT PROGRAM BY COUNTING MAX
C     ERROR CONDITIONS. IF IT "TIMES OUT" WITHOUT BEING RESET TO 1, THE
C     CONTROL LOOP IS SWITCHED TO MANUAL , A MESSAGE IS SENT TO THE
C     OPERATOR AND THE CORRESPONDING BIT IN ISCOP2 IS SET TO ZERO.
C     (THIS IS USED AS A FLAG IN PROGRAM CLOOP)
C
C                   BIT           CONTROL PROGRAM NAME
C                   ---           --------------------
C                    1                  SAFCO
C                    2                  CLFLO
C                    3                  REMLT
C                    4                  CLIME
C                    5                  GASFA
C                    6                  GASFB
C                    7                  GASFC
C
C       MESSAGES:
C            -1 = NOT READING ZERO FROM LAM GRADER.
C            -2 = NOT READING 32767 FROM LAM GRADER.
C             3 = SAFCO HAS GONE OFF-LINE.
C             4 = SAFCO IS NOW ON-LINE.
C             5 = CLFLO HAS GONE OFF-LINE.
C             6 = CLFLO IS NOW ON-LINE.
C             7 = REMLT HAS GONE OFF-LINE
C             8 = REMLT IS NOW ON-LINE.
C-------------------------------------------------------------------------
C
C                 ------ COMMON  ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C    ENG   - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C    ADCV  - A/D VOLTAGES (UPDATED BY SCAD)
```

PAGE 0002 WCHDG 9:18 AM MON., 20 FEB., 1978

```
C   CDACV - D/A VOLTAGES (UPDATED BY CDAC)
C
C   SAFCOD- SATURATOR FLOW CONTROL DATA
C   CLFLOD- CLOUDY LIQUOR FLOW DATA
C   REMLTD- REMELT CONTROL DATA
C   CLIMED- CONTROL LIME DATA
C   GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C   GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C   GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C   FILCYD- FILTER CYCLE MONITER DATA
C   SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C   ISAMT - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C   ISMUL - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C   IRN   - RESOURCE NUMBERS
C   ICIN  - CONTACT STATUS IN (UPDATED BY SCCS)
C   ICOUT - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C   ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C   ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C   ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C-----------------------------------------------------------------------
C
C
      INTEGER MFLAG(16)
C
C-----------------------------------------------------------------------
C  1. INITILISATION.
C
      CALL DECLR(LMADR,1,23,0)
C
      LU = 1
      ISCOP(1)= 0
      ISCOP(2)= 0
      NLOOPS = 7
C                     NUMBER OF ACTIVE CONTROL LOOPS
      MAXNO = 2*(ISMUL(2)*ISMUL(6)/ISMUL(4))
C                     2 CYCLES OF THE CONTROL PROGRAMS RELATIVE TO WCHDG.
      DO 10 I=1,16
        MFLAG(I) = MAXNO
   10 CONTINUE
      CALL WAIT(1,3,IERR)
C                WAIT ONE MINUTE FOR CONTROL PROGRAMS TO
C                INITIALISE ISCOP.
C
C-----------------------------------------------------------------------
C  2. WAIT ON RESOURCE NUMBER 3, CLEARED BY PACIR.
C
  100 CALL RNRQ(2,IRN(3),ISTAT)
C
      CALL SWITF(11)
C
C-----------------------------------------------------------------------
C  3. TRIGGER WATCH-DOG TIMER BY WRITING TO AND READING FROM THE LAM
C     GRADER.
```

PAGE 0003 WCHDG 9:18 AM MON., 20 FEB., 1978

```
C
      ID = 000000B
      CALL CAMAC(16,LMADR,ID,IDUM)
C                        WRITE ZERO
      CALL CAMAC(0,LMADR,IDATA,IDUM)
C                      READ BACK
           IF(IDATA.NE.0)CALL MESAG(-1,0)
C
      ID = 177777B
      CALL CAMAC(16,LMADR,ID,IDUM)
C               WRITE 32767
      CALL CAMAC(0,LMADR,IDATA,IDUM)
C               READ BACK
        IF(IDATA.NE.177777B)CALL MESAG(-2,0)
C
C---------------------------------------------------------------------------
C  4. CHECK ON BITS SET BY THE CONTROL LOOPS.
C
      MAXNO = 2*(ISMUL(2)*ISMUL(6)/ISMUL(4))
      DO 200 J=1,NLOOPS
        I = JBIT(J,ISCOP(1))
C                     NOTE:-  I = 1 WHEN PROGRAM RUNNING
        IF(MFLAG(J).GT.100) MFLAG(J) = 100
C                     PROTECTION WHEN PROGRAMS NOT RUNNING.
        ICNT = MFLAG(J) - MAXNO
C
      IF(ICNT.LT.0)GOTO 120
      IF((ICNT.GT.0).AND.(I.EQ.0))GOTO 120
C
C -------------------------------------------------------------------------
C  5. OUTPUT MESSAGE TO TERMINAL.
C
      JMES=2*J + 1 + I
      K = J-1
      CALL MESAG(JMES,K)
C
C---------------------------------------------------------------------------
C  6. CHANGE THE CONTACT OUTPUT STATUS AND SET FLAG IN ISCOP(2).
C
        CALL WCOUT(K,I)
C                     I=1 CLOSE CONTACT, I=0 OPEN CONTACT
      CALL SETB(J,ISCOP(2),I)
C
C--------------------------------------------------------------------------
C  7. RESETABLE COUNT UP BEFORE SWITCH OVER TO MANUAL.
C
  120   MFLAG(J) = (MFLAG(J) + 1)*(1-I)
C                     INHIBIT COUNT UP IF PROGRAM IS RUNNING
  200 CONTINUE
C
C---------------------------------------------------------------------------
C  8. RESET CONTROL PROGRAM WORD "ISCOP".
C
      ISCOP(1)=0
      GOTO 100
```

```
PAGE 0004  WCHDG  9:18 AM  MON., 20  FEB., 1978

 0166        END

 FTN4 COMPILER: HP92060-16092 REV. 1726

 **  NO WARNINGS **  NO ERRORS **   PROGRAM = 00293       COMMON = 00758
```

PAGE 0001 FTN. 9:42 AM MON., 20 FEB., 1978

```
FTN4,L,T
      PROGRAM SERVO(2,30),191277BDR 050178BDR 310178BDR
C***********************************************************************
C     READS THE SERVO-BALANCE REGISTER AND STORES RAW FEED
C     STATISTICS.
C     DEFINITIONS:
C           TOLD,TNEW,DELT ARE IN HOURS.
C           TMASS(1,K)=MASS OF K-TH TIP AGO(TONNES).
C           TMASS(2,K)=HOURS SINCE K-TH TIP AGO OCCURRED.
C           PROD1=TONNES MELT ACCUMULATED VIA 1ST SERVO-BALANCE.
C           PROD2=TONNES MELT ACCUMULATED VIA 2ND SERVO-BALANCE.
C           PROD=TOTAL TONNES MELT FOR THIS PRODUCTION RUN.
C           SHIFT(I)=HOURLY AVERAGE MELT RATE(TONNES) FOR LAST SHIFT(I)T.
C           HOUR=TONNES MELT PER HOUR FOR LAST HOUR.
C           HOURLY=TONS PER HOUR ON-THE-HOUR.(AN ARRAY CONTAINING THE
C                 LAST 8 HOURS VALUES).
C
C     SERVOD(1) = PROD1 = CUMULATIVE TONS MELT ON SCALE 1.
C     SERVOD(2) = PROD2 = CUMULATIVE TONS MELT ON SCALE 2.
C     SERVOD(3) = PROD  = CUMULATIVE TOTAL TONS MELT.
C     SERVOD(4) = HOUR  = AVERAGE MELT RATE OVER THE IMMEDIATE PAST HOUR
C     SERVOD(5) = BLANK
C     SERVOD(6) = TMASS0= SCALE DUMP IN TONS.
C     SERVOD(7) = IMOT1 = NUMBER OF PULSES FROM SCALE 1.
C     SERVOD(8) = IMOT2 = NUMBER OF PULSES FROM SCALE 2.
C     SERVOD(9) = DELT  = TIME SINCE LAST DUMP (HOURS).
C     SERVOD(10)= SHIFT(1)= SHIFT THROUGHPUT RATE FOR 22H00-6H00.
C     SERVOD(11)= SHIFT(2)= SHIFT THROUGHPUT RATE FOR 6H00-14H00.
C     SERVOD(12)= SHIFT(3)= SHIFT THROUGHPUT RATE FOR 14H00-22H00.
C     SERVOD(13 TO 20)= HOURLY MELT RATES ON-THE-HOUR FOR THE LAST 8
C                       HOURS.(SERVOD(13)=MOST RECENT VALUE.)
C
C***********************************************************************
C
      REAL SERVO1,SERVO2,SERVO3,SERVO4,SERVO5
      INTEGER CHECK(3)
      DOUBLE PRECISION PROD1,PROD2,PROD,SHIF,HOURLY,HOUR
      DIMENSION TMASS(2,90),IT(5),IYEAR(1),SHIFT(3)
C
C                 ------ COMMON ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C    ENG    - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C    ADCV   - A/D VOLTAGES (UPDATED BY SCAD)
C    CDACV  - D/A VOLTAGES (UPDATED BY CDAC)
C
C    SAFCOD- SATURATOR FLOW CONTROL DATA
C    CLFLOD- CLOUDY LIQUOR FLOW DATA
C    REMLTD- REMELT CONTROL DATA
```

```
PAGE 0002  SERVO  9:42 AM  MON., 20  FEB., 1978

C    CLIMED- CONTROL LIME DATA
C    GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C    GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C    GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C    FILCYD- FILTER CYCLE MONITER DATA
C    SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C    ISAMT  - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C    ISMUL  - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C    IRN    - RESOURCE NUMBERS
C    ICIN   - CONTACT STATUS IN (UPDATED BY SCCS)
C    ICOUT  - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C    ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C    ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C    ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C-------------------------------------------------------------------------
C
C
      EQUIVALENCE(SERVOD(1),SERVO1)
      EQUIVALENCE(SERVOD(2),SERVO2)
      EQUIVALENCE(SERVOD(3),SERVO3)
      EXTERNAL IFBRK
      DATA CHECK/2HCH,2HEC,1HK/
C
C
C-------------------------------------------------------------------------
C 1. INITIALISATION.
C
      SHIF=0.
      IFLAG=0
      IFLAG2=0
      CALL DECLR(IREG0,1,16,0)
      CALL DECLR(IREG1,1,16,1)
      CALL EXEC(11,IT,IYEAR)
      TOLD=IT(4)+(IT(3)+IT(2)/60.)/60.
      TNEW =TOLD
      CALL CAMAC(9,IREG0,IDUM,IQ)
      CALL CAMAC(9,IREG1,IDUM,IQ)
C                        CLEAR REGISTERS.
      ICOUT(3) = ICIN(3)
      ICOUT(4) = ICIN(4)
C
C-------------------------------------------------------------------------
C 2. MAIN LOOP STARTS.
C
  100 CALL RNRQ(2,IRN(8),IDUM)
C                                      RESOURCE NUMBER CLEARED BY SCCS
      CALL SWITF(10)
C
C-------------------------------------------------------------------------
C 3. SENSE SWITCH STATUS.
C
      NUMB = ICIN(3)
      ISRVN1=IBIT(16,NUMB)
```

PAGE 0003 SERVO 9:42 AM MON., 20 FEB., 1978

```
      NUMB = ICIN(4)
      ISRVN2=IBIT(1,NUMB)
      NUMB = ICOUT(3)
      ISRVO1=IBIT(16,NUMB)
      NUMB = ICOUT(4)
      ISRVO2=IBIT(1,NUMB)
C
C-----------------------------------------------------------------------
C 4. TEST FOR CONTACT CLOSURE & IGNOR CONTACT OPENING.
C
      IF(ISRVO1-ISRVN1)200,150,150
  150 IF(ISRVO2-ISRVN2)300,900,900
C                      ACT IF ISRVN CHANGES FROM 0 TO 1
C
C-----------------------------------------------------------------------
C 5. READ AND CLEAR REGISTERS.
C
C              READ AND CLEAR REGISTER '0' .
C
  200 IF(IFBRK(IDUM))205,210
  205 WRITE(7,1000)
 1000 FORMAT("ENTER THE NEW SCALE READING.")
      READ(7,*)PROD1
  210 CALL WAIT(10,2,IERR)
      CALL CAMAC(2,IREG0,IMOT1,IQ)
      TMASS0= IMOT1/1000.
      PROD1=PROD1+TMASS0
      SERVO1 = PROD1
      GOTO 400
C
C              READ AND CLEAR REGISTER '1' .
C
  300 IF(IFBRK(IDUM))305,310
  305 WRITE(7,1000)
      READ(7,*)PROD2
  310 CALL WAIT(10,2,IERR)
      CALL CAMAC(2,IREG1,IMOT2,IQ)
      TMASS0= IMOT2/1000.
      PROD2=PROD2+TMASS0
      SERVO2 = PROD2
C
  400 PROD = SERVO3
      PROD = PROD + TMASS0
      SERVO3 = PROD
C
C  TEMPORARY USE FOR DEBUGGING:-
C
      SERVOD(6) = TMASS0
      SERVOD(7) = IMOT1
      SERVOD(8) = IMOT2
C     SERVOD(9) = DELT
C
C
C-----------------------------------------------------------------------
C 6. RECORD THE DUMP INTERVAL.
```

PAGE 0004 SERVO 9:42 AM MON., 20 FEB., 1978

```
C
C
      TOLD = TNEW
      CALL EXEC(11,IT,IYEAR)
      TNEW=IT(4)+(IT(3)+IT(2)/60.)/60.
      DELT=TNEW-TOLD
      IF(DELT.LT.0)DELT=DELT+24.
      SERVOD(9) = DELT
      CALL EXEC(24,CHECK)
C
C---------------------------------------------------------------------------
C 7. UPDATE THE TIP RECORD.
C
      DO 500 I=90,2,-1
        J= I-1
        TMASS(1,I)=TMASS(1,J)
        TMASS(2,I)=TMASS(2,J)+DELT
  500 CONTINUE
      TMASS(1,1)=TMASS0
      TMASS(2,1)=0.
C
C---------------------------------------------------------------------------
C 8. CHECK WHETHER A NEW SHIFT HAS COMMENCED.
C
      IF((TOLD.LT.6.).AND.(TNEW.GE.6.))IFLAG=1
      IF((TOLD.LT.14.).AND.(TNEW.GE.14.))IFLAG=2
      IF((TOLD.LT.22.).AND.(TNEW.GE.22.))IFLAG=3
      IF(IFLAG.LT.1)GOTO 600
C
C---------------------------------------------------------------------------
C 9. CALCULATE AND STORE MEAN TONNES/HOUR.
C
C 9.1 AVERAGE MELT RATE OVER LAST SHIFT OR PART THEREOF:-
C
      IF(SHFT.LE.0.)GOTO 600
      SHIFT(IFLAG) = SHIF/SHFT
      SERVOD(9+IFLAG) = SHIFT(IFLAG)
      SHIF = 0.
      SHFT = 0.
C
C 9.2 UPDATING OF HOURLY MELT RATES OVER THE LAST 8 HOURS, EVERY HOUR
C     ON-THE-HOUR:
C
  600 IFLAG = 0
      DO 610 I=1,24
        IF((TOLD.LT.I).AND.(TNEW.GE.I))IFLAG2 = 1
        IF((TOLD.GT.23).AND.(TNEW.LT.1))IFLAG2=1
  610 CONTINUE
      IF(IFLAG2.NE.1)GOTO 630
      DO 620 K=20,14,-1
        J = K-1
        SERVOD(K) = SERVOD(J)
  620 CONTINUE
      SERVO5 = HOURLY
      SERVOD(13) = SERVO5
```

```
PAGE 0005  SERVO  9:42 AM  MON., 20  FEB., 1978

          HOURLY = 0.
          IF(IDUMY(49).NE.1)GOTO 630
          WRITE(6,4000)IT(5),IT(4),IT(3),SERVOD(13),SAFCOD(20)
     4000 FORMAT("DAY ",I3,I5,"H",I2,5X,"MELT RATE=",F8.3,4X,
       1  ": SAT. SOLIDS=",F8.3,2X,"TPH",/)
C
C 9.3 HOURLY MELT RATE OVER THE IMMEDIATE PAST HOUR:-
C
  630 HOUR=0.
      IFLAG2 = 0
      SHIF = SHIF + TMASS0
      SHFT = SHFT + DELT
      HOURLY = HOURLY + TMASS0
      DO 700 K=1,90
        IF(TMASS(2,K).GT.1.)GOTO 800
        HOUR=HOUR+TMASS(1,K)
        IF(K.EQ.90)HOUR=HOUR/TMASS(2,90)
  700 CONTINUE
  800 SERV04=HOUR
      SERVOD(4) = SERV04
C                       CURRENT RATE OVER IMMEDIATE PAST HOUR.
C
C------------------------------------------------------------------------
C 10. UPDATE WORDS FOR OLD CONTACT STATUS.
C
  900 NUMB =ICOUT(3)
      CALL SETB(16,NUMB,ISRVN1)
      ICOUT(3) = NUMB
      NUMB = ICOUT(4)
      CALL SETB(1,NUMB,ISRVN2)
      ICOUT(4) = NUMB
      GOTO 100
      END


FTN4 COMPILER: HP92060-16092 REV. 1726

**  NO WARNINGS **  NO ERRORS **   PROGRAM = 01380      COMMON = 00758
```

PAGE 0001 FTN. 9:30 AM MON., 20 FEB., 1978

```
FTN4,L,T
      PROGRAM SAFCO(2,40),081277? 180178BDR 230178BDR
C***********************************************************************
C     SAFCO - "SATURATOR FLOW CONTROL".
C
C     SAFCO ADJUSTS THE SATURATOR FLOW SET-POINT IN ACCORDANCE
C     WITH THE AFST & SST LEVELS AND THEIR DERIVATIVES. PROPORTIONAL
C     PLUS INTEGRAL CONTROL IS USED. THE AFST LEVEL MEASUREMENT
C     IS PASSED THROUGH A SECOND-ORDER LOW-PASS FILTER TO
C     PREDICT THE TREND WHILE FILTERING OUT THE TRANSIENTS. THE AFST
C     AND SST LEVELS ARE NORMALISED BY DIVIDING BY THEIR
C     MAXIMUMS. THE REQUIRED FLOW CHANGE IS CALCULATED AND CON-
C     VERTED INTO A NUMBER OF PULSES WHICH ARE GENERATED BY THE
C     CAMAC PULSER MODULE. VALVE POSITION AND TANK LEVEL LIMITS ARE
C     CHECKED. THE PROGRAM ONLY EXECUTES WHEN ITS RESOURCE NUMBER IS
C     CALLED.
C
C
C              HAFST - AFST LEVEL
C              HSST - SST LEVEL
C              HCLT - CLT LEVEL
C              ___M - MAXIMUM LEVEL
C              ___N - NORMALISED LEVEL
C              __DOT - DERIVATIVE
C              ___SP - LEVEL SET-POINT
C              ___F - FILTERED VALUE
C
C     TANK LEVELS ARE MEASURED IN METERS.FLOW IS IN CUBIC METERS/HR.
C           ALARM MESSAGES :
C              1=SST EMPTY
C              2=SST FULL
C              3=AFST EMPTY
C              4=AFST FULL
C              5=SAT. SUPPLY CONTROL VALVE CLOSED
C              6=SAT. SUPPLY CONTROL VALVE FULL OPEN
C              7=CALCULATED SAT. VALVE POSN. DIFFERS FROM TRUE VALUE
C              8=CHANGE IN SAT. SUPPLY VALVE POSN. > 10%
C              9=CHECK THE VALUES OF THE ERROR DERIVATIVES
C                     (I.E. SAFCOD(12) & (13))
C
C
C
C-----------------------------------------------------------------------
C
C                 ------ COMMON ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C   ENG   - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C   ADCV  - A/D VOLTAGES (UPDATED BY SCAD)
```

PAGE 0002 SAFCO 9:30 AM MON., 20 FEB., 1978

```
C    CDACV - D/A VOLTAGES (UPDATED BY CDAC)
C
C    SAFCOD- SATURATOR FLOW CONTROL DATA
C    CLFLOD- CLOUDY LIQUOR FLOW DATA
C    REMLTD- REMELT CONTROL DATA
C    CLIMED- CONTROL LIME DATA
C    GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C    GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C    GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C    FILCYD- FILTER CYCLE MONITER DATA
C    SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C    ISAMT - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C    ISMUL - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C    IRN   - RESOURCE NUMBERS
C    ICIN  - CONTACT STATUS IN (UPDATED BY SCCS)
C    ICOUT - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C    ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C    ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C    ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C-----------------------------------------------------------------------
C
C
C
        EQUIVALENCE (SAFCOD(1),GPA)
C                           GAIN PROPORTIONAL, AFST
        EQUIVALENCE (SAFCOD(2),GPS)
C                           GAIN PROPORTIONAL, SSTL
        EQUIVALENCE (SAFCOD(3),GIA)
C                           INTEGRAL GAIN, AFST
        EQUIVALENCE (SAFCOD(4),GIS)
C                           INTEGRAL GAIN, SSTL
        EQUIVALENCE (SAFCOD(5),W)
C                           CUT OFF FREQUENCY, SECS
        EQUIVALENCE (SAFCOD(6),D)
C                           DAMPING FACTOR
        EQUIVALENCE (SAFCOD(7),HSSSP)
C                           STT LEVEL SET POINT (NORMALISED)
        EQUIVALENCE (SAFCOD(8),HAFSP)
C                           AFST LEVEL SET POINT (NORMALISED)
C        SAFCOD(9) = NUMP,        THE NUMBER OF PULSES OUTPUT
        EQUIVALENCE(SAFCOD(10),HAFF)
C                           FILTERED AFST LEVEL(NORMALISED)
        EQUIVALENCE (SAFCOD(18),ALPHA)
C                           EXPONENTIAL SMOOTHING FOR CUMULATIVE SOLIDS FLOW.
        EQUIVALENCE (SAFCOD(19),RATES)
C                           INSTANTANEOUS SOLIDS FLOW RATE.
        EQUIVALENCE (SAFCOD(20),SOLIDS)
C                           CUMULATIVE SOLIDS FLOW .
C
C      ****DECLARATION STATEMENT *****
        CALL DECLR(IPUL,1,14,0)
C
C
```

PAGE 0003 SAFCO 9:30 AM MON., 20 FEB., 1978

```
C **** SPECIFICATION OF CONSTANT DATA FOR BOTH CONTROL LOOPS.****
C
C                          MAXIMUM FLOW RATES (CU.METERS/HOUR)
      FMAF=153.5
C
C                          MAXIMUM LIQUID LEVELS (M)
      HAFM=3.353
      HSSM=2.261
      HPLM=3.871
C
C                          TANK CROSS-SECTIONAL AREAS (SQ.M.)
      AAFST=6.59
      ASST=11.4
C
C                       TANK VOLUMES (CU.M.)
      VAFST=AAFST*HAFM
      VSST=ASST*HSSM
C
C DEFAULT SET POINTS AND CONTROL GAINS
      HAFSP=0.5
      HSSSP=0.3
      VPLR=5.0
C
      GPA=0.2
C                       AFST PROPORTIONAL GAIN
      GIA=36.
C                       AFST INTEGRAL RESET TIME, MINUTES
      GPS=2.
C                       SST PROPORTIONAL GAIN
      GIS=50.
C                       SST INTEGRAL RESET TIME, MINUTES
      W=0.0015
C                       CUT OFF FREQUENCY, RADIANS/SEC
      D=0.7
C                       DAMPING FACTOR
C
C INITIAL CONDITIONS FOR THE PREDICTOR AND DIFFERENTIAL EQUATIONS.
C
      HSSN1=ENG(1)/100.
      HAFF1=ENG(2)/100.
      HAFF2=HAFF1
      RAF=HAFF1
C
      DLSSV=0.
C                          "CHANGE IN SAT. SUPPLY VALVE POSN."
C
      NUMPT=IFIX(1000.*ENG(7))
C                       INIT. VALUE OF TOTAL NO OF PULSES=VALVE POS*1000
C
      ALPHA = 0.2
C                       EXPONENTIAL SMOOTHING FOR SOLIDS FLOW CALCULATION.
C
C*************************************************************************
C
C                 ****MAIN LOOP FOR SAT. FEED CONTROL STARTS HERE ******
```

```
C
C          1. CALCULATE FILTER AND CONTROL CONSTANTS
C          2. READ NORMALISED LEVELS AND CHECK LIMITS
C          3. ONE-STEP-AHEAD PREDICTION OF MEAN AFST LEVEL
C          4. CALCULATE FLOW CHANGE
C          5. CHECK PULSER AND 130K OPERATION
C          6. WRITE TO PULSER
C
100   CALL RNRQ(2,IRN(11),IDUM)
C                    LOCK ON RESOURCE NUMBER UNTIL CLEARED BY ENGUN
C
      CALL SWITF(9)
C
      MASK=IAND(ICIN(4),4B)
      IF(MASK.NE.4B)GOTO 700
C
C    **** ERROR MESSAGE SUPPRESSION PERIOD(MINUTES) ****
      IREP = 60
C
      T1=FLOAT(ISAMT*ISMUL(2)*ISMUL(6))
C
C          ***** 1. CALC. FILTER CONSTANTS *****
C
      W0=SQRT(1.-D*D)*W
      A=W*D
      THETA=1.57
      IF(W0.GE.0.0001) THETA=ATAN(-A/W0)
      EAT=EXP(-A*T1)
      IF(THETA.EQ.1.57) CA=EAT
      IF(THETA.NE.1.57) CA=EAT*COS(W0*T1+THETA)/COS(THETA)
      CB=2.*EAT*COS(W0*T1)
      CC=EAT*EAT
      CD=1.+CA-CB
      CE=CC-CA
C
C CONTROL LOOP VOLUME GAINS
      GIAV=1./(60.*GIA)
      GISV=1./(60.*GIS)
C
C          ***** 2. READ NORMALISED LEVELS & CHECK LIMITS *****
C
      HAFN=ENG(2)/100.
      HSSN=ENG(1)*ALPHA/100. + (1.-ALPHA)*HSSN1
      HPLN=ENG(5)/100.
      IF(HSSN.LT..05) CALL ERMES(1,IFIX(100.*HSSN),IREP)
      IF(HSSN.GT..95) CALL ERMES(2,IFIX(100.*HSSN),IREP)
      IF(HAFN.LT..05) CALL ERMES(3,IFIX(100.*HAFN),IREP)
      IF(HAFN.GT..95) CALL ERMES(4,IFIX(100.*HAFN),IREP)
C
C   ***** 3.  ONE-STEP-AHEAD PREDICTION OF MEAN AFST LEVEL*****
C
      HAFF=CB*HAFF1-CC*HAFF2+CD*HAFN+CE*RAF
C                 CALCULATE DERIVATIVES AND ERRORS.
      HFDOT=(HAFF-HAFF1)/T1
      DELAF = HAFF-HAFF1
```

```
PAGE 0005  SAFCO  9:30 AM  MON., 20  FEB., 1978

           IF(ABS(DELAF).LT.0.1)GOTO 110
           CALL ERMES(9,IFIX(100.*HFDOT),IREP)
           HFDOT = SIGN(0.1,DELAF)/T1
       110 HSDOT=(HSSN-HSSN1)/T1
           DELSN = HSSN-HSSN1
           IF(ABS(DELSN).LT.0.1)GOTO 120
           CALL ERMES(9,IFIX(100.*HSDOT),IREP)
           HSDOT = SIGN(0.1,DELSN)/T1
       120 EAFT=HAFF-HAFSP
           ESST=HSSN-HSSSP
     C
     C     UPDATE PAST VALUES
     C
           HAFF2=HAFF1
           HAFF1=HAFF
           RAF=HAFN
           HSSN1=HSSN
     C
     C           ***** 4. CALC. FLOW CHANGE *****
     C
           GPISST=GPS*(HSDOT + GISV*ESST)
     C                        SST CONTRIBUTION
     C
           GAIN=0.
           IF((HPLN.LT.0.5).AND.(HAFN.LT.0.5))GAIN=.001
           GPIAST= -GPA*(HFDOT +GIAV*EAFT-GAIN*(0.5-HPLN))
     C                        AFST CONTRIBUTION
           IF(HSSN.GT.HSSSP) DLSF=GPIAST
     C                        IF SST IS ABOVE SP, CONTROL ON AFST ONLY
           IF((HSSN.LT.HSSSP).AND.(GPIAST.GT.0))DLSF=GPISST
     C                        IF SST LOW (BELOW SP) AND AFST TREND IS DOWN
     C                        CONTROL ON SST ONLY (THIS MAY BE SHUT DOWN)
           IF((HSSN.LT.HSSSP).AND.(GPIAST.LT.0))DLSF=GPISST+GPIAST
     C                        IF SST LOW AND AFST TREND IS UP
     C                        CONTROL ON BOTH SST AND AFST
     C                        (SHUT DOWN AND FILTER HOLD UP)
           DELN=DLSF*T1
           DLSSV=DELN+DLSSV
     C                        PICK UP ROUND OFF FROM LAST OUTPUT
     C
           IF(ABS(DLSSV).GT.0.001) GOTO 300
           NUMP=0
           GOTO 500
       300      NUMP=IFIX(DLSSV*1000.)
                DLSSV=AMOD(DLSSV,0.001)
     C                        SAVE ROUND OFF OF LESS THAN ONE PULSE
     C
     C            ***** 5. CHECK PULSER AND 130K OPERATION *****
     C
     C        1.IF CURRENT POSITION OF SET POINT IS NOT EQUAL TO COMPUTED
     C          POSITION, MESSAGE 7 OUT AND RESET NUMPT
     C        2.IF NEXT COMMAND WILL DRIVE SET POINT UNDER OR OVER RANGE,
     C          INHIBIT OUTPUT AND WRITE MESSAGE
     C        3. LIMIT CHANGE TO 10%.
     500   NPOS=ENG(7)*1000.
```

```
PAGE 0006  SAFCO  9:30 AM  MON., 20  FEB., 1978

           IDIF=NPOS-NUMPT
           IF(IABS(IDIF).LT.25) GOTO 600
               CALL ERMES(7,IDIF,IREP)
               NUMPT=NPOS
      C
        600 IF(IABS(NUMP).LT.100)GOTO 610
                 CALL ERMES(8,NUMP,IREP)
                 NUMP=100
      C
      610  IF((NUMP+NUMPT).GT.0) GOTO 620
               CALL ERMES(5,NUMP,IREP)
               NUMP=-NUMPT
      C
      620  IF((NUMP+NUMPT).LT.1000) GOTO 630
               CALL ERMES(6,NUMP,IREP)
               NUMP=1000-NUMPT
      C
      C
      C          ***** 6. WRITE TO PULSER *****
      C
      630  NUMPT=NUMPT+NUMP
      C                  INCREMENT TOTAL NO OF PULSES
           IF(NUMP.NE.0) CALL CAMAC(16,IPUL,NUMP,IQ)
      C                  WRITE PULSE COUNT IF NOT ZERO
           SAFCOD(9) = NUMP
      C
      C
      C    **** CUMULATIVE SOLIDS FLOW CALCULATION ****
      C
           FLOW = FLOW*(1-ALPHA) + ALPHA*ENG(23)
           BRIX =BRIX*(1-ALPHA) + ALPHA*ADCV(3)
           SGPB = 1.23 + 0.013*BRIX
           BRIX2 = BRIX2*(1-ALPHA) + ALPHA*ENG(3)
           RATES = FLOW*SGPB*BRIX2/100.
           DSOLID = RATES*ISAMT*ISMUL(2)*ISMUL(6)/3600.
           SOLIDS =SOLIDS + DSOLID
      C
      C
      C  UPDATE CONTROL WORD FOR AUTO/MANUAL WATCHDOG (PROGRAM: WCHDG)
      C
        700 MASK = ISHFT(1,0)
           ISCOP(1) = IOR(MASK,ISCOP(1))
      C
      C
           GOTO 100
      C
           END

FTN4 COMPILER: HP92060-16092 REV. 1726

**  NO WARNINGS **  NO ERRORS **   PROGRAM = 01127      COMMON = 00758
```

```
PAGE 0001  FTN.   9:32 AM  MON., 20  FEB., 1978

0001  FTN4,L,T
0002        PROGRAM CLFLO(2,40),040777? 230178BDR
0003  C*********************************************************************
0004  C     CLFLO - "CLOUDY-LIQUOR FLOW CONTROL".
0005  C
0006  C     CLFLO ADJUSTS THE ABSOLUTE VALVE POSITION IN THE CLOUDY-
0007  C     LIQUOR RETURNS LINE USING PROPORTIONAL-PLUS-INTEGRAL CONTROL
0008  C     ACTING ON THE NORMALISED ERROR IN THE TANK LIQUOR LEVEL.
0009  C     LIMITS ON THE VALVE POSITION AND TANK LEVEL ARE CHECKED
0010  C     AND MESSAGES SENT TO THE OPERATOR'S CONSOLE IF NECESARY.
0011  C     THE CLT LIQUOR LEVEL IS PASSED THROUGH A MATHEMATICAL FILTER
0012  C     AS FOR SAFCO.
0013  C     THE PROGRAM ONLY EXECUTES WHEN ENGUN RELEASES ITS RESOURCE NUMBER.
0014  C
0015  C     THE CONTROL ACTION CAN BE MADE TO ACT ON THE DIFFERENCE BETWEEN
0016  C     THE AFST AND CLT LEVELS BY DELETING LINES 165 AND 166.
0017  C
0018  C              HAFST - AFST LEVEL
0019  C              HCLT - CLT LEVEL
0020  C              ___M - MAXIMUM LEVEL
0021  C              ___N - NORMALISED LEVEL
0022  C              __DOT - DERIVATIVE
0023  C              ___SP - LEVEL SET-POINT
0024  C              ___F - FILTERED VALUE
0025  C
0026  C     TANK LEVELS ARE MEASURED IN METERS.FLOW IS IN CUBIC METERS/HR.
0027  C           ALARM MESSAGES :
0028  C              1=CLT EMPTY
0029  C              2=CLT FULL
0030  C              3=LIQUOR RETURNS VALVE POSN. CHANGE > 10%
0031  C              4=LIQUOR RETURNS VALVE CLOSED
0032  C              5=LIQUOR RETURNS VALVE FULL OPEN
0033  C              6=CHECK THE VALUE OF THE DERIVATIVE OF THE CLT LEVEL
0034  C                            (I.E. CLFLOD(7))
0035  C
0036  C---------------------------------------------------------------------
0037  C
0038  C                 ------ COMMON  ------
0039  C
0040        COMMON ENG(64),ADCV(64),CDACV(24),
0041       1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
0042       2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
0043       3   SERVOD(20),DUMMY(50),
0044       4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
0045       5   ISCOP(3),IDUMY(50)
0046  C
0047  C   ENG   - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
0048  C   ADCV  - A/D VOLTAGES (UPDATED BY SCAD)
0049  C   CDACV - D/A VOLTAGES (UPDATED BY CDAC)
0050  C
0051  C   SAFCOD- SATURATOR FLOW CONTROL DATA
0052  C   CLFLOD- CLOUDY LIQUOR FLOW DATA
0053  C   REMLTD- REMELT CONTROL DATA
0054  C   CLIMED- CONTROL LIME DATA
0055  C   GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
```

```
PAGE 0002  CLFLO  9:32 AM  MON., 20  FEB., 1978

C   GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C   GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C   FILCYD- FILTER CYCLE MONITER DATA
C   SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C   ISAMT  - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C   ISMUL  - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C   IRN    - RESOURCE NUMBERS
C   ICIN   - CONTACT STATUS IN (UPDATED BY SCCS)
C   ICOUT  - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C   ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C   ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C   ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C---------------------------------------------------------------------------------
C
C
C
        EQUIVALENCE(CLFLOD(1),GPC)
C                           CLT PROPORTIONAL GAIN
        EQUIVALENCE(CLFLOD(2),GIC)
C                           CLT INTEGRAL GAIN
        EQUIVALENCE(CLFLOD(3),VPLR)
C                           LIQUOR RETURNS VALVE POSN.
        EQUIVALENCE(CLFLOD(4),HCLF)
C                            FILTERED CLT LEVEL
C
        EQUIVALENCE(CLFLOD(5),W)
C                       CUT-OFF FREQUENCY
        EQUIVALENCE(CLFLOD(6),D)
C                           DAMPING FACTOR
        EQUIVALENCE(CLFLOD(7),HCDOT)
C                           RATE OF CHANGE OF FILTERED CLT LEVEL.
C
C
C **** SPECIFICATION OF CONSTANT DATA *****
C
C                              MAXIMUM FLOW RATE (CU.METERS/HOUR)
        FMCL=10.
C
C                              MAXIMUM LIQUID LEVEL (M)
        HAFM=3.353
        HCLM=2.165
C
C                              TANK CROSS-SECTIONAL AREA (SQ.M.)
        ACLT=4.67
C
C                           TANK VOLUME (CU.M.)
        VCLT=ACLT*HCLM
C
C DEFAULT SET POINTS AND CONTROL GAINS
        VPLR=5.0
C
        GPC=2.0
C                           CLT PROPORTIONAL GAIN
```

PAGE 0003 CLFLO 9:32 AM MON., 20 FEB., 1978

```
           GIC= 60.0
      C                        CLT INTEGRAL RESET TIME , MINS.
           W=0.0015
      C                        CUT OFF FREQUENCY, RADIANS/SEC
           D=0.7
      C                        DAMPING FACTOR
      C
      C INITIAL CONDITIONS FOR THE PREDICTOR AND DIFFERENTIAL EQUATIONS.
      C
           HAFF1=ENG(2)/100.
           HAFF2=HAFF1
           RAF=HAFF1
           HCLF1=ENG(27)/100.
           HCLF2=HCLF1
           RCL=HCLF1
      C
           DLLRV=0.
      C                           "CHANGE IN LIQUOR RETURNS VALVE POSN."
      C
      C
      100  CALL RNRQ(2,IRN(12),IDUM)
      C                        LOCK ON RESOURCE NUMBER UNTIL CLEARED BY ENGUN
      C
           CALL SWITF(8)
      C
           MASK=IAND(ICIN(4),10B)
           IF(MASK.NE.10B)GOTO 720
      C
      C   **** ERROR MESSAGE SUPPRESSION PERIOD(MINUTES) ****
           IREP = 60
      C
           T1=FLOAT(ISAMT*ISMUL(2)*ISMUL(6))
      C
      C         ***** 1. CALC. FILTER CONSTANTS *****
      C
           W0=SQRT(1.-D*D)*W
           A=W*D
           THETA=1.57
           IF(W0.GE.0.0001) THETA=ATAN(-A/W0)
           EAT=EXP(-A*T1)
           IF(THETA.EQ.1.57) CA=EAT
           IF(THETA.NE.1.57) CA=EAT*COS(W0*T1+THETA)/COS(THETA)
           CB=2.*EAT*COS(W0*T1)
           CC=EAT*EAT
           CD=1.+CA-CB
           CE=CC-CA
      C
      C    ****DETERMINE FILTERED AFST LEVEL *****
      C
           HAFN=ENG(2)/100.
      C
      C  *****   ONE-STEP-AHEAD PREDICTION OF MEAN AFST LEVEL*****
      C
           HAFF=CB*HAFF1-CC*HAFF2+CD*HAFN+CE*RAF
      C                  CALCULATE DERIVATIVES AND ERRORS.
```

```
PAGE 0004  CLFLO  9:32 AM  MON., 20  FEB., 1978

      HFDOT=(HAFF-HAFF1)/T1
C
C     UPDATE PAST VALUES
C
      HAFF2=HAFF1
      HAFF1=HAFF
      RAF=HAFN
C
C
C
C
C**********************************************************************
C
C   ******MAIN LOOP FOR CLOUDY-LIQUOR RETURNS RATE STARTS HERE***
C
C          1.   READ NORMALISED CLT LEVEL & CHECK LIMITS
C          2.   ONE-STEP-AHEAD PREDICTION OF MEAN CLT LEVEL
C                  USING SAME COEFFS. AS FOR AFST.
C          3.   CALC. FLOW CHANGE & CHECK LIMITS
C          4.   OUTPUT TO CONTROL DAC.
C
C     CONTROL LOOP VOLUME GAINS
      GICV=1./(60.*GIC)
C
C ****     1.  READ NORMALISED CLT LEVEL AND CHECK LIMITS. ****
C
      HCLN=ENG(27)/100.
      IF(HCLN.LT..05) CALL ERMES(1,IFIX(100.*HCLN),IREP)
      IF(HCLN.GT..95) CALL ERMES(2,IFIX(100.*HCLN),IREP)
C
C  **** 2.   ONE-STEP-AHEAD PREDICTION OF CLT MEAN LEVEL.*****
C       (USES SAME COEFFICIENTS AS FOR AFST LEVEL)
C
      HCLF=CB*HCLF1-CC*HCLF2+CD*HCLN+CE*RCL
      HCDOT=(HCLF-HCLF1)/T1
      DELCF = HCLF-HCLF1
      IF(ABS(DELCF).LT.0.1)GOTO 110
      CALL ERMES(6,IFIX(100.*HCDOT),IREP)
      HCDOT = SIGN(0.1,DELCF)/T1
C
C      UPDATE PAST VALUES.
C
  110 HCLF2=HCLF1
      HCLF1=HCLF
      RCL=HCLN
C
C    ****** 3.  CALC. FLOW CHANGE & CHECK LIMITS ****
C
      HFDOT=0.
      HAFF=0.3
C                  MAY BE DELETED IF DESIRED.
      DLLR=GPC*((HCDOT-HFDOT)+GICV*(HCLF-HAFF))
      DLLRV=DLLR*10.
      VPLR=VPLR+DLLRV*T1
C                         LIQ. RET. VALVE POSN.(0 TO 10 VOLTS)
```

```
PAGE 0005  CLFLO  9:32 AM  MON., 20  FEB., 1978

C
C         ****CHECK LIMITS *****
C                MAX. CHANGE = 10%
C                0<VPLR<10.
C
      IF(ABS(DLLRV).LT.1.)GOTO 700
        DLLRV=1.
        CALL ERMES(3,IFIX(100.*DLLRV),IREP)
  700 IF(VPLR.GT.0.)GOTO 710
      VPLR=0.1
      CALL ERMES(4,IFIX(100.*VPLR),IREP)
  710 IF(VPLR.LT.10.)GOTO 720
      VPLR=9.9
      CALL ERMES(5,IFIX(100.*VPLR),IREP)
C
C *****   OUTPUT TO CONTROL DAC. *****
C
 720  CALL CDAC(0,VPLR)
C
C   ****UPDATE CONTROL WORD FOR AUTO/MANUAL WATCHDOG (PROGRAM: WCHDG)***
C
      MASK = ISHFT(1,1)
      ISCOP(1) = IOR(MASK,ISCOP(1))
C
C   ****LOCK PROGRAM OUT UNTIL RELEASED AGAIN***
C
      GOTO 100
C
      END


FTN4 COMPILER: HP92060-16092 REV. 1726


**  NO WARNINGS **  NO ERRORS **   PROGRAM = 00712      COMMON = 00758
```

PAGE 0001 FTN. 9:34 AM MON., 20 FEB., 1978

```
FTN4,L,T
      PROGRAM REMLT(2,30),081277BDR 230178BDR 010278BDR
C************************************************************************
C
C       REMLT = RECOVERY REMELT RETURN FLOW CONTROL.
C
C       RECOVERY REMELT TANK LEVEL IS USED TO CONTROL THE RETURN FLOW.
C       THE FLOW CONTROLLER SETPOINT IS AJUSTED BY PULSE TRAIN USING
C       A TWO-TERM PROPORTIONAL PLUS INTEGRAL CONTROL ACTION.
C
C               GPR = REMELT PROPORTIONAL GAIN
C               GIR = REMELT INTEGRAL RESET TIME, MINUTES
C
C       ALARM MESSAGES:-
C               1 = REMELT TANK FULL  ( HRMN > 0.95 )
C               2 = REMELT TANK EMPTY ( HRMN < 0.05 )
C               3 = REMELT CALCULATED FLOW SETPOINT AND FEEDBACK DIFFER.
C               4 = REMELT CALCULATED FLOW SETPOINT CHANGE > 10%.
C               5 = REMELT VALVE CLOSED     (  0% OPEN )
C               6 = REMELT VALVE FULLY OPEN (100% OPEN)
C
C-----------------------------------------------------------------------
C
C                     ------ COMMON ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C   ENG   - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C   ADCV  - A/D VOLTAGES (UPDATED BY SCAD)
C   CDACV - D/A VOLTAGES (UPDATED BY CDAC)
C
C   SAFCOD- SATURATOR FLOW CONTROL DATA
C   CLFLOD- CLOUDY LIQUOR FLOW DATA
C   REMLTD- REMELT CONTROL DATA
C   CLIMED- CONTROL LIME DATA
C   GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C   GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C   GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C   FILCYD- FILTER CYCLE MONITER DATA
C   SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C   ISAMT - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C   ISMUL - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C   IRN   - RESOURCE NUMBERS
C   ICIN  - CONTACT STATUS IN (UPDATED BY SCCS)
C   ICOUT - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C   ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C   ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C   ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
```

PAGE 0002 REMLT 9:34 AM MON., 20 FEB., 1978

```
C-----------------------------------------------------------------------
C
C
C
      EQUIVALENCE (REMLTD(1),GPR)
      EQUIVALENCE (REMLTD(2),GIR)
      EQUIVALENCE (REMLTD(3),ALPHA)
      EQUIVALENCE (REMLTD(4),NUMP)
      EQUIVALENCE (REMLTD(5),HRMNSP)
C
C-----------------------------------------------------------------------
C 1. INITIALISATION  VALUES
C
C    **** DECLARATION STATEMENT ****
      CALL DECLR(IPUL,1,14,1)
C
      HRMM = 1.82
C                                MAXIMUM TANK LEVEL, METERS
      AREA = 10.03
C                                CROSS-SECTIONAL TANK AREA, SQ. METERS
      HRMNSP = 0.25
C                                DEFAULT NORMALISED LEVEL SET-POINT
      DELN = 0.
C                                INITIALISED ROUND-OFF VALUE.
      ALPHA = 0.2
C                               EXPONENTIAL SMOOTHING FACTOR.
      GPR = 1.
C                                PROPORTIONAL GAIN.
      GIR = 50.
C                                INTEGRAL RESET TIME, MINUTES
      NUMPT = IFIX(1000.*ENG(8))
C                                FEEDBACK SIGNAL.
C
C-----------------------------------------------------------------------
C 2. MAIN CONTROL LOOP STARTS
C
  100 CALL RNRQ(2,IRN(13),IDUM)
C                        LOCKS ON RESOURCE NUMBER UNTIL RELEASED BY ENGUN
      CALL SWITF(7)
      IREP = 60
C
      MASK=IAND(ICIN(4),20B)
      IF(MASK.EQ.0)GOTO 200
C                      AUTO/MANUAL SWITCH STATUS CHECK
C
C-----------------------------------------------------------------------
C 3. CALCULATE CONTROL CYCLE INTERVAL.
C
      DELT = FLOAT(ISAMT*ISMUL(2)*ISMUL(6))
C
C-----------------------------------------------------------------------
C 4. CONVERT AND CHECK INPUT DATA.
C
      GIRV = 1./(60.*GIR)
      HRMN = ENG(28)/100.
```

PAGE 0003 REMLT 9:34 AM MON., 20 FEB., 1978

```
      IF(HRMN.GT.0.95)CALL ERMES(1,IFIX(100.*HRMN),IREP)
      IF(HRMN.LT.0.05)CALL ERMES(2,IFIX(100.*HRMN),IREP)
C
C-----------------------------------------------------------------------
C 5. CALCULATE ERROR AND DERIVATIVE ERROR OF SMOOTHED INPUT DATA.
C
      HRNDOT = ALPHA*(HRMN-HRMNS)
      HRMNS = ALPHA*HRMN + (1.-ALPHA)*HRMNS
      ERR = HRMNS- HRMNSP
C
C-----------------------------------------------------------------------
C 6. CONTROL EQUATION.
C
      DELFSP = GPR*HRNDOT + GIRV*ERR*DELT
C
C-----------------------------------------------------------------------
C 7. CONVERT TO PULSES AND CHECK LIMITS OF ACTION.
C
      DELN = DELFSP +DELN
C                       PICK UP ROUND-OFF FROM LAST OUTPUT
      IF(ABS(DELN).GT.0.001)GOTO 110
      NUMP = 0
      GOTO 120
  110 NUMP = IFIX(DELN*1000.)
      DELN = AMOD(DELN,0.001)
C                       SAVE ROUND OFF OF LESS THAN ONE PULSE
  120 NPOS = ENG(8)*1000.
      IDIFF = NPOS - NUMPT
      IF(ABS(IDIFF).LT.25)GOTO 130
      CALL ERMES(3,IDIFF,IREP)
C                       CHECK CALCULATED SETPOINT POSITION AGAINST ACTUAL.
      NUMPT = NPOS
C
  130 IF(IABS(NUMP).LT.100)GOTO 140
      CALL ERMES(4,NUMP,IREP)
C                       LIMIT CHANGE TO 10%.
      NUMP = 100
C
  140 IF((NUMP+NUMPT).GT.0)GOTO 150
      CALL ERMES(5,NUMP,IREP)
C                       INHIBIT OUT OF RANGE OUTPUT
      NUMP = -NUMPT
C
  150 IF((NUMP+NUMPT).LT.1000)GOTO 160
      CALL ERMES(6,NUMP,IREP)
C                       INHIBIT OUT OF RANGE OUTPUT
      NUMP = 1000-NUMPT
C
C
C-----------------------------------------------------------------------
C 8. OUTPUT TO PULSER MODULE.
C
  160 NUMPT = NUMPT +NUMP
      CALL CAMAC(16,IPUL,NUMP,IQ)
C
```

```
PAGE 0004  REMLT  9:34 AM  MON., 20  FEB., 1978

C------------------------------------------------------------------------
C 9. UP-DATE CONTROL WORD FOR AUTO/MANUAL WATCHDOG (PROGRAM : WCHDG).
C
  200 ISCOP(1) = IOR(4,ISCOP(1))
C
C------------------------------------------------------------------------
C 10. LOCK REMLT ONTO ITS RESOURCE NUMBER AGAIN
C
      GO TO 100
C
C------------------------------------------------------------------------
      END

FTN4 COMPILER: HP92060-16092 REV. 1726

**  NO WARNINGS **  NO ERRORS **   PROGRAM = 00408      COMMON = 00758
```

```
PAGE 0001  FTN.   9:36 AM  MON., 20  FEB., 1978

FTN4,L,T
      PROGRAM CLIME(2,30),050178BDR 230178BDR
C-----------------------------------------------------------------------
C       CLIME - CONTROLS THE LIME-SOLIDS RATIO BY REGULATING THE
C               LIME-WHEEL SPEED. LINEAR PROPORTIONAL CONTROL WITH
C               OVER-RIDE IS USED. THE RATIO IS REDUCED WHEN ALL
C              THREE SATS. ARE OUT OF GAS.
C
C
C      NOMENCLATURE :
C
C                ZR=SWITCHING FLAG (EXPONENTIALLY SMOOTHED)
C               ESF=EXPONENTIAL SMOOTHING FACTOR
C                ZA=OUT-OF-GAS FLAG FOR A-SAT.
C                ZB=OUT-OF-GAS FLAG FOR B-SAT.
C                ZC=OUT-OF-GAS FLAG FOR C-SAT.
C               GOR=OVER-RIDE PROPORTIONAL GAIN
C             PHCSP=SET-POINT FOR C-SAT PH CONTROL
C               FCR=LIME/SOLIDS FLOW CONTROL RATIO
C              FCRS=SET-POINT FOR FCR
C
C-----------------------------------------------------------------------
C
C                        ------ COMMON  ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C    ENG   - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C    ADCV  - A/D VOLTAGES (UPDATED BY SCAD)
C    CDACV - D/A VOLTAGES (UPDATED BY CDAC)
C
C    SAFCOD- SATURATOR FLOW CONTROL DATA
C    CLFLOD- CLOUDY LIQUOR FLOW DATA
C    REMLTD- REMELT CONTROL DATA
C    CLIMED- CONTROL LIME DATA
C    GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C    GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C    GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C    FILCYD- FILTER CYCLE MONITER DATA
C    SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C    ISAMT - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C    ISMUL - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C    IRN   - RESOURCE NUMBERS
C    ICIN  - CONTACT STATUS IN (UPDATED BY SCCS)
C    ICOUT - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C    ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C    ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C    ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
```

PAGE 0002 CLIME 9:36 AM MON., 20 FEB., 1978

```
C----------------------------------------------------------------------
C
C
      EQUIVALENCE(CLIMED(1),FCRS),(CLIMED(2),GOR)
      EQUIVALENCE(PHCSP,GASFCD(3))
      EQUIVALENCE(CLIMED(3),FCR),(CLIMED(4),VOLTS)
      EQUIVALENCE(CLIMED(5),ALPHA),(CLIMED(6),ZR)
C                    CLIMED(7) = IZ
C
C----------------------------------------------------------------------
C
C     INITAILISE CONSTANTS
C
      ZR=0.
      IREP = 60
C
      CCAO=10.314
C           %CAO IN LIME SLURRY AT DENSITY 1.090 TON CU.M.
      GOR=2.
C
      ALPHA=0.2
      BRIX=ENG(3)
      FLOW=ENG(23)
      SADV=ADCV(3)
C
      PHC = ENG(22)
      IGASC = IAND(ISCOP(2),000100B)
      IF(IGASC.EQ.0)PHCSP = ENG(22)
C                       NO OVERWRITE IF "GASFC" RUNNING.
C
C----------------------------------------------------------------------
C
C     WAIT UNTIL RESOURCE NUMBER RELEASED BY ENGUN
C
  100 CALL RNRQ(2,IRN(14),IDUM)
      CALL SWITF(5)
C
C----------------------------------------------------------------------
C   CALCULATE SAT. FEED RATE IN TONS/HOUR.
C
      BRIX = BRIX*(1.-ALPHA) + ALPHA*ENG(3)
      FLOW = FLOW*(1.-ALPHA) + ALPHA*ENG(23)
      SADV = SADV*(1.-ALPHA) + ALPHA*ADCV(3)
C                       EXPONENTIAL SMOOTHING OF INPUT DATA
C
      SGPB = 1.23 +0.013*SADV
      SFR = FLOW*SGPB
      SLIDS = SFR*BRIX/100.
C
C----------------------------------------------------------------------
C     CHECK AUTO/MANUAL STATUS & FEEDBACK SIGNAL FOR CHANGES.
C
      MANL = IAND(ICIN(4),40B)
C             MANL EQUALS ZERO ON MANUAL.
      IF(MANL.EQ.40B)GOTO 110
```

PAGE 0003 CLIME 9:36 AM MON., 20 FEB., 1978

```
      FLIM = ENG(9)/1.183
      FCRS = FLIM*CCAO/SLIDS
      IF(ABS(VOLTS-ADCV(9)).LE.0.100)GOTO 110
C                ZR NOT RESET IF NO CHANGE MADE IN MANUAL MODE.
      VOLTS = ADCV(9)
      ZR = 0.
      GOTO 250
C
C-----------------------------------------------------------------------
C     CHECK IF GAS FLOW CONTROL LOOPS RUNNING.
C
  110 NOGO1 = IAND(ISCOP(2),000160B)
      NOGO2 = IAND(ICIN(4),000700B)
      NOGO = NOGO1 + NOGO2
C                      NO LIME CONTROL IF GAS CONTROL OFF.
      IF(NOGO.EQ.001060B)GOTO 140
      CALL ERMES(2,0,IREP)
      ZR=0.
      FLIM = ENG(9)/1.183
      FCRS = FLIM*CCAO/SLIDS
      GOTO 250
C
C-----------------------------------------------------------------------
C
C    CALCULATE SAMPLING INTERVAL,SMOOTHING FACTOR &SET DEFAULT
C
  140 IZ=0
C          DEFAULT ON OUT-OF-GAS SWITCH
C
      DELT=FLOAT(ISAMT*ISMUL(2)*ISMUL(6))
C
      ESF=1.*DELT/(60.*45.)
C
C-----------------------------------------------------------------------
C
C       GET ERROR IN C-SAT PH
C
      PHC=ENG(22)*ALPHA + (1.-ALPHA)*PHC
      ER=PHC-PHCSP
C
C-----------------------------------------------------------------------
C    TEST FOR OUT-OF-GAS CONDITION
C
      ZA = ENG(24)/GASFAD(8)
      ZB = ENG(25)/GASFBD(8)
      ZC = ENG(26)/GASFCD(5)
      IF((ZA.GE.0.97).AND.(ZB.GE.0.97))IZ=1
      IF(.NOT.((ZA.LE.0.1).AND.(ZB.LE.0.1)))GOTO 200
      IF(ZC.GE.0.97)GOTO 200
      IF(ER.LT.0.)IZ=1
      IF(.NOT.((ER.GT.0.).AND.(ZC.LT.0.1)))GOTO 150
      CALL ERMES(1,0,IREP)
      GO TO 100
  150 IF((ER.GT.0.).AND.(ZC.GT.0.1))IZ=-1
C
```

PAGE 0004 CLIME 9:36 AM MON., 20 FEB., 1978

```
C-----------------------------------------------------------------------------
C   ADJUST LIME-SOLIDS RATIO SMOOTHLY
C
  200 ZR=(1.-ESF)*ZR+ESF*IZ
      FCR=FCRS*(1.-GOR*ZR*ER)
C
C-----------------------------------------------------------------------------
C
C         CALCULATE LIME FLOW RATE IN TONNNES/HR
C
      FLIM=FCR*SLIDS/CCAO
C
C-----------------------------------------------------------------------------
C
C        CONVERSION TO VOLTS FOR OUTPUT TO DAC.
C
      SPEED=1.183*FLIM
C     AT LIME FLOW=0.01292 CU.M./MIN./REV.
C        LIME DENSITY= 1.090 TON/CU.M.
      VOLTS=0.937*SPEED+0.836
C
C-----------------------------------------------------------------------------
C
C        OUTPUT CONTROL ACTION
C
  250 CALL CDAC(2,VOLTS)
      CLIMED(7) = FLOAT(IZ)
C
C-----------------------------------------------------------------------------
C
C     UPDATE 4-TH BIT IN ISCOP(1) FOR WCHDG
C
      ISCOP(1)=IOR(10B,ISCOP(1))
C
C-----------------------------------------------------------------------------
C
C       LOCK ON RESOURCE NUMBER
C
      GOTO 100
C
C-----------------------------------------------------------------------------
C
C
  300 CONTINUE
      END
```

FTN4 COMPILER: HP92060-16092 REV. 1726

** NO WARNINGS ** NO ERRORS ** PROGRAM = 00646 COMMON = 00758

```
PAGE 0001  FTN.   9:37 AM  MON., 20  FEB., 1978

FTN4,L
      PROGRAM GASFA(2,30),230178BDR 310178BDR 010278BDR
C-----------------------------------------------------------------------
C       GASFA - CONTROLS THE PH OUT OF A-SATURATOR BY REGULATING
C               THE GAS FEED RATE. A CASCADE CONTROL SYSTEM IS USED
C               WHERE THE GAS FLOW RATE SET-POINT IS ADJUSTED BY
C               PROPORTIONAL PLUS RESET ACTION FROM THE A-SAT PH
C               ERROR. THE GAS FLOW CONTROL VALVE SETTING IS ADJUSTED
C               BY RESET-ONLY ACTION TO MAINTAIN THE FLOW
C               SETPOINT. THE A-SAT PH SETPOINT IS REDUCED ONLY WHEN
C               C-SAT IS OUT OF GAS, IN WHICH CASE A SIMPLE PROPORTIONAL
C               OVER-RIDE IS BROUGHT INTO ACTION.
C
C
C      NOMENCLATURE :
C
C              GPS  = PROPORTIONAL GAIN FOR GAS FLOW SETPOINT
C              GIRS = INTEGRAL GAIN FOR GAS FLOW SETPOINT
C              GIF  = INTEGRAL GAIN FOR FLOW CONTROL
C              PHAC = CONTROL POINT FOR A-SAT. PH
C              PHASP = PH SET-POINT
C              VPASP = VALVE POSITION SET-POINT
C              IZC  = OUT-OF-GAS FLAG FOR C-SAT(OVER-RIDES A-SAT PH)
C              GOA  = OVER-RIDE PROPORTIONAL GAIN
C
C        ERROR MESSAGES :
C
C          1=A-SATURATOR OUT OF GAS.
C          2=A-SATURATOR GAS SUPPLY VALVE CLOSED.
C          3=WARNING - "DELT" REDUCED TOO LOW -  VALVE CONTROL AFFECTED
C
C-----------------------------------------------------------------------
C
C                  ------ COMMON  ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C    ENG   - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C    ADCV  - A/D VOLTAGES (UPDATED BY SCAD)
C    CDACV - D/A VOLTAGES (UPDATED BY CDAC)
C
C    SAFCOD- SATURATOR FLOW CONTROL DATA
C    CLFLOD- CLOUDY LIQUOR FLOW DATA
C    REMLTD- REMELT CONTROL DATA
C    CLIMED- CONTROL LIME DATA
C    GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C    GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C    GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C    FILCYD- FILTER CYCLE MONITER DATA
C    SERVOD- SERVOBALANS SCALE MONITOR DATA
```

PAGE 0002 GASFA 9:37 AM MON., 20 FEB., 1978

```
C
C    ISAMT - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C    ISMUL - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C    IRN   - RESOURCE NUMBERS
C    ICIN  - CONTACT STATUS IN (UPDATED BY SCCS)
C    ICOUT - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C    ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C    ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C    ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C-------------------------------------------------------------------------------
C
C
      EQUIVALENCE(GASFAD(1),PHAC),(GASFAD(2),GINDEP)
      EQUIVALENCE(GASFAD(3),GPS),(GASFAD(4),GIRS)
      EQUIVALENCE(GASFAD(5),GOA),(GASFAD(6),PHASP)
      EQUIVALENCE(GASFAD(7),GASA),(GASFAD(8),GASAMX)
      EQUIVALENCE(GASFAD(9),VLIM),(GASFAD(10),VPA)
      EQUIVALENCE(GASFCD(3),PHCSP)
C
C-------------------------------------------------------------------------------
C
C   INITIALISE CONSTANTS
C
      GIRS=30.
C       FLOW SETPOINT ADJUSTMENT INTEGRAL RESET TIME IN MINS.
      GPS = 0.25
C       FLOW SETPOINT ADJUSTMENT PROPORTIONAL GAIN
      GINDEP = 0.03125
C       FLOW CONTROL VALVE INTEGRAL RESET TIME IN MINS/SEC.
      GOA = 1.0
C       A-SAT. PH SET-POINT OVER-RIDE GAIN
C
      PHA = ENG(20)
C       A-SAT. PH FOR EXP. SMOOTHING
      PHACO = PHA
C       SET POINT PH LAST CYCLE (INITIALISED)
      PHC = ENG(22)
C       C-SAT PH FOR EXP. SMOOTHING
      PHASP = ENG(20)
C       A-SAT PH SETPOINT
C
      IGASFC = IAND(ISCOP(2),000100B)
      IF(IGASFC.NE.100B)PHCSP = GASFCD(3)
C
      VPA = 0.55
      VLIM = 0.65
      GASA = 0.5
C
      GASAMX = 2720.
C             MAXIMUM FLOW CONSTRAINT =  4350 CU.M/HR(1600CFM)
C
      ALPHA = .2
C
      IREP=60
```

```
C              SUPRESSION PERIOD (MINS.) FOR ERMES
C
C-----------------------------------------------------------------------
C
  100 CALL RNRQ(2,IRN(15),IDUM)
C         WAIT UNTIL RESOURCE NUMBER RELEASED BY ENGUN.
C
      CALL SWITF(4)
C
      IFLAG = IAND(ICIN(4),100B)
C                        AUTO/MANUAL SWITCH CHECK
      IF(IFLAG.NE.100B)GOTO 300
C
C-----------------------------------------------------------------------
C     CALCULATE SAMPLING INTERVAL
C
      DELT=FLOAT(ISAMT*ISMUL(2)*ISMUL(6))
      IF(DELT.LT.6)CALL ERMES(3,0,IREP)
C
C-----------------------------------------------------------------------
C      CALCULATE INTEGRAL GAIN
C
      GIRF = GINDEP*DELT
C                        INTEGRAL RESET TIME FOR CONTROL VALVE,MINS.
      GIF = 1./(60.*GIRF)
      GIS = 1./(60.*GIRS)
C
C-----------------------------------------------------------------------
C    CALCULATE SMOOTHED A&C PH'S, C-SAT PH ERROR & A-SAT PH CHANGE
C
      EAPDOT = ALPHA*(ENG(20)-PHA)
      PHA=ENG(20)*ALPHA + (1.-ALPHA)*PHA
      PHC=ENG(22)*ALPHA + (1.-ALPHA)*PHC
      ERPHC = PHC-PHCSP
C
C-----------------------------------------------------------------------
C   CALCULATE A-SAT CONTROL PH SETPOINT IN CASE C-SAT IS OUT OF GAS
C
      PHAC=PHASP-GOA*GASFCD(4)*ERPHC
      SPPDOT = PHAC-PHACO
      PHACO = PHAC
C
C-----------------------------------------------------------------------
C      CALCULATE A-SAT ERROR
C
      ERAPH = PHA-PHAC
C
C-----------------------------------------------------------------------
C     A-SAT GAS FLOW SETPOINT
C
      DELFA =  GPS*(EAPDOT-SPPDOT) + GIS*ERAPH*DELT
      GASA = GASA + DELFA
      IF((GASA*5436.6).GT.GASAMX)GASA = GASAMX/5436.6
      IF(GASA.LE.0.)GASA=0.01
C
```

PAGE 0004 GASFA 9:37 AM MON., 20 FEB., 1978

```
C-----------------------------------------------------------------------
C          CALCULATE PRESENT GAS FLOW RATE
C
       ARG = (ADCV(24)-2.)/8.
       IF(ADCV(24).GT.2.)GOTO 110
       FLOWA = 0.001
       GO TO 120
   110 FLOWA = SQRT(ARG)
C
C-----------------------------------------------------------------------
C          CALCULATE FLOW ERROR
C
   120 ERAF = FLOWA-GASA
C
C-----------------------------------------------------------------------
C      CALCULATE VALVE POSITION.
C
       DELVA = GIF*DELT*ERAF
C
       VPA = VPA - DELVA
       IF(VPA.GT.VLIM)VPA = VLIM
C
C-----------------------------------------------------------------------
C       CHECK IF VALVE POSITION LIMITING.
C
       IF(FLOWA.LT.(GASAMX/5436.6))GOTO 200
          CALL ERMES(1,0,IREP)
          GOTO 300
C
   200 IF(VPA.GT.0.)GOTO 300
         VPA = 0.
         CALL ERMES(2,0,IREP)
C
C-----------------------------------------------------------------------
C       OUTPUT CONTROL ACTION
C
   300 VPAO=10.*(1.-VPA)
       IF(VPAO.LE.(10.*(1.-VLIM)))VPAO=(10.*(1.-VLIM))
       CALL CDAC(3,VPAO)
C
C-----------------------------------------------------------------------
C     UPDATE 4-TH BIT IN ISCOP(1) FOR WCHDG
C
   400 ISCOP(1) = IOR(20B,ISCOP(1))
C
C-----------------------------------------------------------------------
C      LOCK ON RESOURCE NUMBER
C
CC     WRITE(7,1000)ERPHC,ERAPH,EAPDOT,SPPDOT,DELFA,GASA,ENG(24),FLOWA,
CC    1  GASFCD(4),ERAF,DELVA,VPA
C1000 FORMAT(//,2(6F12.6,/))
       GOTO 100
C
C-----------------------------------------------------------------------
C
```

```
PAGE 0005  GASFA  9:37 AM  MON., 20  FEB., 1978

 0221     500 CONTINUE
 0222         END


 FTN4 COMPILER: HP92060-16092 REV. 1726

 **  NO WARNINGS **  NO ERRORS **   PROGRAM = 00557      COMMON = 00758
```

PAGE 0001 FTN. 9:39 AM MON., 20 FEB., 1978

```
FTN4,L
      PROGRAM GASFB(2,30),230178BDR 310178BDR 010278BDR
C-----------------------------------------------------------------------
C       GASFB - CONTROLS THE PH OUT OF B-SATURATOR BY REGULATING
C               THE GAS FEED RATE. A CASCADE CONTROL SYSTEM IS USED
C               WHERE THE GAS FLOW RATE SET-POINT IS ADJUSTED BY
C               PROPORTIONAL PLUS RESET ACTION FROM THE B-SAT PH
C               ERROR. THE GAS FLOW CONTROL VALVE SETTING IS ADJUSTED
C               BY RESET ACTION ONLY TO MAINTAIN THE FLOW
C               SETPOINT. THE B-SAT PH SETPOINT IS REDUCED ONLY WHEN
C               C-SAT IS OUT OF GAS, IN WHICH CASE A SIMPLE PROPORTIONAL
C               OVER-RIDE IS BROUGHT INTO ACTION.
C
C
C      NOMENCLATURE :
C
C              GPS  = PROPORTIONAL GAIN FOR GAS FLOW SETPOINT
C              GIRS = INTEGRAL GAIN FOR GAS FLOW SETPOINT
C              GIF  = INTEGRAL GAIN FOR FLOW CONTROL
C              PHBC = CONTROL POINT FOR B-SAT. PH
C              PHBSP = PH SET-POINT
C              VPBSP = VALVE POSITION SET-POINT
C              IZC  = OUT-OF-GAS FLAG FOR C-SAT(OVER-RIDES B-SAT PH)
C              GOB  = OVER-RIDE PROPORTIONAL GAIN
C
C        ERROR MESSAGES :
C
C          1=B-SATURATOR OUT OF GAS.
C          2=B-SATURATOR GAS SUPPLY VALVE CLOSED.
C          3=WARNING - DELT REDUCED SO LOW THAT VALVE CONTROL AFFECTED
C
C-----------------------------------------------------------------------
C
C                   ------ COMMON ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1  SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2  GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3  SERVOD(20),DUMMY(50),
     4  ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5  ISCOP(3),IDUMY(50)
C
C   ENG    - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C   ADCV   - A/D VOLTAGES (UPDATED BY SCAD)
C   CDACV  - D/A VOLTAGES (UPDATED BY CDAC)
C
C   SAFCOD- SATURATOR FLOW CONTROL DATA
C   CLFLOD- CLOUDY LIQUOR FLOW DATA
C   REMLTD- REMELT CONTROL DATA
C   CLIMED- CONTROL LIME DATA
C   GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C   GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C   GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C   FILCYD- FILTER CYCLE MONITER DATA
C   SERVOD- SERVOBALANS SCALE MONITOR DATA
```

```
C
C   ISAMT - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C   ISMUL - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C   IRN   - RESOURCE NUMBERS
C   ICIN  - CONTACT STATUS IN (UPDATED BY SCCS)
C   ICOUT - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C   ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C   ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C   ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C-----------------------------------------------------------------------
C
C
      EQUIVALENCE(GASFBD(1),PHBC),(GASFBD(2),GINDEP)
      EQUIVALENCE(GASFBD(3),GPS),(GASFBD(4),GIRS)
      EQUIVALENCE(GASFBD(5),GOB),(GASFBD(6),PHBSP)
      EQUIVALENCE(GASFBD(7),GASB),(GASFBD(8),GASBMX)
      EQUIVALENCE(GASFBD(9),VLIM),(GASFBD(10),VPB)
      EQUIVALENCE(GASFCD(3),PHCSP)
C
C-----------------------------------------------------------------------
C
C  INITIALISE CONSTANTS
C
      GIRS=30.
C       FLOW SETPOINT ADJUSTMENT INTEGRAL RESET TIME IN MINS.
      GPS = 0.25
C       FLOW SETPOINT ADJUSTMENT PROPORTIONAL GAIN
      GINDEP = 0.03125
C       FLOW CONTROL VALVE INTEGRAL RESET TIME IN MINS/SEC.
      GOB = 1.
C       B-SAT. PH SET-POINT OVER-RIDE GAIN
C
      PHB = ENG(21)
C       B-SAT. PH FOR EXP. SMOOTHING
      PHBCO = PHB
C       SET POINT PH LAST CYCLE (INITIALISED)
      PHC = ENG(22)
C       C-SAT PH FOR EXP. SMOOTHING
      PHBSP = ENG(21)
C       B-SAT PH SETPOINT
C
      IGASFC = IAND(ISCOP(2),000100B)
      IF(IGASFC.NE.100B)PHCSP = GASFCD(3)
C
      VPB = 0.55
      VLIM = 0.65
      GASB = 0.5
C
      GASBMX = 2720.
C             MAXIMUM FLOW CONSTRAINT =  1600 CFM.
C
      ALPHA = .2
C
      IREP=60
```

PAGE 0003 GASFB 9:39 AM MON., 20 FEB., 1978

```
C              SUPRESSION PERIOD (MINS.) FOR ERMES
C
C-----------------------------------------------------------------------------
C
  100 CALL RNRQ(2,IRN(16),IDUM)
C         WAIT UNTIL RESOURCE NUMBER RELEASED BY ENGUN.
C
      CALL SWITF(3)
C
      IFLAG = IAND(ICIN(4),200B)
C                  AUTO/MANUAL SWITCH CHECK
      IF(IFLAG.NE.200B)GOTO 300
C
C-----------------------------------------------------------------------------
C     CALCULATE SAMPLING INTERVAL
C
      DELT=FLOAT(ISAMT*ISMUL(2)*ISMUL(6))
      IF(DELT.LT.6.)CALL ERMES(3,0,IREP)
C
C-----------------------------------------------------------------------------
C      CALCULATE INTEGRAL GAIN
C
      GIRF = GINDEP*DELT
C                  RESET TIME FOR CONTROL VALVE,MINS.
      GIF = 1./(60.*GIRF)
      GIS = 1./(60.*GIRS)
C
C-----------------------------------------------------------------------------
C    CALCULATE SMOOTHED B&C PH'S, C-SAT PH ERROR & B-SAT PH CHANGE
C
      EBPDOT = ALPHA*(ENG(21)-PHB)
      PHB=ENG(21)*ALPHA + (1.-ALPHA)*PHB
      PHC=ENG(22)*ALPHA + (1.-ALPHA)*PHC
      ERPHC = PHC-PHCSP
C
C-----------------------------------------------------------------------------
C   CALCULATE B-SAT CONTROL PH SETPOINT IN CASE C-SAT IS OUT OF GAS
C
      PHBC=PHBSP-GOB*GASFCD(4)*ERPHC
      SPPDOT = PHBC-PHBCO
      PHBCO = PHBC
C
C-----------------------------------------------------------------------------
C      CALCULATE B-SAT ERROR
C
      ERBPH = PHB-PHBC
C
C-----------------------------------------------------------------------------
C     B-SAT GAS FLOW SETPOINT
C
      DELFB =  GPS*(EBPDOT-SPPDOT) + GIS*ERBPH*DELT
      GASB = GASB + DELFB
      IF((GASB*5436.6).GT.GASBMX)GASB = GASBMX/5436.6
      IF(GASB.LE.0.)GASB=0.01
C
```

```
PAGE 0004  GASFB  9:39 AM  MON., 20  FEB., 1978

      C-----------------------------------------------------------------------
      C         CALCULATE PRESENT GAS FLOW RATE
      C
            ARG = (ADCV(25)-2.)/8.
            IF(ADCV(25).GT.2.)GOTO 110
            FLOWB = 0.001
            GO TO 120
        110 FLOWB = SQRT(ARG)
      C
      C-----------------------------------------------------------------------
      C         CALCULATE FLOW ERROR
      C
        120 ERBF = FLOWB-GASB
      C
      C-----------------------------------------------------------------------
      C    CALCULATE VALVE POSITION.
      C
            DELVB = GIF*DELT*ERBF
      C                    GIF*DELT IS INDEPENDENT OF DELT!!!
      C
            VPB = VPB - DELVB
            IF(VPB.GT.VLIM)VPB = VLIM
      C
      C-----------------------------------------------------------------------
      C     CHECK IF VALVE POSITION LIMITING.
      C
            IF(FLOWB.LT.(GASBMX/5436.6))GOTO 200
               CALL ERMES(1,0,IREP)
               GOTO 300
      C
        200 IF(VPB.GT.0.)GOTO 300
              VPB = 0.
              CALL ERMES(2,0,IREP)
      C
      C-----------------------------------------------------------------------
      C     OUTPUT CONTROL ACTION
      C
        300 VPBO=10.*(1.-VPB)
            IF(VPBO.LE.(10.*(1.-VLIM)))VPBO=(10.*(1.-VLIM))
            CALL CDAC(4,VPBO)
      C
      C-----------------------------------------------------------------------
      C   UPDATE 5-TH BIT IN ISCOP(1) FOR WCHDG
      C
        400 ISCOP(1) = IOR(40B,ISCOP(1))
      C
      C-----------------------------------------------------------------------
      C    LOCK ON RESOURCE NUMBER
      C
            GOTO 100
      C
      C-----------------------------------------------------------------------
      C
        500 CONTINUE
            END
```

```
PAGE 0005  GASFB  9:39 AM  MON., 20  FEB., 1978

 FTN4 COMPILER: HP92060-16092 REV. 1726

 **  NO WARNINGS **  NO ERRORS **   PROGRAM = 00556      COMMON = 00758
```

PAGE 0001 FTN. 9:40 AM MON., 20 FEB., 1978

```
FTN4,L
      PROGRAM GASFC(2,30),050178BDR 230178BDR 010278BDR
C-----------------------------------------------------------------------
C      GASFC - CONTROLS THE PH OUT OF C-SATURATOR BY REGULATING
C                THE GAS FEED RATE. PROPORTIONAL PLUS INTEGRAL
C               CONTROL IS USED.
C               WHEN OUT OF GAS, THE FLAG IZC (EQUIVALENT TO
C               GASFCD(4) IN COMMON) IS SET TO 1.
C
C
C     NOMENCLATURE :
C
C               GPS   = PROPORTIONAL GAIN FOR GAS FLOW SETPOINT
C               GPF   = PROPORTIONAL GAIN FOR GAS VALVE CONTROL
C               GIRS = INTEGRAL GAIN FOR GAS FLOW SETPOINT
C               GIRF = INTEGRAL GAIN FOR GAS VALVE CONTROL
C               PHCSP= C-SAT PH SET-POINT
C                IZC = OUT-OF-GAS FLAG FOR C-SAT.
C
C        ERROR MESSAGES :
C
C                1 = C-SATURATOR OUT OF GAS.
C                2 = C-SATURATOR GAS SUPPLY VALVE CLOSED.
C                3 = SAMPLING INTERVAL TOO SHORT FOR CONTROL ALGORITHM.
C
C-----------------------------------------------------------------------
C
C                 ------ COMMON  ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C   ENG   - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C   ADCV  - A/D VOLTAGES (UPDATED BY SCAD)
C   CDACV - D/A VOLTAGES (UPDATED BY CDAC)
C
C   SAFCOD- SATURATOR FLOW CONTROL DATA
C   CLFLOD- CLOUDY LIQUOR FLOW DATA
C   REMLTD- REMELT CONTROL DATA
C   CLIMED- CONTROL LIME DATA
C   GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C   GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C   GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C   FILCYD- FILTER CYCLE MONITER DATA
C   SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C   ISAMT - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C   ISMUL - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C   IRN   - RESOURCE NUMBERS
C   ICIN  - CONTACT STATUS IN (UPDATED BY SCCS)
C   ICOUT - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
```

```
PAGE 0002  GASFC  9:40 AM  MON., 20  FEB., 1978

C     ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C     ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C     ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C-------------------------------------------------------------------------------
C
C
        EQUIVALENCE(GASFCD(1),GPS),(GASFCD(2),GIRS)
        EQUIVALENCE(GASFCD(3),PHCSP),(GASFCD(5),GASCMX)
        EQUIVALENCE(GASFCD(7),ALPHA),(GASFCD(8),GINDEP)
        EQUIVALENCE(GASFCD(9),GASC),(GASFCD(10),VLIM)
C                     GASFCD(4) = FLOAT(IZC)
C                     GASFCD(6) = VPC
C
C-------------------------------------------------------------------------------
C
        VPC=0.55
C                     INITIALISED VALVE STEM POSITION
        VLIM = 0.65
C                     VALVE OPENING LIMIT
        GASC = 0.5
C                     INITIALISED GAS FLOW SETPOINT (NORMALISED)
        GASCMX = 1360.
C                     C-SAT MAXIMUM FLOW RATE, CU.M/HR
        ALPHA = 0.2
C                     EXPONENTIAL SMOOTHING DEFAULT VALUE
        IZC=0
C                     DEFAULTED TO "NOT OUT OF GAS"
        PHCSP=ENG(22)
C                     C-SAT PH SETPOINT.
        PHC=ENG(22)
C                     C-SAT PH (EXPONENTIALLY SMOOTHED)
        GIRS=30.
C             SETPOINT INTEGRAL RESET TIME IN MINS.
        GPS=0.5
C             SETPOINT PROPORTIONAL GAIN
        GINDEP = 0.02417
C             INDEPENDENT GAS VALVE INTEGRAL RESET TIME,MINS/SEC
C             NOTE :-  THIS RESET-ONLY ALGORITHM IS POSSIBLE ONLY BECAUSE
C                      THE NORMAL VALVE RESPONSE,(INCLUDING DEAD-TIME),
C                      IS OF THE ORDER OF 6-7 SECONDS. THE ALGORITHM
C                      DETERIORATES IF "DELT" IS PERMITTED TO FALL TO
C                      BELOW THIS RESPONSE TIME.
C
        IREP=60
C             SUPRESSION PERIOD (MINS.) FOR ERMES
C
C-------------------------------------------------------------------------------
C
    100 CALL RNRQ(2,IRN(17),IDUM)
C          WAIT UNTIL RESOURCE NUMBER RELEASED BY ENGUN.
C
C
        CALL SWITF(2)
        IFLAG = IAND(ICIN(4),400B)
```

PAGE 0003 GASFC 9:40 AM MON., 20 FEB., 1978

```
      IF(IFLAG.EQ.0)GOTO 500
C                         PROTECTION AGAINST INTEGRAL CONTROL WIND-UP
C
C-----------------------------------------------------------------------
C     CALCULATE SAMPLING INTERVAL
C
      DELT=FLOAT(ISAMT*ISMUL(2)*ISMUL(6))
      IF(DELT.LT.6.)CALL ERMES(3,0,IREP)
C
C-----------------------------------------------------------------------
C      CALCULATE INTEGRAL GAINS
      GIS=1./(60.*GIRS)
C                         INTEGRAL GAIN FOR GAS FLOW SETPOINT
      GIRF = GINDEP*DELT
C                         CONTROL VALVE INTEGRAL GAIN,MINS.
C      NOTE:-             GIRF IS DEPENDENT ON SAMPLING INTERVAL.
      GIF = 1./(60.*GIRF)
C                         INTEGRAL GAIN FOR CONTROL VALVE ACTION.
C
C-----------------------------------------------------------------------
C      CALCULATE C-SAT ERROR & ITS DERIVATIVE
C
      ECDOT=ALPHA*(ENG(22)-PHC)
      PHC=ENG(22)*ALPHA + (1.-ALPHA)*PHC
C
      ERC=PHC-PHCSP
C
C-----------------------------------------------------------------------
C      CALCULATE NEW FLOW SETPOINT
C
      DELFC=GPS*ECDOT+GIS*ERC*DELT
      GASC = GASC + DELFC
      IF((GASC*2718.3).GT.GASCMX)GASC=GASCMX/2718.3
      IF(GASC.LE.0.)GASC=0.01
C
C-----------------------------------------------------------------------
C       CALCULATE PRESENT GAS FLOW RATE
C
      ARG = (ADCV(26)-2.)/8.
      IF(ADCV(26).GT.2.)GOTO 160
      FLOWC = 0.001
      GOTO 170
  160 FLOWC = SQRT(ARG)
C
C-----------------------------------------------------------------------
C       CALCULATE FLOW ERROR AND ITS DERIVATIVE
C
  170 ERFC = FLOWC-GASC
C
C-----------------------------------------------------------------------
C       CALCULATE VALVE STEM POSITION
C
      DELVC = GIF*ERFC*DELT
C                         GIF*DELT IS INDEPENDENT OF DELT!!!
      VPC = VPC-DELVC
```

```
PAGE 0004  GASFC  9:40 AM  MON., 20  FEB., 1978

            IF(VPC.GT.VLIM)VPC = VLIM
      C
      C------------------------------------------------------------------------
      C     CHECK IF VALVE POSITION LIMITING.
      C
            IF(FLOWC.LT.(0.96*GASCMX/2718.3))GOTO 200
      C
               IZC=1
               CALL ERMES(1,0,IREP)
               GOTO 300
      C
        200 IZC=0
            IF(VPC.GT.0.)GOTO 300
      C
              VPC=0.
              CALL ERMES(2,0,IREP)
      C
      C------------------------------------------------------------------------
      C     OUTPUT CONTROL ACTION
      C
        300 VPCO=10.*(1.-VPC)
            IF(VPCO.LE.(10.*(1.-VLIM)))VPCO= 10.*(1.-VLIM)
      C                          AIR-TO-CLOSE
            CALL CDAC(5,VPCO)
      C
            GASFCD(4)=FLOAT(IZC)
            GASFCD(6)=VPC
      C
      C------------------------------------------------------------------------
      C   UPDATE 6-TH BIT IN ISCOP(1) FOR WCHDG
      C
        500 ISCOP(1)=IOR(100B,ISCOP(1))
      C
      C------------------------------------------------------------------------
      C    LOCK ON RESOURCE NUMBER
      C
            GOTO 100
      C
      C------------------------------------------------------------------------
      C
        400 CONTINUE
            END


FTN4 COMPILER: HP92060-16092 REV. 1726


**  NO WARNINGS **  NO ERRORS **   PROGRAM = 00482       COMMON = 00758
```

```
PAGE 0001  FTN.   9:25 AM  MON., 20  FEB., 1978

FTN4,L
      PROGRAM FILCY(3,60),141277PSH,010278PSH
C
C       PROGRAMMED BY P.S.HUSSEY.
C
C               VERSION : 1-2-1978
C
C******************************************************
C      THIS PROGRAM CALCULATES FILTER CYCLE TIMES AND STORES THEM IN
C      A FILE CALLED "FILDAT" ON THE DISC AT 8H00 EVERY MORNING.
C      "FILDAT" CONSISTS OF THE SEQUENCE OF RECORDS FORMED BY THE
C      ARRAY "IFCT" MADE UP OF THE FOLLOWING, FOR I=1,12 &J=1,12:
C          IFCT(1,K)=TIME AT WHICH DATA WAS STORED,K=1 TO 5.
C          IFCT(1,6)=SAMPLING FREQUENCY (SECS.).
C
C     IFCT(4I-2,J)=TIME  SINCE MOST RECENT START(MINS).
C     IFCT(4I-1,J)  =VARIABLE PRESS.(CONST. FLOW) OPERATING PERIOD(MINS)
C     IFCT(4I,J)=TOTAL FILTER CYCLE OPERATING PERIOD(MINS).
C     IFCT(4I+1,J)=FITERABILITY(% OF MAXIMUM)
C
C
C      IN THE ARRAY "FSTIM",
C          FSTIM(1,K) ACCUMULATES THE FILTRATION PARAMETER
C          FSTIM(2,K) STORES TIME WHEN FILTER STARTED.
C          IFILS STORES THE NUMBER OF THE FILTER WHICH STARTED LAST.
C
C
C          WHEN THE PROGRAM RUNS FOR THE FIRST TIME, THE ARRAYS
C      "IFCT" AND "FSTIM" ARE INITIALISED TO ZERO AND -1.  FOLLOWING
C      THIS SECTION IS A RESOURCE HOLD STATEMENT WHICH LOCKS THE
C      PROGRAM OUT UNTIL IT IS RELEASED BY "SCCS".THE TIME SINCE MIDNIGHT
C      (TNEW) IS THEN IMMEDIATELY DETERMINED.
C         A FILTERABILITY PARAMETER IS UPDATED EVERY TIME FILCY IS
C      RELEASED.
C           THE ORIGIN AND NATURE OF THE CONTACT
C      CHANGE IS ANALYSED BY COMPARING THE PRESENT STATUS
C      IN ICNEW WITH THE OLD STATUS IN ICOLD. IF FILTER K WENT "ON"
C      ITS START TIME IS STORED IN FSTIM(2,K) AND DIFFERENCED FROM
C      THE NEXT MOST RECENT START TIME, STORED IN FSTIM(2,IFILS).
C      IFILS IS THEN UPDATED TO K.
C        WHEN THE PRESSURE SWITCH IS TRIGGERED THE FILTERABILITY
C      IS CALCULATED AND STORED IN IFCT.
C           WHEN THE VALVE SWITCH IS TRIGGERED
C      OR THE FILTER GOES OFF-LINE, ITS OPERATING
C      TIME IS CALCULATED BY SUBTRACTING FSTIM(2,K) FROM TNEW
C      (A TEST FOR PASSING MIDNIGHT IS DONE USING TOLD). THIS VALUE IS
C      THEN STORED IN THE APPROPRIATE ARRAY POSITION IN "IFCT".
C          IF THE PRESSURE AND VALVE SWITCHES ARE NOT TRIGGERED,
C      A ZERO WILL APPEAR AT THE POSITION IN THE IFCT ARRAY WHERE
C      THE DATA FOR THAT CYCLE WOULD NORMALLY GO.
C      IF ANY COLUMN IN "IFCT" IS FULL, THE WHOLE ARRAY IS DUMPED
C      ONTO DISC. THE CURRENT VALUE IS THEN PLACED IN THE 2ND  ROW
C      OF THE "CLEAN" IFCT ARRAY.TOLD IS UPDATED AND THE PROGRAM
C      WAITS FOR THE NEXT CALL.
C
```

PAGE 0002 FILCY 9:25 AM MON., 20 FEB., 1978

```
C       ALSO NOTE THAT :
C           IRAY(1,K)=STATUS OF FILTER K
C                    =-1 IF ON-LINE & PRESSURE VARIABLE
C                    =0  IF ON-LINE & PRESSURE CONSTANT.
C                    =+1 IF OFF-LINE.
C
C         THE DATA IS STORED IN THE IFCT ARRAY IN BLOCKS OF 3 WORDS.
C            IRAY(2,K)=POSITION OF BLOCK IN ARRAY
C            ISTAT    =POSITION OF WORD IN BLOCK
C            ICNT     =POSITION OF WORD IN ARRAY.
C
C             NNEW1/NOLD1 = STATUS OF ON/OFF SWITCH
C             NNEW2/NOLD2 = STATUS OF PRESSURE SWITCH
C             NNEW3/NOLD3 = STATUS OF VALVE SWITCH
C
C
C       FILCY ONLY STORES DATA FOR A GIVEN FILTER FROM THE FIRST
C       FILTER START-TIME ONWARDS. ALL PREVIOUS DATA IS
C       TREATED AS GARBAGE AND OVER-WRITTEN. FILCY ALSO DOES  ON-LINE
C       MEASUREMENT OF THE FILTERABILITY,"FILBY".IN GENERAL :
C          FILBY=FILPR/(FILAR*FILAR*DPDT)
C        WHERE.....
C                FILPR=MU*FS*FS
C                MU=VISCOSITY(CENTIPOISE)
C                FS=SET-POINT FLOWRATE TO A SINGLE FILTER(CU.M/HR)
C                FILAR=FILTER AREA(SQ.M)
C                     =NLFIL(K)*APERL
C                NLFIL=NO. OF OPERATIVE LEAVES PER FILTER
C                APERL=AREA PER LEAF
C                DPDT=RATE OF CHANGE OF PRESSURE DROP ACROSS
C              FILTER UNDER CONSTANT FLOW CONDITIONS(KPA/S)
C
C       APPROXIMATING DPDT=VPP/DELP
C            WHERE  VPP=OPERATING PERIOD UNDER VARIABLE PRESSURE(S)
C                   DELP=RANGE OF VARIABLE PRESSURE(KPA)
C       AND SINCE MEAN(FILPR)=INTEGRAL(FILPR)/VPP
C       THE PROGRAM USES THE FORMULA :
C             FILBY=INTEGRAL(FILPR)/(FILAR*FILAR*DELP)
C       IF FILAR AND DELP ARE SET EQUAL TO UNITY,FILBY WILL BE EQUAL
C       TO THE VARIABLE PRESSURE PERIOD IN HOURS.
C
C       THE SUMMATION OF FILPR IS DONE OVER THE PERIOD VPP AND
C       STORED IN FSTIM(1,K). WHEN FILTER K "STARTS", FSTIM(1,K)
C       IS INITIALISED TO ZERO AND INCREMENTED BY FILPR*DELT
C       EVERY TIME THE PROGRAM IS RELEASED.(DELT = TIME SINCE
C       LAST CONTACT STATUS CHANGE (SECS.)) FILPR IS TRANSFERRED
C       FROM ENGUN VIA COMMON. EVERY TIME A VARIABLE PRESSURE
C       PERIOD TERMINATES FILBY IS CALCULATED AND PRINTED.
C
C         ERROR MESSAGES :
C            K=FILTERABILITY OF FILTER K(%),K=1,12
C           13=UNABLE TO OPEN FILE FILDAT.
C           14=UNABLE TO WRITE TO FILE FILDAT.
C           15=UNABLE TO CLOSE FILE FILDAT.
C
```

PAGE 0003 FILCY 9:25 AM MON., 20 FEB., 1978

```
C
C------------------------------------------------------------------------------
C
C                      ------ COMMON ------
C
       COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C     ENG    - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C     ADCV   - A/D VOLTAGES (UPDATED BY SCAD)
C     CDACV  - D/A VOLTAGES (UPDATED BY CDAC)
C
C     SAFCOD- SATURATOR FLOW CONTROL DATA
C     CLFLOD- CLOUDY LIQUOR FLOW DATA
C     REMLTD- REMELT CONTROL DATA
C     CLIMED- CONTROL LIME DATA
C     GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C     GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C     GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C     FILCYD- FILTER CYCLE MONITER DATA
C     SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C     ISAMT - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C     ISMUL - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C     IRN   - RESOURCE NUMBERS
C     ICIN  - CONTACT STATUS IN (UPDATED BY SCCS)
C     ICOUT - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C     ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C     ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C     ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C------------------------------------------------------------------------------
C
C
C
       INTEGER IFCT(50,12),IT(5),IRAY(2,12),IDCB(144),IBUF(600)
       INTEGER NNEW1(12),NNEW2(12),NOLD1(12),NOLD2(12),NLFIL(12)
       INTEGER FILDAT(3),IDUN(12)
       DIMENSION FSTIM(2,12),STRTS(10),NNEW3(12),NOLD3(12)
       EQUIVALENCE (FILBY,FILCYD(1))
C
       DATA NLFIL/12*56/,FILDAT/2HFI,2HLD,2HAT/
C
C---------------------------------------------------------------------
C
C      ****INITIALISATION SECTION ******
C
C
       DO 200  K=1,12
       FSTIM(1,K)=0.
       FSTIM(2,K)=-1.
```

```
PAGE 0004  FILCY  9:25 AM  MON., 20  FEB., 1978

      IRAY(1,K)=1
      IRAY(2,K)=1
      DO 200 J=1,50
      IFCT(J,K)=0
  200 CONTINUE
       IFCT(1,12) = 1
C                        FLAG TO RFLDT ON FIRST DUMP OF DATA
      TOLD=-1.
      APERL=2.089
C                        SQ.M/FILTER LEAF
      DELP=200.
C                        ASSUMING A START POINT OF 50KPA WITH A SWITCH POINT
C                        OF 250KPA.
      IFILS=0
C
C-----------------------------------------------------------------------------
C
C     *****MAIN PROGRAM ******
C
C
C     **********WAIT UNTIL RESOURSE NUMBER RELEASED *******
  300 CALL RNRQ(2,IRN(7),IDUM)
      CALL SWITF(6)
C
C     ***********WHAT TIME IS IT?
      CALL EXEC(11,IT,IYEAR)
C
C     **********CALCULATE HOURS SINCE MIDNIGHT****
      TNEW=IT(4)+(IT(3)+IT(2)/60.)/60.
C
C     ***** COLLECT LATEST STATUS VALUES ****
C
      ICNEW1=ICIN(1)
      ICNEW2=ICIN(2)
      ICNEW3=ICIN(3)
C
      IF(TOLD.EQ.-1)GOTO 1050
C     ****** CALC TIME SINCE LAST CONTACT STATUS CHANGE *****
C
C    *****  DUMP DATA ARRAY IFCT(50,12) TO FILDAT FILE AT 8H00  *****
C
      IF(.NOT.((TOLD.LT.8.).AND.(TNEW.GT.8.))) GOTO 310
C
      IRTN=310
      GOTO 1200
C
  310 DELT=TNEW-TOLD
        IF(DELT.LT.0.)DELT=DELT+24.
C
C
C-----------------------------------------------------------------------------
C
C     ***MASK OFF SECTIONS OF STATUS WORDS ****
C
      DO 345 J=1,12
```

PAGE 0005 FILCY 9:25 AM MON., 20 FEB., 1978

```
          NNEW1(J)=IBIT(J,ICNEW1)
          NOLD1(J)=IBIT(J,ICOLD1)
          IF(J.GE.5)GOTO 340
          NNEW2(J)=IBIT(J+12,ICNEW1)
          NOLD2(J)=IBIT(J+12,ICOLD1)
          NNEW3(J+8)=IBIT(J,ICNEW3)
          NOLD3(J+8)=IBIT(J,ICOLD3)
          GOTO 345
  340     NNEW2(J)=IBIT(J-4,ICNEW2)
          NOLD2(J)=IBIT(J-4,ICOLD2)
          NNEW3(J-4)=IBIT(J+4,ICNEW2)
          NOLD3(J-4)=IBIT(J+4,ICOLD2)
  345 CONTINUE
C
C
C------------------------------------------------------------------
C
C        ***** CALCULATE PREREQUISITE FILTERABILITY INFORMATION
C
C
      B=ENG(3)
      T=ENG(14)
      Z=-1.234*B/(111-B)+246.527/(111+T)+659.543*B/((111-B)*(111+T))
      AMU=EXP(Z-2.257)
C
C
C
C  *** HOW MANY FILTERS ARE ON? ***
      NUMB = 0
      DO 320 J=1,12
        NUMB = 12 - NNEW1(J)
  320 CONTINUE
C  *** CALCULATE THE AVERAGE FLOW RATE PER FILTER ***
      DO 330 J=1,12
        ENG(J+32) = ENG(23)/NUMB
        FILPR = ENG(J+32)*ENG(J+32)*AMU
        FSTIM(1,J) = FSTIM(1,J) + FILPR*DELT
  330 CONTINUE
C
C
C
C-----------------------------------------------------------------
C
      DO 1000 K=1,12
C
C
C     *****  CHECK STATUS OF MECHANICAL SWITCH ******
C
      IF(NNEW1(K).EQ.NOLD1(K))GOTO 400
      IF(NNEW1(K).EQ.1)GOTO 490
C
C
C--------------------------------------------------------------------
C
  350 CONTINUE
```

PAGE 0006 FILCY 9:25 AM MON., 20 FEB., 1978

```
C
C        ****** THEN FILTER BROUGHT ON-LINE (PRESSURE VARIABLE) *****
C
                   ISTAT=-1
         IF(IRAY(1,K).EQ.1)GOTO 360
         WRITE(1,3000)K,IRAY(1,K),ISTAT
   3000 FORMAT("FILTER OPERATING SEQUENCE AWRY. FILTER #",I3,
       1  " LAST STATE=",I3,". CURRENT STATE=",I3)
         GOTO 1000
C
    360                 FSTIM(1,K)=0.
                        IDUN(K)=0
                        FSTIM(2,K)=TNEW
             IF(IFILS.EQ.0)GOTO 370
C
                   TINT=TNEW-FSTIM(2,IFILS)
                   IF(TINT.LT.0.)TINT=TINT+24.
                   ICNT=IRAY(2,K)*4-2
                   IFCT(ICNT,K)=IFIX(TINT*60.+0.5)
                   IFILS=K
C
C       **** RUNNING AVERAGE OF 10 FILTER STARTS ****
C
           IF(ISTAT.NE.-1)GOTO 800
           DO 700 J=1,9
             STRTS(11-J) = STRTS(10-J)
    700    CONTINUE
           STRTS(1) = TINT*60.
           AVST = 0.
           DO 710 J=1,10
             AVST = AVST + STRTS(J)
    710    CONTINUE
           AVST = AVST/10.
           FILCYD(2) = AVST
           WRITE(1,4000)K,IT(4),IT(3),STRTS(1),AVST
   4000    FORMAT("FILCY*** FILTER #",I2," ON AT ",I2,"H",I2,
       1     ". STARTS=",F6.1," : AV.STARTS=",F6.1,"MINS.",/)
C
                   GOTO 900
C
    370 IFILS=K
        GOTO 900
C
C
C-------------------------------------------------------------------
C
C         ****** CHECK STATUS OF PRESSURE SWITCH *****
C
    400 IF(.NOT.((NNEW2(K).EQ.1).AND.(NOLD2(K).EQ.0)))GOTO 450
C        ****** IGNOR PRESSURE SWITCH OFF STATUS ******
C
C          **** DO WE HAVE A START TIME? ****
C
        IF(FSTIM(2,K).LT.0.)GOTO 1000
C
```

PAGE 0007 FILCY 9:25 AM MON., 20 FEB., 1978

```
         IF(IDUN(K).EQ.1)GOTO 1000
C                    IGNOR CONTACT BOUNCE
C
C         ******* ELSE CALCULATE FILTERABILITY *****
C
C
                       FILAR=NLFIL(K)*APERL
                    FILBY=FSTIM(1,K)/(FILAR*FILAR*DELP)
                    IFBY = (FILBY*100./3.1432E-03) + 0.5
C      3.1432E-3 = MAXIMUM ESTIMATED FILBY VALUE WHEN:-
C                      FLOW=11 CU.M/FILTER/HOUR
C                      T   =82 DEG. C
C                      BRIX=68
C                      AMU = 11.85 C.POISE
C                      DELP= 200 KPA
C                      NLFIL= 56 LEAVES/FILTER
C                      APERL= 2.089 SQ.M/LEAF
C                      SIGMA(DELT)= 6 HOURS
C   3.1432E-3= FLOW*FLOW*AMU*SIGMA(DELT)/(DELP*NLFIL*NLFIL*APERL*APERL)
            WRITE(1,2000)K,IT(4),IT(3),IFBY
  2000      FORMAT("FILCY*** FILTER #",I2,"  TIME-",I2,"H",I2,
     1    "   :FILTERABILITY=",I6,/)
C   ****  OUTPUT FILTERABILITY AS % OF EXPECTED MAXIMUM
C
C
         ICNT=4*IRAY(2,K)+1
         IFCT(ICNT,K)=IFBY
C
         IDUN(K)=1
         GOTO 1000
C
C--------------------------------------------------------------------
C
C
C      ***** CHECK STATUS OF VALVE SWITCH ****
C
     450 IF(.NOT.((NNEW3(K).EQ.1).AND.(NOLD3(K).EQ.0)))GOTO 1000
C           IGNOR VALVE SWITCH OFF STATUS
C
C
C        **** ELSE VARIABLE PRESSURE PERIOD TERMINATED ****
C
C
            ISTAT = 0
            IF(IRAY(1,K).NE.-1)WRITE(1,3000)K,IRAY(1,K),ISTAT
C
C        ***IGNOR CONTACT BOUNCE ***
C
         IF(IRAY(1,K).EQ.0)GOTO 1000
C
         GOTO 560
C
C-------------------------------------------------------------
C
C            ******** ELSE FILTER TAKEN OFF LINE *****
```

PAGE 0008 FILCY 9:25 AM MON., 20 FEB., 1978

```
C
  490   ISTAT=1
      IF(IRAY(1,K).NE.0)WRITE(1,3000)K,IRAY(1,K),ISTAT
      IF(IRAY(1,K).EQ.1)GOTO 1000
C
C-----------------------------------------------------------------------
C
C
C           ****** LOOK BEFORE YOU LEAP! ****
C
C
  500 CONTINUE
C
C
C
C         ******** IS THE IFCT ARRAY FULL ? *****
      IF(IRAY(2,K).LE.12)GOTO 560
C          ****** IF  >12  DUMP ON DISC ******
C
      IRTN=560
      GOTO 1200
C
C
  560 CONTINUE
C        ******DO WE HAVE A START TIME? *****
C
      IF(FSTIM(2,K).LT.0.)GOTO 900
C
C
C     ***** GET START TIME OF FILTER K *****
C
C
      TINT=TNEW-FSTIM(2,K)
C
C-----------------------------------------------------------------------
C
C     *** STORE OPERATING TIME OF FILTER K IN MINUTES ****
C
  600 ICNT=4*IRAY(2,K)+ISTAT-1
      IF(TINT.LT.0.)TINT=TINT+24.
      IFCT(ICNT,K)=IFIX(TINT*60. + 0.5)
C
C         ****UPDATE RECORD COUNT ******
C
  800 IRAY(2,K)=IRAY(2,K)+1
  900 IRAY(1,K)=ISTAT
 1000 CONTINUE
C
C
C-----------------------------------------------------------------------
C
C     **** UPDATE AND WAIT FOR NEXT PROGRAM CALL***
 1050     TOLD=TNEW
      ICOLD1=ICNEW1
      ICOLD2=ICNEW2
```

```
PAGE 0009  FILCY  9:25 AM  MON., 20  FEB., 1978

       ICOLD3=ICNEW3
           GOTO 300
C
C
C---------------------------------------------------------------------
C
C
C     ******ROUTINE FOR DUMPING IFCT DATA INTO DISC FILE "FILDAT"
C
C  (RETURNS TO STATEMENT NUMBER GIVEN BY IRTN)
C
 1200 DO 1250 I=1,5
        IFCT(1,I) = IT(I)
 1250 CONTINUE
      IFCT(1,6) = ISAMT
      DO 1410 J=1,50
      DO 1400 I=1,12
        LAST = 4*IRAY(2,I)
        IF(IFCT(LAST,I).NE.0)IRAY(2,I) = IRAY(2,I)+1
          M = (J-1)*12 + I
      LIM=4*(IRAY(2,I)-1)+1
        IFCT(50,I) = IRAY(2,I)-1
      IF((J.GT.LIM).AND.(J.LT.50))GOTO 1300
          IBUF(M) = IFCT(J,I)
      GOTO 1400
 1300 IBUF(M)=0
 1400 CONTINUE
 1410 CONTINUE
      CALL OCEND(IDCB,FILDAT,IERR)
      IF(IERR.LT.0)CALL MESAG(-13,IERR)
      CALL WRITF(IDCB,IERR5,IBUF,600)
      IF(IERR5.LT.0)CALL MESAG(-14,IERR5)
      CALL CLOSE(IDCB)
C
C     RE-INITIALISE ARRAYS :
C
      DO 1420 K=1,12
      DO 1420 I=1,4
      M=4*IRAY(2,K)+I-3
      IFCT(I+1,K)=IFCT(M,K)
 1420 CONTINUE
C
      DO 1450 I=1,12
        IRAY(2,I) = 1
        DO 1450 L=6,50
          IFCT(L,I) = 0
 1450   CONTINUE
      IFCT(1,12) = 0
C
      IF(IRTN.EQ.310)GOTO 310
      IF(IRTN.EQ.560)GOTO 560
C
      END
```

```
PAGE 0010  FILCY  9:25 AM  MON., 20  FEB., 1978

 FTN4 COMPILER: HP92060-16092 REV. 1726

 **  NO WARNINGS **  NO ERRORS **  PROGRAM = 03247      COMMON = 00758
```

PAGE 0001 FTN. 9:22 AM MON., 20 FEB., 1978

```
FTN4,L,T
      PROGRAM  MESEG,3,80
C
C-----------------------------------------------------------------------------
C
C  MESEG - SCHEDULED BY MESAG TO PRINT EITHER INFORMATIVE OR
C          ERROR MESSAGE.
C                          VERSION:  4-10-1977.
C-----------------------------------------------------------------------------
C
C
C     QUEUE SCHEDULE WITH WAIT
C     PARAMETERS:
C       IP1,IP2,IP3 - 6 LETTER NAME (ORIGINATING PROGRAM)
C       IP4 - >0 - MESSAGE NUMBER
C             <0 - ERROR NUMBER
C       IP5 - PARAMETER (OPTIONAL)
C
C
C  A RECORD OF ALL MESSAGES IS ALSO KEPT IN THE FILE "ERROR"
C  TO LIMIT THE DISC SEARCHING TIME THE ERROR FILES ARE LIMITED TO 500
C  RECORDS. A NEW FILE WITH AN INCREMENTED SERIAL NO I.E. ERROR 1,
C  ERROR 2, .... ETC. IS CREATED
C
C  NOTE:- 1.MODIFIED TO ACCEPT NEGATIVE ERROR MESSAGES FROM "ERMES".
C           THESE ARE DECODED AND HANDLED AS POSITIVE ERROR CALLS, BUT
C           THE NEGATIVE VALUE IS SENSED FOR SENDING NEGATIVE MESSAGES
C           TO LU=1 ONLY, WHILST OTHER MESSAGES CAN BE SENT ELSEWHERE.
C         2.COMMON HAS BEEN ADDED FOR ACCESS TO ISAMT & ISMUL VALUES.
C           THEREFORE LOAD WITH REVERSE COMMON. (:RU,LOAD,38)
C
C****************************************************************************
C
      INTEGER IP(5),ITIME(5),ERROR(3),ERRORX(3)
      INTEGER IBUF(11),IDCB(400),JBUF(40)
C
C                 ------ COMMON  ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C   ENG   - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C   ADCV  - A/D VOLTAGES (UPDATED BY SCAD)
C   CDACV - D/A VOLTAGES (UPDATED BY CDAC)
C
C   SAFCOD- SATURATOR FLOW CONTROL DATA
C   CLFLOD- CLOUDY LIQUOR FLOW DATA
C   REMLTD- REMELT CONTROL DATA
C   CLIMED- CONTROL LIME DATA
C   GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C   GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
```

PAGE 0002 MESEG 9:22 AM MON., 20 FEB., 1978

```
C     GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C     FILCYD- FILTER CYCLE MONITER DATA
C     SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C     ISAMT - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C     ISMUL - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C     IRN   - RESOURCE NUMBERS
C     ICIN  - CONTACT STATUS IN (UPDATED BY SCCS)
C     ICOUT - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C     ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C     ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C     ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C-----------------------------------------------------------------------------
C
        DATA ERROR/2HER,2HRO,2HR /,ERRORX/2HER,2HRO,2HR0/
C
        CALL RMPAR(IP)
C
        CALL EXEC(11,ITIME)
C
C
        GO TO 30
C                            SKIP OVER ROUTINE TO LOG ERRORS ON DISC FILE.
C                            (REMOVE AT A LATER STAGE IF/WHEN FOUND NECESSARY)
C
C
C    OPEN ERROR FILE AND SKIP TO END, PICKING UP LAST ERROR MESSAGE
C    IF THERE IS ONE.
C
      1 CALL OPEN(IDCB,IERR,ERROR,1,0,0,400)
        IF(IERR.NE.-6)GOTO 15
          CALL CREAT(IDCB,IERR,ERROR,100,10,0,-2,400)
          IF(IERR.LT.0)WRITE(1,1021)IERR
 1021   FORMAT("UNABLE TO CREATE ERROR FILE"I6)
          CALL OPEN(IDCB,IERR,ERROR,1,0,0,400)
   15   IF(IERR.LT.0) WRITE(1,1020) IERR
 1020   FORMAT("CANNOT OPEN ERROR FILE, ERROR NO"I6)
C
C    SEARCH FOR END OF FILE, CREATE NEW FILE OF MORE THAN 500 RECORDS
C
        DO 20 IREC=1,500
          CALL READF(IDCB,IERR,IBUF,11,LEN)
          IF(LEN.LT.0) GOTO 30
 20     CONTINUE
        IF(IREC.LT.500)GOTO 30
C
C         ROUTINE TO OPEN A NEW FILE.
C
          IFILN=0
   25     IFILN=IFILN+1
          IF(IFILN.GT. 9) WRITE(1,1030)
          ERRORX(3)=2HR0+IFILN
 1030     FORMAT("MAX NO OF ERROR FILES EXCEEDED -"
     1      "DELETE FILES ERRORX (X=1 TO 9) OR COPY FILES TO MT")
```

PAGE 0003 MESEG 9:22 AM MON., 20 FEB., 1978

```
            CALL NAMF(IDCB,IERR,ERROR,ERRORX)
      C                     RENAME
            IF(IERR.EQ.-2) GOTO 25
      C                     DUPLICATE NAME, GO TRY NEXT ONE
            IF(IERR.LT.0) WRITE(1,1040)IERR
      1040  FORMAT("RENAME ERROR "I6)
            WRITE(1,1050) ERRORX
      1050   FORMAT(72("*"),//,"NEW ERROR FILE CREATED - FILE NAME = ",3A2,
           1   //,72("*"))
      C                     REMNDER OF FILE NO CURRENTLY IN USE
      C
            CALL CREAT(IDCB,IERR,ERROR,100,10,0,-2,400)
      C                     100 BLOCKS ON LU 2 (SYSTEM DISC)
            IF(IERR.EQ.-6) CALL CREAT(IDCB,IERR,ERROR,100,10,0,-13,400)
      C                     NO SPACE, TRY REMOVABLE PLATTER
            IF(IERR.LT.0) WRITE(1,1060)
      1060  FORMAT(/,72("*"),/,5X,"NO DISC SPACE FOUND ON EITHER DISC FOR"
           1   " NEW ERROR MESSAGE FILE. ",/,5X,"***** HELP !!! *****"
           2   "     ******** URGENT ********",/,5X,"PURGE REDUNDANT FILES."
           3   /,72("*"))
      C                     NOGO, GIVE UP
            CALL OPEN(IDCB,IERR,ERROR,1,0,0,400)
      C                     NON- EXCLUSIVE OPEN
      C                     THIS ALSO REWINDS FILE, I.E. OPENS AT RECORD NO 1
      C
         30 CONTINUE
      C***********EXTRACT THE PACKED REPETITION COUNT******
      C
      C      UNPACK TO FIND THE NUMBER OF COUNTS TRANSMITTED BY "ERMES"
      C      AND A POSITIVE ERROR NUMBER
      C
             IFLAG =0
             JIP = IP(4)
             IF(JIP.LT.0)IFLAG=1
             IF(IABS(JIP).GT.100)GOTO 40
             ICNT = 0
             IER = IABS(JIP)
             GOTO 70
         40  ICNT = IABS(JIP)/100
             IER = (IABS(JIP) - ICNT*100)
      C
      C***********RECORD ERROR/MESSAGE ON DISC**********
      C
         70  DO 75 I=1,3
             IBUF(I)=IP(I)
      75     IBUF(I+5)=ITIME(6-I)
             IBUF(4)=IER
             IBUF(5)=IP(5)
             IRATE = 60/(ISAMT*ISMUL(2)*ISMUL(6))
             IMIN  = ICNT/IRATE
      C                               MEANINGFUL CHANGE OF A COUNT TO TIME,
      C                               APPLYING ONLY TO THE FACTORY CONTROL
      C                               PROGRAMMES.
      C
             IBUF(9)=IMIN
```

```
PAGE 0004  MESEG  9:22 AM  MON., 20  FEB., 1978

      C
            GOTO 77
      C                         SUPPRESSION OF ERROR FILING ON DISC (TEMPORARY ?!)
         76 CALL WRITF(IDCB,IERR,IBUF,11)
              IF(IERR.LT.0) WRITE(1,1031) IERR
      1031    FORMAT("WRITE ERROR"I6"  IN PROGRAM MESEG")
            CALL CLOSE(IDCB)
      C
      C
      C   **********PRINT ERROR MESSAGE*************
      C
         77 LU=1
      C   .......................PUT LU=9 IN PREVIOUS LINE WHEN TERMINAL
      C                            BECOMES AVAILABLE.
            IF(IFLAG.EQ.1)LU=1
      C
      C***  SEARCH FILE FOR A SPECIFIED ERROR NUMBER AND PRINT IT.
      C
            IP(3)=IOR(IAND(IP(3),177400B),000077B)
      C              ****** APPEND "?" TO PROGRAM NAME FOR ERROR FILE NAME.
            KHC=0
            CALL OPEN(IDCB,IERR,IP,1)
            IF(IERR.LT.0)WRITE(LU,7000)IERR
       7000 FORMAT("ERROR IN OPENING ERROR FILE ASSOCIATED WITH CALLING",
           1  " PROGRAM",/,20X,"(IERR = ",I4,"  )")
      C
      C
         80 CALL READF(IDCB,IERR1,JBUF,40,LEN)
                IF(LEN.LT.0)GOTO 90
      C                          THEN EOF FOUND
            IF(IXOR(IAND(JBUF(1),177400B),021400B).EQ.0)KHC=KHC+1
      C                        LOOK FOR # IN FIRST CHARACTER.
            IF(KHC.GT.IER)GOTO 90
            IF(KHC.LT.IER)GOTO 80
      C
            WRITE(LU,1000)
       1000 FORMAT(" ")
            CALL EXEC(2,LU,JBUF,LEN)
      C                OUTPUT CONTENTS OF THIS RECORD.
            WRITE(LU,1010)(IBUF(J),J=1,9)
       1010 FORMAT(2A2,A1," (#",I2,", VALUE=",I4," DAY",I4,", TIME",I3,
           1"H",I2,") FOR ",I3," MINUTES SINCE LAST REPORTED")
            GOTO 80
      C
         90 CALL CLOSE(IDCB)
      C
      C
            END


FTN4 COMPILER: HP92060-16092 REV. 1726


**  NO WARNINGS **  NO ERRORS **   PROGRAM = 01315      COMMON = 00758
```

```
PAGE 0001  FTN.   9:25 AM  MON., 20  FEB., 1978

FTN4,L,T
      SUBROUTINE ERMES(IERR,IPRAM,IREP)
C
C***********************************************************************
C           "ERMES" = ERROR MESSAGES.
C           ERMES SUPPRESSES ERROR MESSAGES PRINTED BY MESEG.
C
C                        IERR        = A POSITIVE  MESSAGE NUMBER OR
C                                      A NEGATIVE ERROR MESSAGE NUMBER
C                                      (SEE LISTING IN CALLING PROGRAM)
C                        ICNT(N1,N2)=NUMBER OF COUNTS OF ERROR MESSAGE NUMBER
C                                     "N2", FROM CALLING PROGRAM OF CODE = N1,
C                                     SINCE LAST REPORTING THE MESSAGE.
C                        IREP        = PERIOD (IN MINUTES) DURING WHICH THE
C                                      MESSAGE IS TO BE SUPPRESSED.
C         THIS SUBROUTINE WILL SUPPRESS ERROR MESSAGES IN THE CONTROL
C         PROGRAM FOR A PERIOD EQUAL TO IREP (IN MINUTES). THE INFORMATION
C         PASSED TO MESAG, FOR PRINTING & STORING ON DISC FILE, IS PACKED
C         INTO AN INTEGER NUMBER WHERE THE TWO LEAST SIGNIFICANT DIGITS
C         ARE THE ERROR MESSAGE NUMBER AND THE NEXT DIGITS ARE THE NUMBER
C         OF OCCURRENCES SINCE LAST REPORTING THE MESSAGE.
C                                   VERSION : 18-8-1977
C***********************************************************************
      INTEGER IERA(60),ICNT(60),IT(5)
      DATA ICNT/60*0/
      DATA IERA/60*0/
      CALL EXEC(11,IT,IYEAR)
      IF(IERR.EQ.0) RETURN
      IZ=ISIGN(1,IERR)
      IERR=IABS(IERR)
      IER  = IERR*IZ
      TNEW=FLOAT(60*IT(4)+IT(3))
      IF(IERA(IERR).EQ.0)GOTO 200
      TOLD=FLOAT(IERA(IERR))
      IF((TNEW-TOLD).LT.0.)TOLD=TOLD-1440.
      IF(IFIX(TNEW-TOLD).GE.IREP)GOTO 100
      ICNT(IERR)=ICNT(IERR)+1
      RETURN
  100 IER =IZ*(100*ICNT(IERR)+IERR)
  200 CALL MESAG(IER,IPRAM)
      IERA(IERR)=IFIX(TNEW)
      ICNT(IERR)=0
      RETURN
      END

FTN4 COMPILER: HP92060-16092 REV. 1726

**  NO WARNINGS **  NO ERRORS **   PROGRAM = 00269      COMMON = 00000
```

PAGE 0001 FTN. 9:24 AM MON., 20 FEB., 1978

```
FTN4,L
      SUBROUTINE MESAG(MESN,IPRAM),050178ADH 240577??
C
C-----------------------------------------------------------------------------------------
C
C                         MESAG - MESSAGE OUTPUT
C
C                         VERSION : 24-5-1977
C MOD  5-1-78 : QUEUE SCHEDULE WITHOUT WAIT
C-----------------------------------------------------------------------------------------
C
C  MESAG PROVIDES A GENERAL PURPOSE METHOD OF OUTPUTTING AN INFORMATIVE
C  OR ERROR MESSAGE TO THE SYSTEM CONSOLE.  IT SCHEDULES THE BACKGROUND
C  PROGRAM MESEG TO PRINT THE ERROR AND THEREFORE AVOIDS FORMATTED I/O
C  STATEMENTS IN FOREGROUND PROGRAMS.
C
C  USE:
C    CALL MESAG(MESN,IPRAM)
C      MESN - MESSAGE NUMBER <0 ERROR "-MESN" IN PROGRAM "NAME"
C                            >0 MESSAGE "MESN" IN PROGRAM "NAME"
C      IPRAM - AN OPTIONAL PARAMETER TO ENABLE ADDITIONAL INFORMATION
C              TO BE PASSED (E.G. THE IERR FROM A FMGR PROGRAM CALL)
C
C  SEE MESEG LISTING FOR ADDITIONAL FUNCTIONS PERFORMED.
C
      INTEGER MESEG(3),NAME(3)
      DATA MESEG/2HME,2HSE,2HG /
C
      CALL GEPNM(NAME)
C                  GET EXECUTING PROGRAM NAME
      CALL EXEC(24,MESEG,NAME(1),NAME(2),NAME(3),MESN,IPRAM)
C                  SCHEDULE MESEG
C                  QUEUE SCHEDULE WITHOUT WAIT TO AVOID
C                  TIME OUT IN WCHDG AND CONTROL PROGS
      RETURN
      END
```

FTN4 COMPILER: HP92060-16092 REV. 1726

** NO WARNINGS ** NO ERRORS ** PROGRAM = 00042 COMMON = 00000

PAGE 0001 FTN. 9:15 AM MON., 20 FEB., 1978

```
FTN4,L,T
      PROGRAM STRUP(11,80),??ADH 041077BDR 040178ADH
C-----------------------------------------------------------------------*
C
C                           STRUP - START UP PROGRAM.
C
C                             VERSION :  4-10-1977  (BDR)
C
C                      LOAD IN BACKGROUND, USING REVERSE COMMON
C-----------------------------------------------------------------------
C
C                       ------ COMMON ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1  SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2  GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3  SERVOD(20),DUMMY(50),
     4  ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5  ISCOP(3),IDUMY(50)
C
C   ENG    - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C   ADCV   - A/D VOLTAGES (UPDATED BY SCAD)
C   CDACV  - D/A VOLTAGES (UPDATED BY CDAC)
C
C   SAFCOD- SATURATOR FLOW CONTROL DATA
C   CLFLOD- CLOUDY LIQUOR FLOW DATA
C   REMLTD- REMELT CONTROL DATA
C   CLIMED- CONTROL LIME DATA
C   GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C   GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C   GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C   FILCYD- FILTER CYCLE MONITER DATA
C   SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C   ISAMT  - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C   ISMUL  - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C   IRN    - RESOURCE NUMBERS
C   ICIN   - CONTACT STATUS IN (UPDATED BY SCCS)
C   ICOUT  - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C   ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C   ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C   ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C-----------------------------------------------------------------------
C
      INTEGER M(12),S(6),ITIME(5),ENGUN(3),HANGO(3)
      INTEGER SCAD(3),FMGR(3),SCCS(3),PACIR(3),SAMT(3)
      INTEGER YDAY,YEAR,HOURS,DAY,SECS
      EQUIVALENCE(ITIME(2),SECS),(ITIME(3),MINS),(ITIME(4),HOURS),
     1             (ITIME(5),YDAY)
      DATA M(1),M(2),M(3),M(4),M(5),M(6),M(7),M(8),M(9),M(10),M(11),
     1    M(12)/31,28,31,30,31,30,31,31,30,31,30,31/
      DATA S(1),S(2),S(3),S(4),S(5),S(6)/2H0 ,2H1 ,2H2 ,2H3 ,2H4 ,2H5 /
      DATA SCAD/2HSC,2HAD,2H1 /,FMGR/2HFM,2HGR,2H  /
      DATA SCCS/2HSC,2HCS,2H  /,PACIR/2HPA,2HCI,2HR /
```

```
PAGE 0002  STRUP  9:15 AM  MON., 20  FEB., 1978

      DATA HANGO/2HHA,2HNG,2HO /,ENGUN/2HEN,2HGU,2HN /
C
C     THE START UP PROGRAM PERFORMS THE
C     FOLLOWING FUNCTIONS:
C                       1. WRITE HEADING AND GET TIME AND DATE
C                       2. INITIALISE CAMAC CRATE
C                       3. ALLOCATE RESOURCE NUMBERS
C                       4. SCHEDULE HANGO TO START PACIR AND TO
C                          INITIALISE COMMON FROM THE FILE "COMDAT".
C                       5. START CONTROL PROGRAMS
C
C------------------------------------------------------------------------
C 1.HEADING AND DATE,TIME
C
      CALL RMPAR(ITIME)
      YEAR=ITIME(1)
      IF(YEAR.LT.1978)YEAR=1978
C
10    WRITE(1,1000)
      READ(1,*)IDAY,MONTH
      IF((MONTH.LT.1).OR.(MONTH.GT.12))GOTO 10
      IF(MOD(YEAR,4).EQ.0)M(2) = 29
      IF((IDAY.LT.1).OR.(IDAY.GT.M(MONTH))) GOTO 10
      YDAY = IDAY
      IF (MONTH .EQ.1) GOTO 30
      DO 20 I=1,MONTH-1
20      YDAY=YDAY + M(I)
30    WRITE(1,1010)
      READ(1,*) HOURS,MINS,SECS
C                 GET TIME
      CALL SETTI(YEAR,ITIME,IRESP)
      IF(IRESP.NE.0) GOTO 10
C                 SET TIME
      CALL PTAD(1)
C                 PRINT TIME AND DATE TO VERIFY
C
C------------------------------------------------------------------------
C 2.INITIALIZE CAMAC CRATE AND DO LAM GRADER TEST
C
      ICRAT=1
      CALL CAMCO(2**(ICRAT-1),IERR)
      IF(IERR.NE.0)CALL CAMER(IERR,0,ICRAT*512)
      CALL CAMZC(ICRAT,IERR)
      IF(IERR.NE.0)CALL CAMER(IERR,0,ICRAT*512)
C
      CALL DECLR (ILAMG,ICRAT,23,0)
C
      DO 50 J=1,16
        I=ISHFT(1,J-1)
        CALL CAMAC (16,ILAMG,I,IQ)
        CALL CAMAC (0,ILAMG,I1,IQ)
        IF(I.NE.I1) WRITE(1,340) I,I1
50    CONTINUE
C
340   FORMAT("LAM GRADER TEST ERROR: WROTE ",I5," BUT READ ",I5)
```

```
PAGE 0003  STRUP  9:15 AM  MON., 20  FEB., 1978

C-----------------------------------------------------------------------
C 3.DE-ALLOCATE AND THEN RE-ALLOCATE ALL RESOURCE NUMBERS
C
      DO 500 IRNI=1,20
         CALL RNRQ(140040B,IRN(IRNI),ISTAT)
C                       CLEAR (DE-ALLOCATE) + NO WAIT OR ABORT
        GOTO 510
509   IIDIOT=0
C                       IGNORE ERRORS
510   CONTINUE
C
        CALL RNRQ(140020B,IRN(IRNI),ISTAT)
C                       GLOBAL ALLOCATE + NO WAIT +NO ABORT
           GOTO 520
519   IIDIOT=0
520      CONTINUE
C                       IGNORE ERRORS
500   CONTINUE
C-----------------------------------------------------------------------
C 4. SET PACIR GOING
C
      CALL EXEC(9,HANGO,1)
C                       SCHEDULE HANGO WITH PARAMETER = 1 TO START PACIR
C                       IMMEDIATELY. THIS ALSO READS COMMON FROM DISC FILE.
C                       START AND STOP TIMES ARE NOT REQUESTED. RUN HANGO
C                       DIRECTLY TO DO THIS WITH PARAMETER = 0.
C
C-----------------------------------------------------------------------
C 5.CONTROL PROGRAMS
C
      CALL EXEC(10,SCAD)
C                       SCAN A TO D - IMMEDIATE SCHEDULE NO WAIT
      CALL EXEC(10,SCCS)
C                       SCAN CONTACT SENSE - IMMEDIATE SCHEDULE NO WAIT
      CALL EXEC(10,ENGUN)
C                       ENGINEERING UNITS CONVERSION
C
C
C
C-----------------------------------------------------------------------
C
      STOP
1000  FORMAT(//"HULETTS REFINERY CONTROL PROJECT"//"SET DATE AND TIME"
     1        //"DAY,MONTH ? ")
1010  FORMAT(/"HOURS,MINS,(SECS) ? ")
1020  FORMAT(/"SAMPLING TIMES-MASTER AND SUB-MULTIPLES"
     1         /"ISAMT,SMUL5,SMUL6")
      END


FTN4 COMPILER: HP92060-16092 REV. 1726


**  NO WARNINGS **  NO ERRORS **   PROGRAM = 00469      COMMON = 00758
```

```
PAGE 0004  FTN.   9:15 AM  MON., 20  FEB., 1978

      SUBROUTINE SETTI(IYEAR,ITIME,IRESP)
C
C  --- SETTI --- SET TIME BY CALL TO MESSS ------------------------------
C
C ITIME HAS SAME FORMAT AS EXEC(11) COMMAND
C IRESP IS RESPONSE TO SET TIME COMMAND, ERROR IF.NE.0
C
C
      DIMENSION IPB(33),ITIME(5)
      DATA IPB(1),IPB(2),IPB(3),IPB(4)/2,2HTM,2H  ,2H  /
      DATA IPB(5),IPB(9),IPB(13),IPB(17),IPB(21),IPB(33)/1,1,1,1,1,6/
      DATA IPB(25),IPB(26),IPB(29),IPB(30)/4*0/
C
C                          FINISH SETTING UP PARSE BUFFER
      IPB(6)=IYEAR
      IPB(10)=ITIME(5)
      IPB(14)=ITIME(4)
      IPB(18)=ITIME(3)
      IPB(22)=ITIME(2)
C
C                          DO INVERSE PASS TO CONVERT DATA TO ASCII COMMAND
      CALL INPRS(IPB,IPB(33))
C                          EXECUTE COMMAND BY CALL TO MESSS
      IRESP=MESSS(IPB,48)
C                          INPRS RETURNS 8 CHARACTERS/PARAM I.E. MESSS CNT =8*6
      IF(IRESP.EQ.0)RETURN
C                          INVALID CALL, PRINT RESPONSE ON SYSTEM CONSOLE
      CALL EXEC(2,1,IPB,-IRESP)
      RETURN
      END

FTN4 COMPILER: HP92060-16092 REV. 1726

**  NO WARNINGS **  NO ERRORS **   PROGRAM = 00124      COMMON = 00000
```

PAGE 0001 FTN. 10:24 AM WED., 26 APR., 1978

```
FTN4,L,T
      PROGRAM HANGO
C*********************************************************************
C          HANGO - "HANG-UP"  AND/OR  "GO"
C          FOR SCHEDULING TEMPORARY SUSPENSION OF PACIR.
C          ALSO USED FOR COLD START BY SHEDULE FROM STRUP WITH PARAM 1=1
C          NOTE : HANGO MUST BE LOADED INTO FOREGROUND
C
C          HANGO CAUSES PACIR TO SUSPEND ITSELF BY SETTING ISMUL(1)=-1 .
C          PACIR IS SCHEDULED IN SUBROUTINE RCDSP.
C
C                      VERSION :  4-10-1977  (BDR)
C*********************************************************************
C
      INTEGER PACIR(3),IT(5),IP(5),COMDAT(3),IDCB(400),ITOT(358),IC(38)
      DIMENSION CMC(160)
C
C                    ------ COMMON  ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1   SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2   GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3   SERVOD(20),DUMMY(50),
     4   ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5   ISCOP(3),IDUMY(50)
C
C    ENG   - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C    ADCV  - A/D VOLTAGES (UPDATED BY SCAD)
C    CDACV - D/A VOLTAGES (UPDATED BY CDAC)
C
C    SAFCOD- SATURATOR FLOW CONTROL DATA
C    CLFLOD- CLOUDY LIQUOR FLOW DATA
C    REMLTD- REMELT CONTROL DATA
C    CLIMED- CONTROL LIME DATA
C    GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C    GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C    GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C    FILCYD- FILTER CYCLE MONITER DATA
C    SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C    ISAMT - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C    ISMUL - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C    IRN   - RESOURCE NUMBERS
C    ICIN  - CONTACT STATUS IN (UPDATED BY SCCS)
C    ICOUT - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C    ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C    ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C    ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C---------------------------------------------------------------------
C
      EQUIVALENCE(ISMUL(1),ISMUL1),(IC(1),ITOT(1)),(CMC(1),ITOT(39))
      DATA PACIR/2HPA,2HCI,2HR /,COMDAT/2HCO,2HMD,2HAT/
C
C---------------------------------------------------------------------
```

PAGE 0002 HANGO 10:24 AM WED., 26 APR., 1978

```
C   1. CHECK FOR IMMEDIATE STARTUP REQUESTED BY STRUP.
C
         CALL RMPAR(IP)
         LU = 1
         IF(IP(1).EQ.1) GOTO 110
C                            START IMMEDIATELY
C
C-----------------------------------------------------------------------------
C   2. REQUEST SUSPEND & RESTART TIMES FROM THE OPERATOR.
C
         CALL EXEC(11,IT,IYEAR)
      10 WRITE(LU,1000)IT(5)
         READ(LU,*)ISTOP,IY,IZ
         ASTOP = IZ + 60.*(IY + 24.*ISTOP)
         WRITE(LU,1100)
         READ(LU,*)ISTART,IY,IZ
         START = IZ + 60.*(IY + 24.*ISTART)
         IF((ISTOP.GT.0).AND.(ISTART.GT.0).AND.((START-ASTOP).LT.0))
        1   GOTO 10
C
C-----------------------------------------------------------------------------
C   3. ACT ON IMMEDIATE RESPONSE REQUESTS.
C
         IF(ISTOP.EQ.0)GOTO 40
         IF(ISTART.EQ.0)GOTO 110
         IF((ISTOP.LT.0).AND.(ISTART.LT.0))STOP 0001
C                                  END IF NO START TIME AVAILABLE.
C
C-----------------------------------------------------------------------------
C   4. CHECK CURRENT TIME AGAINST INPUT TIMES.
C
      20 CONTINUE
         CALL EXEC(11,IT,IYEAR)
         IF((IT(5).NE.ISTOP).AND.(IT(5).NE.ISTART))GOTO 30
         TNOW = IT(3) + 60.*(IT(4) + 24.*IT(5))
           IF((TNOW.LT.ASTOP).AND.(ISTOP.NE.-1))GOTO 30
           IF((TNOW.GE.START).AND.(ISTART.NE.-1))GOTO 110
           IF((ISMUL1.NE.-1).AND.(TNOW.GE.ASTOP).AND.(ISTOP.NE.-1))GOTO 40
C
C-----------------------------------------------------------------------------
C   5. WAIT FOR ONE MINUTE IF NO ACTION REQUIRED.
C
      30 CALL WAIT(60,2,IDUM)
         GOTO 20
C
C-----------------------------------------------------------------------------
C   6. UPDATE COMDAT FILE.
C
      40 CALL OPEN(IDCB,IERR,COMDAT,10,0,0,400)
         IF(IERR.LT.0)WRITE(LU,1500)COMDAT
C
         DO 50 I=1,32767
           CALL READF(IDCB,IERR,ITOT,358,LEN)
C                SKIP TO END OF FILE
           IF(LEN.EQ.-1)GOTO 60
```

```
PAGE 0003  HANGO 10:24 AM  WED., 26  APR., 1978

      50 CONTINUE
C
      60 DO 80 I=1,38
           IF(I.GE.6)GOTO 70
           IC(I) = IT(I)
      70   IC(6) = ISAMT
           IF(I.LT.7)GOTO 80
           IC(I) = ISMUL(I-6)
      80 CONTINUE
         DO 90 I=1,10
           CMC(I)     = SAFCOD(I)
           CMC(I+10) = SAFCOD(I+10)
           CMC(I+20) = CLFLOD(I)
           CMC(I+30) = REMLTD(I)
           CMC(I+40) = CLIMED(I)
           CMC(I+50) = GASFAD(I)
           CMC(I+60) = GASFBD(I)
           CMC(I+70) = GASFCD(I)
           CMC(I+80) = FILCYD(I)
           CMC(I+90) = SERVOD(I)
           CMC(I+100) = SERVOD(I+10)
           CMC(I+110) = DUMMY(I)
           CMC(I+120) = DUMMY(I+10)
           CMC(I+130) = DUMMY(I+20)
           CMC(I+140) = DUMMY(I+30)
           CMC(I+150) = DUMMY(I+40)
      90 CONTINUE
C
         CALL WRITF(IDCB,IERR,ITOT,358)
C                  WRITE UPDATED COMMON INTO COMDAT FILE
         IF(IERR.LT.0)WRITE(LU,1600)COMDAT
C
         CALL CLOSE(IDCB)
C
C-----------------------------------------------------------------------------------
C  7. SUSPEND PACIR AND NOTIFY THE OPERATOR.
C
         ISMUL(1)=-1
C                        SUSPEND PACIR. NOTE THAT THIS CANNOT BE DONE WITH
C                        CALL EXEC(6,PACIR,2) BECAUSE PACIR IS A BASTARD AT
C                        THIS STAGE !! TRY IT IF YOU DONT BELIEVE ME!(ADH)
         WRITE(LU,1200)
         CALL PTAD(LU)
         CALL WAIT(10,2,IDUM)
C                        10 SECOND WAIT UNTIL SDATA SUSPENDED BEFORE RUNNING
C                        IT ONCE MORE TO CLOSE THE CURRENT FDATA FILE.
         CALL RNRQ(4,IRN(5),ISTAT)
C
C-----------------------------------------------------------------------------------
C  8. SUPPRESS RESTART MESSAGE AND STOP IF PACIR NOT RE-SCHEDULED.
C
         IF(ISTART.LE.0)GOTO 120
         WRITE(LU,1300)ISTART,IY,IZ
     100 CONTINUE
         GOTO 20
```

```
PAGE 0004  HANGO 10:24 AM  WED., 26  APR., 1978

C
C-----------------------------------------------------------------------------------
C  9. SCHEDULE PACIR & NOTIFY THE OPERATOR.
C
   110 CALL RCDSP
       WRITE(LU,1400)
       CALL PTAD(LU)
   120 STOP 0002
C
C-----------------------------------------------------------------------------------
C  10. FORMATS
C
  1000 FORMAT("TODAY IS DAY NUMBER ",I5,". ",/,
      1  "ENTER STOP/START TIMES NOW. (0=IMMEDIATE RESPONSE, -1=IGNORED)"
      2  ,//,"STOP TIME (DAY,HOUR,MINUTE)?")
  1100 FORMAT("RESTART TIME (DAY,HOUR,MINUTE)?")
  1200 FORMAT("PACIR SUSPENDED ON COMMAND")
  1300 FORMAT("SCHEDULED TO RE-START ON DAY ",I5," AT ",I2,"H",I2)
  1400 FORMAT("PACIR COMMENCING ON SCHEDULE")
  1500 FORMAT("FILE OPENING ERROR IN ""HANGO""-(",3A2,")")
  1600 FORMAT("FILE WRITING ERROR IN ""HANGO""-(",3A2,")")
C
C-----------------------------------------------------------------------------------
C
       END


FTN4 COMPILER: HP92060-16092 REV. 1726

**  NO WARNINGS **  NO ERRORS **   PROGRAM = 01672      COMMON = 00758
```

```
PAGE 0005  FTN.  10:24 AM  WED., 26  APR., 1978

      SUBROUTINE RCDSP
C-----------------------------------------------------------------------
C RCDSP - READ COMMON DATA AND SCHEDULE PACIR
C-----------------------------------------------------------------------
C
      INTEGER RCOMD(3),PACIR(3)
C
C                       ------ COMMON ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1    SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2    GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3    SERVOD(20),DUMMY(50),
     4    ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
     5    ISCOP(3),IDUMY(50)
C
C   ENG   - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C   ADCV  - A/D VOLTAGES (UPDATED BY SCAD)
C   CDACV - D/A VOLTAGES (UPDATED BY CDAC)
C
C   SAFCOD- SATURATOR FLOW CONTROL DATA
C   CLFLOD- CLOUDY LIQUOR FLOW DATA
C   REMLTD- REMELT CONTROL DATA
C   CLIMED- CONTROL LIME DATA
C   GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C   GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C   GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C   FILCYD- FILTER CYCLE MONITER DATA
C   SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C   ISAMT - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C   ISMUL - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C   IRN   - RESOURCE NUMBERS
C   ICIN  - CONTACT STATUS IN (UPDATED BY SCCS)
C   ICOUT - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C   ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C   ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C   ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C-----------------------------------------------------------------------
C
C
      DATA RCOMD/2HRC,2HOM,2HD /,PACIR/2HPA,2HCI,2HR /
C
C SCHEDULE RCOMD TO GET LAST SET OF COMMON DATA
C
10    CALL EXEC(23,RCOMD)
C                       QUEUE SCHEDULE WITH WAIT
C
20    CALL RNDTM(ISAMT,0,NSECS,NMIN,NHOUR)
C                       ROUND TIME UP TO NEXT HALF MINUTE OR WHATEVER
C
      ISMUL(1)=1
C                       SET FLAG FOR PACIR
30    CALL EXEC(12,PACIR,2,ISAMT,NHOUR,NMIN,NSECS,0)
```

```
PAGE 0006  RCDSP 10:24 AM  WED., 26  APR., 1978

 0246  C                    SET PACIR TO RUN EVERY ISAMT SECONDS
 0247        RETURN
 0248        END


 FTN4 COMPILER: HP92060-16092 REV. 1726

 **  NO WARNINGS **  NO ERRORS **   PROGRAM = 00048      COMMON = 00758
```

PAGE 0001 FTN. 9:29 AM MON., 20 FEB., 1978

```
FTN4,L
      PROGRAM RFLDT
C-----------------------------------------------------------------
C   RFLDT - READ FILE "FILDAT" ON DISC
C
C  THIS PROGRAM READS THE FILE FILDAT GENERATED BY FILCY AND LISTS
C  THE DATA ON THE PRINTER. IT THEN CALCULATES THE MEAN AND STANDARD
C  DEVIATION OF THE FOUR PARAMETERS :
C    "START INTS" =START INTERVALS
C   "VAR PR PERDS" =VARIABLE PRESSURE PERIODS
C   "CYCLE  PERDS" =TOTAL CYCLE PERIODS
C   "FILTRABILITY" =FILTERABILITY
C                 FOR EACH INDIVIDUAL FILTER AND LISTS THEM.
C
C    FINALLY THE OVER-ALL MEAN AND STANDARD DEVIATION OF EACH
C    PARAMETER FOR ALL FILTERS TAKEN TOGETHER ARE CALCULATED & LISTED.
C
C                   VERSION: 15-12-1977.
C-----------------------------------------------------------------
C
C
      INTEGER FILDAT(3),IDCB(144),IFCT(50,12),IBUF(600),IBLK(12)
      DIMENSION AV(12,4),SDV(12,4),TAV(4),TSDV(4)
      DATA FILDAT/2HFI,2HLD,2HAT/
C
C
      DO 50 N=1,4
      TAV(N)=0.
      TSDV(N)=0.
   50 CONTINUE
C
      CALL OPEN(IDCB,IERR,FILDAT,1,0,0)
      IF(IERR.GE.0)GOTO 100
C
      WRITE(1,1020)IERR
 1020 FORMAT("UNABLE TO OPEN FILE FILDAT - IERR=",I6)
      GOTO 2000
C
  100 WRITE(6,1040)
 1040 FORMAT(80("*"),//,28X,"FILTER DATA FILE",//,80("*"),/)
  150 CALL READF(IDCB,IERR,IBUF,600,LEN)
      IF(IERR.GE.0)GOTO 200
C
      WRITE(1,1030)IERR
 1030 FORMAT("UNABLE TO READ FROM FILE FILDAT - IERR=",I6)
      GOTO 2000
C
  200 IF(LEN.EQ.-1)GOTO 2000
C
      DO 500 J=1,50
      DO 500 I=1,12
      K=(J-1)*12+I
      IFCT(J,I)=IBUF(K)
  500 CONTINUE
      IM = IFCT(1,12)
```

PAGE 0002 RFLDT 9:29 AM MON., 20 FEB., 1978

```
      IJ = 1+IM
C
      WRITE(6,1050)(IFCT(1,I),I=5,2,-1)
 1050 FORMAT(5X,"DAY   ",I3,4X,I2,"H",I2,":",I2,3X,":-",/)
      WRITE(6,1060)
 1060 FORMAT(15X,"******STORED OPERATING DATA FOR EACH FILTER******",//)
      WRITE(6,1070)(I,I=1,12)
 1070 FORMAT(33X,"FILTER NUMBER.",/,4X,12I6,/)
      WRITE(6,1000)((IFCT(J,K),K=1,12),J=2+4*IM,49)
 1000 FORMAT(12(4(4X,12I6,/),/),//)
      WRITE(6,1190)
 1190 FORMAT(15X,"*****STATISTICS FOR FILTER STATION OPERATION*****",//,
     1  31X,"INDIVIDUAL FILTERS",/)
      WRITE(6,1080)
 1080 FORMAT(22X,"AVERAGES",14X,"*",9X,"STANDARD DEVIATIONS.",/)
      WRITE(6,1200)
 1200 FORMAT("FILTER",2X,"FILTER",2X,"VARIABLE",3X
     1,"CYCLE",2X,"FILTER-",6X,"FILTER",2X,"VARIABLE",3X
     2,"CYCLE",2X,"FILTER-",/,"NUMBER",2X,"STARTS",2X,"PRESSURE",3X,
     3 "TIMES",2X,"ABILITY",6X,"STARTS",2X,"PRESSURE",3X,"TIMES",2X,
     4 "ABILITY",/,8X,"(MINS)",3X,"(MINS)",3X,"(MINS)",4X,"(%)",8X,
     5 "(MINS)",3X,"(MINS)",3X,"(MINS)",4X,"(%)",/)
      NUM=0
      DO 250 I=1,12
      IBLK(I)=IFCT(50,I)-IM
      IF(IBLK(I).LT.0)IBLK(I)=0
      NUM=NUM+IBLK(I)
  250 CONTINUE
      IF(NUM.LT.1)NUM = 1
      DO 400 N=1,4
      TAV(N) = 0.
      DO 350 I=1,12
      SUM=0.
      DO 300 J=IJ,IBLK(I)+IM
      L=4*J+N-3
      SUM=SUM+IFCT(L,I)
      TAV(N)=TAV(N)+IFCT(L,I)
  300 CONTINUE
      IF(IBLK(I).LT.1)IBLK(I)=1
      AV(I,N)=SUM/FLOAT(IBLK(I))
  350 CONTINUE
      TAV(N)=TAV(N)/FLOAT(NUM)
  400 CONTINUE
C
C
      DO 700 N=1,4
        TSDV(N) = 0.
      DO 650 I=1,12
      SUM=0.
      DO 600 J=IJ,IBLK(I)+IM
      L=4*J+N-3
      DUM1=IFCT(L,I)-AV(I,N)
      DUM2=IFCT(L,I)-TAV(N)
      SUM=SUM+DUM1*DUM1
      TSDV(N)=TSDV(N)+DUM2*DUM2
```

```
PAGE 0003  RFLDT  9:29 AM  MON., 20  FEB., 1978

   600 CONTINUE
       IF(IBLK(I).GE.2)GOTO 610
       SDV(I,N) = 0.
       GOTO 650
   610 ARG = SUM/FLOAT(IBLK(I)-1)
       SDV(I,N)=SQRT(ARG)
   650 CONTINUE
       IF(NUM.GE.2)GOTO 660
       TSDV(N) = 0.
       GOTO 700
   660 ARG = TSDV(N)/FLOAT(NUM-1)
       TSDV(N)=SQRT(ARG)
   700 CONTINUE
       DO 550 I=1,12
       WRITE(6,1090)I,(AV(I,N),N=1,4),(SDV(I,M),M=1,4)
  1090 FORMAT(I4,2F10.1,2F8.1,4X,2F10.1,2F8.1)
   550 CONTINUE
C
       WRITE(6,1180)
  1180 FORMAT(//,32X,"OVERALL RESULTS.",/,22X,"AVERAGES",21X,
      1  "STANDARD DEVIATIONS",/)
       WRITE(6,1220)TAV(1),TSDV(1)
  1220 FORMAT("STARTS",14X,F8.1,27X,F8.1)
       WRITE(6,1230)TAV(2),TSDV(2)
  1230 FORMAT("VARIABLE PRESSURE",3X,F8.1,27X,F8.1)
       WRITE(6,1240)TAV(3),TSDV(3)
  1240 FORMAT("CYCLE TIMES",9X,F8.1,27X,F8.1)
       WRITE(6,1250)TAV(4),TSDV(4)
  1250 FORMAT("FILTERABILITY",7X,F8.1,27X,F8.1)
       WRITE(6,1260)
  1260 FORMAT(/,80("*"),//)
       GOTO 150
  2000 CONTINUE
       CALL CLOSE(IDCB)
       END


FTN4 COMPILER: HP92060-16092 REV. 1726

**  NO WARNINGS **  NO ERRORS **   PROGRAM = 02895       COMMON = 00000
```

PAGE 0001 FTN. 9:20 AM MON., 20 FEB., 1978

```
FTN4,L
      PROGRAM CLOOP
C***********************************************************************
C
C         CLOOP - MONITORS THE CHANGES IN STATUS OF THE CONTROL LOOP
C                OPERATOR'S SWITCHES. IF CONTROL PROGRAMS ARE
C          RUNNING THE SWITCHES SHOULD ONLY BE 'OFF'-ED IN AN EMERGENCY.
C          SUCH AN OCCURENCE IS LOGGED AND STORED IN THE DISC FILE
C         "SMAUTS" (STATUS OF MANUAL/AUTOMATIC SWITCHES) FOR RETRIEVAL
C         IN THE PROCESS MANAGER'S REPORT.
C
C       THE DATA IS STORED IN A 5-WORD ARRAY WHERE :
C             IBUF(1)=SWITCH NUMBER(1=MASTER OVER-RIDE SWITCH)
C                                  (2=SAFCO LOCAL/COMPUTER SWITCH)
C                                  (3=CLFLO LOCAL/COMPUTER SWITCH)
C                                  (4=REMLT LOCAL/COMPUTER SWITCH)
C                                  (5=CLIME LOCAL/COMPUTER SWITCH)
C                                  (6=GASFA LOCAL/COMPUTER SWITCH)
C                                  (7=GASFB LOCAL/COMPUTER SWITCH)
C                                  (8=GASFC LOCAL/COMPUTER SWITCH)
C                                  (9-15 = BLANK)
C             IBUF(2)=CURRENT STATUS.(0=ON LOCAL)
C                                    (1=ON COMPUTER)
C             IBUF(3-5)=DAY,HOUR,MIN AT TIME OF SWITCH.
C
C          ERROR MESSAGES :
C
C                  1 : MASTER CONTROL SWITCH TO LOCAL MODE.
C                  2 : MASTER CONTROL SWITCH TO COMPU MODE.
C                  3 : SATURATOR FLOW ON LOCAL CONTROL
C                  4 : SATURATOR FLOW ON COMPU CONTROL
C                  5 : CLOUDY LQUOR FLOW ON LOCAL CONTROL
C                  6 : CLOUDY LQUOR FLOW ON COMPU CONTROL
C                  7 : REMELT FLOW ON LOCAL CONTROL
C                  8 : REMELT FLOW ON COMPU CONTROL
C                  9 : A-SAT GAS FLOW ON LOCAL CONTROL
C                 10 : A-SAT GAS FLOW ON COMPU CONTROL
C                 11 : B-SAT GAS FLOW ON LOCAL CONTROL
C                 12 : B-SAT GAS FLOW ON COMPU CONTROL
C                 13 : C-SAT GAS FLOW ON LOCAL CONTROL
C                 14 : C-SAT GAS FLOW ON COMPU CONTROL
C                 15 : LIME ADDITION RATE ON LOCAL CONTROL
C                 16 : LIME ADDITION RATE ON COMPU CONTROL
C
C                        VERSION : 9-11-1977.
C***********************************************************************
C
      INTEGER IDCB(144),IBUF(5),IT(5),IP(5),SMAUTS(3)
C                 ------ COMMON ------
C
      COMMON ENG(64),ADCV(64),CDACV(24),
     1  SAFCOD(20),CLFLOD(10),REMLTD(10),CLIMED(10),
     2  GASFAD(10),GASFBD(10),GASFCD(10),FILCYD(10),
     3  SERVOD(20),DUMMY(50),
     4  ISAMT,ISMUL(32),IRN(40),ICIN(4),ICOUT(4),
```

PAGE 0002 CLOOP 9:20 AM MON., 20 FEB., 1978

```
     5   ISCOP(3),IDUMY(50)
C
C     ENG    - ENGINEERING UNITS (CALCULATED BY ENGUN FROM ADCV VOLTAGES)
C     ADCV   - A/D VOLTAGES (UPDATED BY SCAD)
C     CDACV  - D/A VOLTAGES (UPDATED BY CDAC)
C
C     SAFCOD- SATURATOR FLOW CONTROL DATA
C     CLFLOD- CLOUDY LIQUOR FLOW DATA
C     REMLTD- REMELT CONTROL DATA
C     CLIMED- CONTROL LIME DATA
C     GASFAD- GAS FLOW CONTROL DATA FOR "A" SATURATOR
C     GASFBD- GAS FLOW CONTROL DATA FOR "B" SATURATOR
C     GASFCD- GAS FLOW CONTROL DATA FOR "C" SATURATOR
C     FILCYD- FILTER CYCLE MONITER DATA
C     SERVOD- SERVOBALANS SCALE MONITOR DATA
C
C     ISAMT  - MASTER SAMPLING RATE (PACER FREQUENCY, SECS)
C     ISMUL  - SUB-RATE SAMPLING TIMES (PERIOD(X)=ISAMT*ISMUL(X))
C     IRN    - RESOURCE NUMBERS
C     ICIN   - CONTACT STATUS IN (UPDATED BY SCCS)
C     ICOUT  - CONTACT STATUS WORDS UPDATED BY CONTROL PROGRAMMES.
C     ISCOP(1)- FLAG USED BY WCHDG AND THE CONTROL PROGRAMMES.
C     ISCOP(2)- STATUS OF CONTROL PROGRAMMES.(I.E. RUNNING OR OFF)
C     ISCOP(3)- STATUS OF AUTO/MANUAL SWITCHES.
C
C--------------------------------------------------------------------------------
C
C
C
      DATA SMAUTS/2HSM,2HAU,2HTS/
C
C      PICK UP PARAMETERS FROM CALLING PROGRAM (SCCS) :
C
         CALL RMPAR(IP)
           JCNTO=IP(1)
           JCNTN=IP(2)
C
      CALL OCEND(IDCB,SMAUTS,IERR)
C       OPEN FILE SMAUTS AND STEP TO END
C
      ISTAT=0
C
      DO 200 I=1,15
C
      IF((I.EQ.15).AND.(ISTAT.EQ.1))GOTO 100
      IF(I.EQ.1)GOTO 100
      J = I-1
        J=IBIT(J,ISCOP(2))
        IF(J.EQ.0)GOTO 200
C
           ISTAT=1
C         FLAG THAT AT LEAST ONE CONTROL PROGRAM IS RUNNING.
C
  100     K=IBIT(I,JCNTN)
          L=IBIT(I,JCNTO)
```

```
PAGE 0003  CLOOP  9:20 AM  MON., 20  FEB., 1978

          IF((K.EQ.0).AND.(L.EQ.0))GOTO 200
          IF((K.EQ.1).AND.(L.EQ.1))GOTO 200
          J=2*I-(1-K)
          CALL MESAG(-J,I)
                    CALL EXEC(11,IT,IY)
                    IBUF(1)=I
                    IBUF(2)=K
                    IBUF(3)=IT(5)
                    IBUF(4)=IT(4)
                    IBUF(5)=IT(3)
    C
                    CALL WRITF(IDCB,IERR,IBUF,5)
    C
      200 CONTINUE
    C
          CALL CLOSE(IDCB)
    C
          END


FTN4 COMPILER: HP92060-16092 REV. 1726

**  NO WARNINGS **  NO ERRORS **   PROGRAM = 00359      COMMON = 00758
```