QUAESTIONES INFORMATICAE

Vol. 1 No. 1

June, 1979

Quaestiones Informaticae

An official publication of the Computer Society of South Africa
'n Amptelike tydskrif van die Rekenaarvereeniging van Suid-Afrika

Editors: Dr. D. S. Henderson,
Vice Chancellor, Rhodes University, Grahamstown, 6140, South Africa.
Prof. M. H. Williams,
Department of Computer Science and Applied Maths,
Rhodes University, Grahamstown, 6140, South Africa.

Editorial Advisory Board

Subscriptions

Annual subscriptions are as follows:

	SA	US	UK
Individuals	R2	$3	£1.50
Institutions	R4	$6	£3.00

Quaestiones Informaticae is prepared for publication by SYSTEMS PUBLISHERS (PTY) LTD for the Computer Society of South Africa.

Real-Time Interactive Multiprogramming

A. D. Heher

NEERI, CSIR, Pretoria, South Africa*

Abstract

This paper is concerned with the problems of interactive computing when multiple real-time tasks are involved. The interactive facilities which can be provided are described and some of the important software engineering concepts which should be incorporated in modern programming systems are discussed. Memory management is a crucial activity in interactive systems and a new memory management technique called Software Virtual Memory Management (SVMM) has been developed to allow the interactive facilities of monoprogrammed systems to be extended and improved in a multiprogramming environment. This management technique has been incorporated in an interpreter-oriented system called VIPER (Virtual Interactive Process Executive for Real-time control) and experience with this system is reported here. The paper concludes with a brief analysis of the performance of VIPER which shows that the SVMM technique can be used to improve the performance of interactive real-time systems constructed with the aid of an interpreter to the extent where they can be competitive with compiler-oriented systems in certain applications.

1. Introduction

In terms of both processing power and ease of use, a significant gap exists in real-time operating systems between stand-alone BASIC-type systems and disc-based compiler-oriented executives. There are many applications where it is desirable to utilize the interactive facilities of a language like BASIC while retaining the multiprogramming capabilities of the compiler-based executives. This paper discusses the problem of constructing such a real-time interactive multiprogramming system and describes a system which has been implemented to demonstrate the concepts involved.

The objective of bridging this gap is to permit more complex programming tasks to be undertaken by application-oriented staff by providing them with simple but powerful software tools. The success of this approach has been reported by a number of workers [2-5, 7-8, 19-22, 24]. Despite their limitations and language defects, interactive systems like BASIC, supplemented by a range of high-level real time functions, have proved useful in a variety of applications. To encourage the programming task to be undertaken by those who understand the problem, the task of programming should be simplified in every possible way. This permits users to get into the task 'without undue effort spent in learning computerese' [3], and also 'to improve software reliability by reducing the opportunity for error' [7].

The simplicity and security of the software system which is used are important factors in the production of reliable software [16], particularly if more complex multiprogramming tasks are to be undertaken. In constructing a system which supports multiple concurrent tasks, it is not sufficient to merely extend the facilities of monoprogrammed systems; radically different software techniques must be applied in order to maintain and improve upon the reliability achieved by these simple systems.

There are five aspects to which particular attention must be paid in real-time systems, viz

1. Data structure and access.
2. Protection of code and data areas.
3. Synchronization and control of concurrency.
4. Structured programming.
5. Program and system documentation.

Before dealing with these aspects in Section 3, three introductory topics are discussed in Section 2. The discussion in this paper is strongly related to an interactive system which has been constructed. An overview of this system is therefore given first (2.1) to enable the discussion which follows to be placed in context, followed by a discussion of memory management in interactive systems (2.2). The third topic deals with the concept of interaction itself; what is meant by interactive and the interactive facilities which can, and should, be provided (2.3).

2.1 An Overview of Viper

VIPER is an interpretive system which evolved from an earlier monoprogrammed real-time BASIC called PROSIC [9, 10]. PROSIC in turn was a development from the original Varian BASIC. [6]. VIPER is coded in Varian Assembler and, like BASIC, is a stand-alone system containing all its own operating system functions. It has been run on a Varian 620 and on a microprogrammable microprocessor system called MIKROV [26]. The memory-resident portion of VIPER is 13 K words (16 bit) in size.

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 which is similar to a stand-alone BASIC program in many respects. The procedures (= code segments) are created and manipulated interactively from an input device.M ore 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. Fig. 1 illustrates some of these interactive operations. Figs. 2, 3, 4 and 5 illustrate other types of VIPER statements which are described in the sections that follow.

The segments of code are moved to and from a bulk storage device under control of a memory manager. At any given time some, or all, of the modules of a task may be swapped out of memory to make room for other modules of the same, or some other, task. If a module which is not in local memory is referenced, directly or indirectly, by any command or statement, it is swapped-in under control of the executive. The logical space required by a set of modules can therefore be larger than the physical space. This gives VIPER virtual-memory properties.

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 protection 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. One of the most important properties of VIPER is that interactive operations, including the execu-

This paper was presented at the SACAC Symposium on Real-time Software for Industrial Applications in Pretoria on 29-30 November, 1978.

*The author is now with AECI Limited in the Consulting Engineers Department.

tion of commands and the addition of statements, can continue while a procedure is executing. Operations of this type are illustrated in Fig. 1. The provision of interactive debugging operations on executing real-time tasks is not merely a convenient feature — it is a powerful tool for the testing and debugging of real-time software. Hoare [16] has noted that this commissioning of software can be "the most tiresome, expensive and unpredictable phase" 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.

Shared data areas are an important resource in a real-time environment and require protection from inadvertent or illegal modification if the system is to be secure. Shared data segments are provided in VIPER which are referenced and defined in a manner analogous to that of named COMMON blocks 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. Figs. 2 and 3 illustrate some of the commands and statements available for the manipulation of data elements. Shared and local data facilities are described in more detail in Section 3.1.

2.2 Memory Management

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 recompiled 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 [9, 10], which it would be desirable to extend to the multi-tasking environment.

A number of interactive systems with either multiprogramming or multi-user capabilities have been constructed [4, 19, 23, 29]. However, they all suffer from the following shortcomings:

1. A lack of independent named procedures and subroutines which are essential for a structured programming approach.
2. Poor shared data facilities and a lack of protection for any of the facilities provided.
3. Restricted interactive facilities. None of these systems, nor any system known to the author, permits interactive operations to be applied to 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 multiprogrammed system the difficulty occurs in the attempt to allow multiple tasks or procedures to undergo this dynamic change in size and structure simultaneously. The addition of a multi-user capability further complicates the memory management task, as does the need for flexible access to shared data areas.

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 virtual memory management, and for this reason the technique used has been called 'Software Virtual Memory Management' (SVMM). As a result of the software implementation, a resident operating system nucleus must exist in a fixed memory partition and only the remaining memory is available for virtual storage operations.

The term 'virtual memory' has two connotations in the context of this paper: the first is related to the usual concept of addressing a logical space which is larger than the physical space; and 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 memory management structures have been described elsewhere [13] and as they are fairly complex, space precludes a discussion of their operation in this paper.

2.3 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 second 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 paper. Even the term "interactive program development" is not well-defined — it is used by some authors to refer to computing services of the time-sharing type and by others to incremental compilation and direct execution, such as is possible with BASIC. 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-tape 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 paper.

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 programs.
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 a statement to (or delete it from) 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 by means of the same command and programming language 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.

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 permits software modules to be tested as they are written, or as soon afterwards as possible and allows 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 needs to be stopped or taken off-line before 'test' or 'debugging' functions can be included, the commissioning task is made considerably more difficult and timeconsuming.

Wilkes [23] 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."

3.1 Data Structure and Access

The organization and protection of the data elements in a programming system are recognised to be crucial factors in the development of reliable software [15]. In VIPER there are three main categories of data types, viz local variables, shared data, and variables passed as parameters in a procedure call. Protection facilities are applied to all three categories and they are all accessed by means of similar mechanisms. The dominant characteristic of the data element is that most of them can be created dynamically.

The links between segments which are required to reference items in another segment are established dynamically on the first reference to the variable (incurring some overhead), but , once established, the links are maintained until released by either the user or the system. After establishing the links, subsequent references are therefore performed efficiently with a minimal overhead.

1. Local variables: The only local variable types provided so far in VIPER are simple and array floating-point variables. Provision has, however, been made for additional types including string, bit and integer variables and these could be included within the SVMM framework without difficulty. Local array variables can be defined dynamically or statically. Within a subprocedure, for example, a variable size area can even be allocated in accordance with a passed parameter and then de-allocated (released) before exiting from the procedure. Fig. 3 shows a schematic outline of this type of operation. Read or write access to an array can also be specified.

2. Shared data elements: Shared data elements are grouped together in named segments which can be manipulated by the SVMM routines. The syntactic structures for creating and referencing these elements are similar to those used for named COMMON blocks in FORTRAN, as illustrated in Fig. 2. Both simple and array variable elements within a particular segment can be individually read- or write-protected. A segment can also be password-protected to prohibit access to all but password holders. Data segments could be swapped like procedure segments, but this has not been implemented in VIPER. They can, however, be moved to and from bulk storage devices under user control. All data segments also have a semaphore associated with them which can be used for synchronization, as discussed in Section 3.3.

3. Parameter passing: Parameters can be passed from one procedure to another using a normal FORTRAN-like CALL-SUB-RETURN sequence. The invoked subprocedure need not be contiguous to or owned by the calling procedure, however, and may not even be resident in memory when it is invoked. Parameter types are matched and must agree and default access states are established where appropriate. If a constant appears in the actual parameter list, for example, the corresponding formal parameter state is set to read-only. Specific access states of actuals are also copied through to the formulas. The detection of illegal mappings (mismatched types) is performed at the CALL-SUB set-up time while access violations are checked on each reference to a formal parameter. 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 the programming. The SVMM structures minimise this overhead and operations involving formal parameters take only 4% longer than the equivalent operation using local variables only. More dramatic, is a factor of four improvement in VIPER compared with PROSIC, despite the extra protection functions.

3.2 Protection

The types of protection facilities which can be applied to data elements in a real-time system were discussed in the preceding section. Of equal inportance in an interactive system is the protection of code segments. Protection functions are provided in many operating systems but these often apply only to bulk storage resident files. In VIPER, the protection facilities 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.

The basic mechanism for access protection is a capability list which is associated with each segment; the primary function of this is to control a set of well-defined states in which a procedure can operate. These states provide protection for the user against himself as well as against 'pirates'. A password is used to control the commands which can be used to manipulate (indirectly) entries in these capability lists. Before any input is accepted from a user at a keyboard he must LOGON with an appropriate password.

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 the associated data elements which control a particular section of a plant, for example, can 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, in that no modifications can be made to any of the procedures if the appropriate password is not specified. (It may still be possible to examine and execute the procedures if this has been permitted by their 'owner'.)

The control of access to code segments illustrates the type of functions which can be provided by means of these protection facilities. A procedure can exist in one of four primary states, together with several sub-states.

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

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. Statement execution frequency counts can also be invoked in this mode.

Monitor: This mode permits 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 f a procedure and this mode can therefore be made freely available to process staff. A substate of the mode can be invoked, however, to limit access to password holders.

Execute: This mode has three substates; free execute; password execute; and no access. The latter category enables the execution of a procedure to be prohibited until one of the other substates is established with a specific command.

The modes CHANGE, DEBUG and MONITOR are entered with commands of the form CHANGE ‹procedure name› while substates are specified by commands with the form ACCESS (‹procedure name›) = ‹attribute› . Only the appropriate passholders can enter the CHANGE and DEBUG modes or change the access attributes.

3.3 Synchronization

The semaphore 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:

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

For these reasons other language constructs, such as the "REGION" construct [1] and the "MONITOR" [14] have been proposed. Hoare's monitor concept has been noted to be the most general and secure structure, but it would appear to be more suitable for operating system structure 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 so extensively used 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 semaphore associated with the data area and can only proceed to execute the critical region statements if the semaphore is not already locked. If the semaphore is locked, the procedure is suspended and waits for the semaphore to be cleared (unlocked) by an END REGION statement. In addition to the REGION construct, primitive operations are provided for the direct manipulation of semaphores. These simple but powerful facilities assist in the modular decomposition of tasks into separate and independent sub-tasks which are safer and which are much simpler to code and debug.

3.4 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."

M.V. WILKES [28]

The term "structured programming" has acquired a variety of meanings, but Wilkes' concise statement captures the essential properties of this programming discipline. The development of structured programming techniques is a current topic of research and a wide variety of control structures have been proposed and discussed. Because of this fluidity only the simplest and most widely used structures were used in VIPER. There are two aspects of structured programming which are of particular relevance to VIPER; viz. modularity and structures.

3.4.1. Modularity

In VIPER each named code module, which may be either a procedure or a 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" that 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.

One of the recommended practices associated with the art of structured programming is the independent testing of the individual modules of a task as they are written. Some sophisticated software tools have been developed for this type of operation, particularly for cases where top-down design or stepwise refinement strategies are used. VIPER makes no specific provision for this design procedure but the ease with which modules can be tested individually, together with the flexible data structures which simplify the generation and linking of test data, enables this practice to be carried out with the aid of 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 in any event) 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.

3.4.2 Structures

The control structures incorporated in VIPER are as follows:
IF - THEN - ELSE -ENDIF
FOR - NEXT
DO WHILE - END DO
CASE - ENDCASE
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 Fig. 5. Despite the simplicity of the structures they have a markedly beneficial

effect on both the clarity and the ease of understanding of the programs.

The simple GOTO was retained in VIPER as it has quite clearly been shown [18] that it is sometimes required, even in well-structured programs, to avoid awkward and clumsy constructions. An interesting observation arose, however, from the case study discussed in section 4. 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.

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. An example illustrating this facility is given in Fig. 5. 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 [16], so this problem is not just a development phenomenon. 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 unfamiliar 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. 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 intended to distinguish it from the body of the program. Examples of this facility can be seen in Figs. 3 and 5.

3.5 Documentation

Interactive systems like BASIC or VIPER are frequently used for experimental or investigatory work, an environment where the maintenance of good documentation is as difficult as it is important. Interpretive systems also frequently suffer from the disadvantage of not being able to use source text layout to improve program visibility and understanding because of the back-listing or decompilation this is performed to recreate text. Special effort must therefore be made to assist and encourage the documentation of interpretive programs. Both program and system documentation functions are important, as discussed below.

3.5.1. Language structure

The most important factor in the production of clear, well-documented software is the language structure itself. No quantity of comments can overcome basic defects in the language. There are four important properties of language structure, viz:

Structured language. This 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 3.4 and an example of the VIPER facilities is given in Fig. 5. There is a strong case for all interpretive systems to use a structured language, for the sake of documentation, if nothing else.

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 quite unnecessary, as an extension of PROSIC has shown [10]. In VIPER, all names, including variables, data areas and procedure names, can be up to 16 characters in length. These longer names are an invaluable aid to clear documentation and reduce the need for trivial comments to explain the meaning of variables.

Syntactical structures. The syntax of the language and operating system commands can contribute to the readability of programs. The structured programming keywords such as CASE, DOWHILE and the synchronization function REGION are examples of syntactical structures with clear and unambiguous meanings. Other functions of importance are operations such as RUN and WAIT. Some examples illustrating the syntax of these statements in VIPER are given in Fig. 4. The action which is required is immediately apparent without reference to manuals to determine the meaning of obscure parameters.

Conciseness. FORTRAN and 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.) One simple but successful aid is to allow multiple assignments on a line. This is effective because simple assignments are the most common statements in typical programs [17]. 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.

3.5.2 Comment facilities

All languages make provision for comments in one form or another, but the actual syntactical forms used are of crucial importance [16, 25] The ease with which comments can be inserted, and their readability once inserted, are important factors in 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. In VIPER end-of-line comments are right justified in the back-listing operation, as shown in Fig. 5. This achieves the required visibility while maintaining conciseness.

3.5.3. System documentation

Typical real-time programming tasks are made up 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, and to know which modules call others (the hierarchical relationship) and which modules access particular data areas. The relationships 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 documentation systems which produce the required information from an off-line analysis of the source program listing have been reported on in the literature. (It is assumed that the only precise and up-to-date source of internal documentation for most software is the programs themselves.) However, the programs are typically large (10 000 — 15 000 statements) which illustrates the complexity of producing the information from source listings. In interpretive systems, and in the SVMM structures of VIPER in particular, the information is readily

available within the various symbol tables, and system documentation information on the overall structure of the task can easily be produced on-line. This is a particular advantage in an interactive system in that the information that is produced represents the actual state of the system at that time. Dynamic information on structures that vary with time can also be produced.

4. Performance

Interpretive systems have a reputation for poor performance which is not always entirely justified, as the data below shows. (This reputation would appear to have arisen, at least in part, from the notoriety of some early BASICs which interpreted source code directly.) Data published by a number of authors indicate that in typical applications, interpretive tasks are approximately five times slower than the equivalent compiled tasks. Similar results have been obtained from comparisons between VIPER and FORTRAN programs. This direct comparison does not always give the complete picture, however, as illustrated by the case study below. Before these results are discussed, the relationship between VIPER and other interpretive systems is briefly mentioned to place VIPER in perspective.

1. Comparison with interpreters. A variety of simple benchmark programs have been run to compare the performance of VIPER with that of its monoprogrammed predecessor PROSIC, as well as with that of a number of other BASIC or BASIC-like systems (10, 11). From the measurements that have been made it has been observed that the performance of VIPER is similar to that of both PROSIC and other BASICs when considering small benchmarks (5-20 statements). In larger programs (100-200 statements) the performance of VIPER is better than that of PROSIC by a factor of two or three and in the 900-1 000 statement category VIPER can achieve a performance improvement of up to 10 to 1. This effect and results of one particular benchmark are tabulated in Tables 1 and 2. This large program superiority of VIPER results directly from the modular SVMM structures and their efficient linking mechanisms. They avoid the increase in execution time in accordance with program size, which is a characteristic of many BASIC systems.

2. Comparison with compiled code (a case study). A FORTRAN-based process control package had been developed for an experimental process control investigation. This had ddc and supervisory control modules in addition to scanning, logging and alarm functions,. The system ran on a Hewlett Packard 21MX computer under control of the RTE II Executive in 32K words of memory. In the configuration used, 19 primary real-time tasks (15 of which ran cyclically) executed in a single foreground partition, resulting in numerous disc swapping operations. Semaphores were used extensively for synchronization and to control access to shared data which consisted of a single (unprotected) global common area. The semaphores facilitated the modular decomposition of tasks, but contributed to the high swapping rate. The FORTRAN programs were recoded into VIPER, resulting in 35 SVMM segments (27 code and 8 data), 16 of which execute repetitively. Table 3 summarises the statistics of the FORTRAN and VIPER programs. Two results of this case study are of interest:

Memory requirements. The VIPER programs require approximately one third of the memory space required by the FORTRAN programs and all the repetively executed modules can be memory-resident in a 32K word machine.

Execution time. Averaged over 60s, the repetitive tasks consume 1, 2 s of CPU time in the FORTRAN system versus 7,9 s in VIPER; a ratio of 6,6 to 1. When the time spent swapping is taken into account, however, the real-time foreground partition of the RTE system is busy for 9,0 s in 60 s compared with the 7,9 s in VIPER. In terms of real-time task throughput, the interpretive VIPER system is therefore slightly faster than its FORTRAN-based counterpart.

Ignoring all differences between the two computers used the results indicate that VIPER, running on a microprogrammed microprocessor emulator [26] 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 Hewlett-Packard RTE system could also support concurrent tasks in the resident and background partitions and it is not claimed that VIPER is equivalent to a system like the HP RTE in computational power. What is claimed, is that given a set of real-time tasks such as those encountered in the case study, an interpretive system can have much the same performance as a compiler oriented system and could be used in many applications where much larger and more complex operating systems had to be used previously.

5. Conclusion

It has been shown that interpretive systems are capable of significantly better performance when the techniques of software virtual memory management are employed. As the structures required can be implemented entirely in software, the system is machine-independent and can be implemented on any average mini- or microcomputer. The software system which has been constructed has a uniform command and programming language, and this simplifies operation and enables outstanding debugging facilities to be provided in an interactive, multi-user mode of operation. Incorporated into the system are excellent protection mechanisms which are easy to use but permit a number of users co-operatively to share a secure data base in a real-time environment. Despite the ease of use and the extensive protection mechanisms, the performance of the system is better than that of similar interpreter-based real-time systems and is competitive with compiler-oriented systems in some applications.

Acknowledgements

The support by the National Electrical Engineering Research Institute of the work reported in this paper is gratefully acknowledged. I would also like to express my thanks to my supervisor, Prof. H. L. Nattrass, of the University of Natal, for his guidance and advice. Many members of the NEERI staff also assisted with the work and I would in particular like to note the efforts of our software librarian Mrs. N. D. Thomson for many months of patient text editing, Mr. H. Frommann for work on the Varian Cross Assembler, Mr. J.P. Los for constructing the MIKROV computer, Mr. P. Grift for maintaining (and improving) our old Varians and Mr. P. Hussey for writing many of the FORTRAN programs in the case study and for translating some of these programs into VIPER.

References

1. BRINCH HANSEN P. (1973). *Operating System Principles,* Prentice Hall, Englewood Clifs.
2. CLAGGETT, E. (1977). Interpreters versus compilers for on-line microcomputers, *Control Engineering,* August 1977, 24-27.
3. DIEHL, W. (1976). Software for industrial computer control — a review, *1st IFAC/IFIP symposium on software for computer control,* Tallin, USSR, 279-285.
4. DIEHL, W. and SANDERS, D. (1975). Real-time BASIC used in a distributed network, *IFAC/IFIP workshop on real-time programming,* Boston, 159-163.
5. GAINES, B.R. and FACEY, P.V. (1975). Some experience in interactive system development and application, *Proc IEEE,* **63,** 894-911.
6. GOUWS, J. H. (1973). *User's notes on the process BASIC interpreter,* Internal report Ea-41, National Electrical Engineering Research Institute, Pretoria.
7. GRIEM, P. D. (1975). On the principle of unique definition, in *Proc AFIPS NCC,* **44,** Anaheim, 265 - 270.
8. HAASE, V. M. (1976). Evaluation of BASIC as a programming language for real-time systems, *IFAC/IFIP workshop on real-time programming,* Roquencourt, 331-339.
9. HEHER, A.D. (1976). Some features of a real-time BASIC executive, *Software Practice and Experience,* **6,** 387-391.
10. HEHER, A.D. (1976). PROSIC: A real-time BASIC executive, in *Proceedings of SACAC symposium on minicomputers and microprocessors,* Durban, 56-65.
11. HEHER, A.D. (1977). A real-time interactive software system for process modelling, optimization and control, *5th IFAC/IFIP digital computer applications to process control.* The Hague, 647-653.
12. HEHER, A.D. (1977). Refinery Computer Control Project Software organization and CAMAC drivers, CSIR Special Report ELEK 130, CSIR, Pretoria.
13. HEHER, A.D. (1978). Software virtual memory management in a real-time interactive software system, to appear in *IEEE Trans. on Software Engineering.*
14. HOARE, C.A.R. (1974). MONITORS: An operating system structuring system, *Comm ACM,* **17** *549-557.*
15. HOARE, C.A.R. (1975). Data reliability, *Proceedings of IEEE international conference on reliable software,* New York, 528-533.
16. HOARE, C.A.R. (1975). 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, 505 - 534.
17. KNUTH, D.E. (1971). An empirical study of FORTRAN programs, *Software Practice and Experience,* **1,** 105-133.
18. KNUTH, D.E. (1974). Structured programming with GOTO statements, *ACM Computing Surveys,* **6,** 261-301.
19. KOPETZ, H. and CRIGHTON, E.R. (1976). Ergonomics and security of real-time programming, *IFAC/IFIP Workshop on real-time programming,* Roquencourt, 331-339.
20. LARSON, S.L.G. (1974). Adapting BASIC for process control — meeting the needs of a conversational control system, *Advances in Instrumentation,* **29,** 512-517.
21. LAURANCE, N. (1975). Structured programming in a real-time environment, *IFAC/IFIP workshop on real-time programming,* Boston, 49-58.
22. MAPLES, M.D. and FISCHER, E.R. (1977). High-level language applications at Lawrence Livermore Laboratory, Univ. of California, U.S.A., *Microprocessors,* **1,** 312-316.
23. PERSEUS COMPUTING AND AUTOMATION (1976). MULTEX-BASIC manual WU-000168-01.
24. SCHOEFFLER J.D. and KEYES M.A. (1976). Hierarchical language processing, *1st IFAC/IFIP symposium on software for computer control,* Tallin, USSR, 263-272.
25. SCOWEN, R.S. and WICHMANN, B.A. (1974). The definition of comments in programming languages, *Software Practice and Experience,* **4,** 181-188.
26. VAN AARD, J.L. (1977). Microprocessor emulation of a Varian minicomputer, report NIDR 77/03, CSIR, Pretoria.
27. VAN MEURS, J. and CARDOZO, E.L. (1977). Interfacing the user, *Software Practice and Experience,* **7,** 85-93.
28. WILKES, M.V. (1976). Software engineering and structured programming, *IEEE Trans. on Software Engineering,* **SE-2,** 274-276.
29. WILKINS, A.G. (1976). Swepspeed-1 user guide, Central Electricity Generating Board, London, SSD/SW/75/N16.

Figure 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	USER 1	Create a procedure called ABC and associate input device with it. ABC has default password USER1.
10 . . .	ABC	Enter statements 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 statement 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.

Figure 2. Some Examples of Shared Data Manipulation

CONSOLE INPUT	COMMENT
LOGON USER1	Password USER1 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;	A: read and write;
ACCESS (B)=0	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 (no data)
.	
.	
.	
100 LOCK COM2	ABC will suspend until FREE COM1 in line 210 of XYZ
.	
.	
.	
LOGOFF	

Figure 3: Data Manipulation Example

```
SUBROUTINE SUB A(N,X)
   M = 20                          Dimension according to a variable value
   DIM A(M)                        Static allocation
   A(1)=10; A(2)=27; A(3)=. . .    Assign some values
   ACCESS (A) = READA              Protect by setting access to read only
START                              End of initialization section
DIM B(N), C(N, 3*N)                Dynamic allocation of array space
.
.
.
.                                  Perform operations with arrays
DIM B(0), C(0)                     De-allocate before exit
RETURN
```

Figure 4: Examples of Run and Wait Statement Syntax

```
RUN STARTUP                        Run once
RUN LOOP. CONTROL EVERY 5 SECS
                                   Execute repetitively
LOGH= 1,5                          Times can be fractional
RUN LOOP. LOG EVERY LOGH HOURS IN 2O*LOGH MINS
                                   Repetitive after initial delay
NEXT SHIFT=8* INT(HOURS+8)/8)
                                   Next shift time (HOUR=current time)
RUN SHIFT. REPORT EVERY 8 HOURS AT NEXTSHIFT: 0
                                   Run at shift changeover
RUN DAILY.REPORT AT 08:00:30 EVERY 24 HOURS
                                   Run every day at 30 secs after 8 a.m.
WAIT  1.3 SECS                     Floating point values and expressions
WAIT X*Y/Z MINS                    permitted, HOURS is also a qualifier
```

FIGURE 5: 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 1: Viper VS Prosic and HP Basic

Total number of statements [1]	Seconds to execute loop VIPER[2]	PROSIC[3]	HP BASIC[4]
7	4.4	4.4	3.0
50	4.4	8.5	6.6
100	4.4	12.8	10.6
200	4.4	21.3	18.7

```
100 FOR I = 1 TO 1 000
101 IF I < 500 THEN 104
102 X = I
103 GOTO 105
104 X = I
105 NEXT I
106 END
```

Notes:
(1) The additional statements are outside the loop.
(2) VIPER running on MIKROV [26]
(3) PROSIC on MIKROV [26]
(4) Hewlett Packard BASIC 20392A running on HP21MX

Table 2: A Benchmark

COMPUTER AND LANGUAGE	MILLISECS PER LOOP
Published Data (10)	
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 running on MIKROV [26]	12.0
VIPER using floating point firmware and reverse Polish	4.2

```
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
```

Table 3: Case Study Program Statistics

	Module type	No. of modules	No. lines source code[1]	Code size words	Averages Lines module	Words line	Words module	Time busy per 60 secs CPU	Seconds Swap	Total
Fortran	A	15	1 178	32 276[2]	78.5	27.4	2 151	1.2	7.8	9.0
	B[4]	4	238	19 807	59.5	83.2	4 951			
	C	7		34 085						
	D	1	—	758	(memory resident global common areas)					
	E	10	(Disc files)							
Viper	A	16	638	12 501[3]	42.7	18.3	781	7.9	0	7.9
	B[4]	8	171	2 830	21.4	16.5	353			
	C	—	(not required because of interactive facilities)							
	D	8	—	390						
	E	3	123	2 799	41.0	22.8	933			

A — Repetitive tasks
B — Non-repetitive or infrequent
C — Monitoring and operator
D — Common data areas
E — Error messages

Notes
(1) Excluding comments
(2) Including non-reentrant library modules
(3) Including symbol table and SVMM overhead
(4) Type B functionally equivalent but not comparable line-for-line
(5) Hewlett Packard RTE FORTRAN 92060 - 16092 on 21MX.
(6) VIPER running on MIKROV (18)

Notes for Contributors

The purpose of this Journal will be to publish original papers in any field of computing. Papers submitted may be research articles, review articles, exploratory articles or articles of general interest to readers of the Journal. The preferred languages of the Journal will be the congress languages of IFIP although papers in other languages will not be precluded.

Manuscripts should be in double-spaced typing on one side only of Henderson or Prof. M. H. Williams at

Rhodes University
Grahamstown 6140
South Africa

Form of manuscript

Manuscripts should be in double-space typing on one side only of sheets of A4 size with wide margins. The original ribbon copy of the typed manuscript should be submitted. Authors should write concisely.

The first page should include the article title (which should be brief), the author's name, and the affiliation and address. Each paper must be accompanied by a summary of less than 200 words which will be printed immediately below the title at the beginning of the paper, together with an appropriate key word list and a list of relevant Computing Review categories.

Tables and figures

Illustrations and tables should not be included in the text, although the author should indicate the desired location of each in the printed text. Tables should be typed on separate sheets and should be numbered consecutively and titled.

Illustrations should also be supplied on separate sheets, and each should be clearly identified on the back in pencil with the Author's name and figure number. Original line drawings (not photoprints) should be submitted and should include all relevant details. Drawings, etc., should be about twice the final size required and lettering must be clear and "open" and sufficiently large to permit the necessary reduction of size in block-making.

Where photographs are submitted, glossy bromide prints are required. If words or numbers are to appear on a photograph, two prints should be sent, the lettering being clearly indicated on one print only. Computer programs or output should be given on clear original printouts and preferably not on lined paper so that they can be reproduced photographically.

Figure legends should be typed on a separate sheet and placed at the end of the manuscript.

Symbols

Mathematical and other symbols may be either handwritten or typewritten. Greek letters and unusual symbols should be identified in the margin. Distinction should be made between capital and lower case letters between the letter O and zero; between the letter l, the number one and prime; between K and kappa.

References

References should be listed at the end of the manuscript in alphabetical order of author's name, and cited in the text by number in square brackets. Journal references should be arranged thus:

1. ASHCROFT, E. and MANNA, Z. (1972). The Translation of 'GOTO' Programs to 'WHILE' Programs, in *Proceedings of IFIP Congress 71,* North-Holland, Amsterdam, 250-255.
2. BÖHM, C. and JACOPINI, G. (1966). Flow Diagrams, Turing Machines and Languages with only Two Formation Rules, *Comm. ACM,* **9**, 366-371.
3. GINSBURG, S. (1966). *Mathematical Theory of Context-free Languages,* McGraw Hill, New York.

Proofs and reprints

Galley proofs will be sent to the author to ensure that the papers have been correctly set up in type and not for the addition of new material or amendment of texts. Excessive alterations may have to be disallowed or the cost charged against the author. Corrected galley proofs, together with the original typescript, must be returned to the editor within three days to minimize the risk of the author's contribution having to be held over to a later issue.

Fifty reprints of each article will be supplied free of charge. Additional copies may be purchased on a reprint order form which will accompany the proofs.

Only original papers will be accepted, and copyright in published papers will be vested in the publisher.

Letters

A section of "Letters to the Editor" (each limited to about 500 words) will provide a forum for discussion of recent problems.

Hierdie notas is ook in Afrikaans verkrygbaar.

Quaestiones Informaticae

Contents/Inhoud

Spectrum Offset