ACKNOWLEDGMENTS

I would like to take this opportunity to express my sincere thanks to my thesis supervisor Dr. T. Fancott who outlined the philosophy and direction of this research, and has read the numerous drafts of this report making suggestions for its improvement.

I would also like to thank my parents who have patiently encouraged me through this work, and my wife, Elizabeth, who has not only shown great patience but has also contributed by reading and typing many drafts of this thesis and by making suggestions to improve it.

Finally, I would like to thank my colleagues and friends for their moral support and assistance, especially, Angus for several useful discussions on programming language and system implementation issues, and Peter for his assistance in typing the references.

I gratefully acknowledge the financial assistance from the FCAC.

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

This thesis is concerned with the design of language structures for implementing real-time/system programs on a "bare machine". The term "bare machine" is used here to indicate the absence of any nucleus, kernel, or virtual machine on which programs in the language implementing the structures run. The class of machines for which the structures are designed are small computers, characterised by the present generation of mini and micro computers. These are the architectures most often used in dedicated applications and custom operating system work.

The usefulness of a set of language structures may be assessed by properties such as: freedom from run time support and preempted system design decisions, and functional completeness which refers in this work to the coding of real-time/system application programs without recourse to the use of assembly language.

The importance of these properties and the "bare machine" concept must be emphasized, especially for systems to be implemented on small machines. The time critical nature of most real-time/systems work makes language structures which preempt system design decisions undesirable. The philosophy of the language designer in this regard must be to provide language structures which do not require the language implementer to make system design decisions. Freedom from run time support is desirable in any real-time/systems implementation language. This property has additional significance in a small computer environment where the principal application areas are dedicated or stand alone systems. Functional completeness is another desirable language property. Functional completeness makes it possible to eliminate the use of assembly language in system implementation. This not only enhances the portability of programs implemented in the language, but also makes it relatively simple to adapt these programs to changes in system configuration.

Language structures characteristically required in real-time/systems work are: structures for sequential programming, structures for multiprogramming, and structures for performing I/O. Our major efforts in this thesis have been directed towards determining language structures which not only provide these facilities, but also possess the desirable properties mentioned above. Structures for sequential programming which satisfy our requirements are almost routinely provided by most real-time/systems implementation languages. Language structures for the two other capabilities provided in contemporary implementation languages do not, however, fully satisfy our requirements. The principal language design decisions therefore addressed in this work are: decisions concerning structures for multiprogramming, and decisions concerning structures for programming peripheral devices.

Contemporary real-time/system implementation languages address these subjects from different points of view which classify them into two categories. Languages in the first category provide a set of structures which allows the implementation of a complete system in a single language. These structures, however, imply a run time system, which preempts an extensive set of design decisions from the system designer. Languages in the second category do not preempt system design decisions. They do not, however, provide a complete set of language structures for implementing real-time/systems programs. These languages therefore require the use of assembler language for any realistic real-time/system work.

Our investigations concerning the structures required for implementing multiprograms, and for performing real-time I/O, have led us to some fundamental conclusions regarding their characteristics. Firstly, the study suggested the need for language structures to represent paths of control in a program, and explicit means for controlling them. The sequential nature of the machine suggested that the means for controlling the control paths should be explicit rather than implicit. Consideration of the way in which contemporary hardware supports multiprogramming, and our efforts in determining structures for performing I/O, forced us to recognise the need for language structures for processing the interrupts in a system. The dual purpose served by such a structure requires that it have certain capabilities. Its role in processing I/O interrupts requires that it possess the capability of returning to the interrupted control path. On the other hand its role in processing timer interrupts requires that it have the capability of transferring control to a control path other than the one which was interrupted. In either case, however, it is essential that the structure processing the interrupt preserve the status of the interrupted control path.

These conclusions have led us to introduce in this work, two new language structures: "coprocesses" for representing multiple paths of control in a program, and "interrupt handlers" for processing the interrupts in a system. These structures are sufficient for implementing multiprograms, and for implementing structures which perform I/O in a bare machine environment. In conformity with the requirements of structures for implementing real-time/system programs discussed above, they neither require run time support nor preempt system design decisions. In addition, they are functionally complete, which allows implementation of a complete system without recourse to the use of assembler language.

This thesis is organised in two parts: a general background in language design considerations, and the design of specific language constructs for the implementation of real-time/system programs on small computers in a bare machine environment.

The first part, which consists of chapter 2, discusses the major issues in language design. These include: application area, programming methodology, and implementation considerations. Insofar as application area and implementation considerations have been extensively discussed in the literature, the discussion of these subjects in chapter 2 is brief. Moreover, since our major interest in this thesis is real-time/system implementation languages, the discussions are oriented towards these applications. A large part of chapter 2 is devoted towards identifying the influence of the programming methodology "structured programming" on the design of programming languages. This has been included because of the growing recognition of its importance in language design. This question of methodology underlies the motivation of the structures developed in this work. The coprocess and interrupt handler structures introduced in part 2 of this thesis are instances of temporal and procedural abstractions which have been designed specifically for implementation in a bare machine environment. Chapter 2 provides the reader with the background of the work done in this area along with a general concept of abstractions in programming languages.

Part 2 of this thesis, which consists of chapters 3, 4, and 5, is essentially devoted to the design of language structures for implementing real-time/system programs for small computers in a bare machine environment. The structures are designed to be incorporated in a block structured programming language. Since PASCAL is rapidly gaining in popularity, it is used as a host for the concepts developed here. The application area, essentially real-time/systems applications, and the host language PASCAL suggest the name "real-time/system PASCAL", abbreviated here to RT-PASCAL. This name, RT-PASCAL, is used to refer to the language incorporating the structures developed in these chapters. Chapter 3 provides a basis for the design of RT-PASCAL, and discusses an approach for making language design decisions concerning structures for multiprogramming and for performing real-time I/O. Structures for these purposes, namely coprocesses and interrupt handlers, are developed in detail in chapters 4 and 5 respectively.

CHAPTER 2

ISSUES IN LANGUAGE DESIGN

2.0 INTRODUCTION

A great deal of research has been done on language design considerations. Suggestions to language designers have been given by [Hoare-81], [Richard-76], [Wirth-74b], [Wulf-77]. Their suggestions can be classified into: application area, programming methodology, and implementation considerations. The application area enables a language designer to characterize programs of the application, thus providing a basis for the design of the language as well as providing criteria for its evaluation. Programming methodologies address the problems of design and coding of programs. For full effectiveness in the use of these methodologies, it is necessary that programming languages have structures which support their use. Programming methodologies therefore provide a basis for the design of features in a programming language. Implementation considerations permit a language designer to estimate the cost of implementing the features in a language.

A typical language design cycle is as shown in figure 2.1. The required features of a language are defined by a number of factors. The requirements of the application area will have a

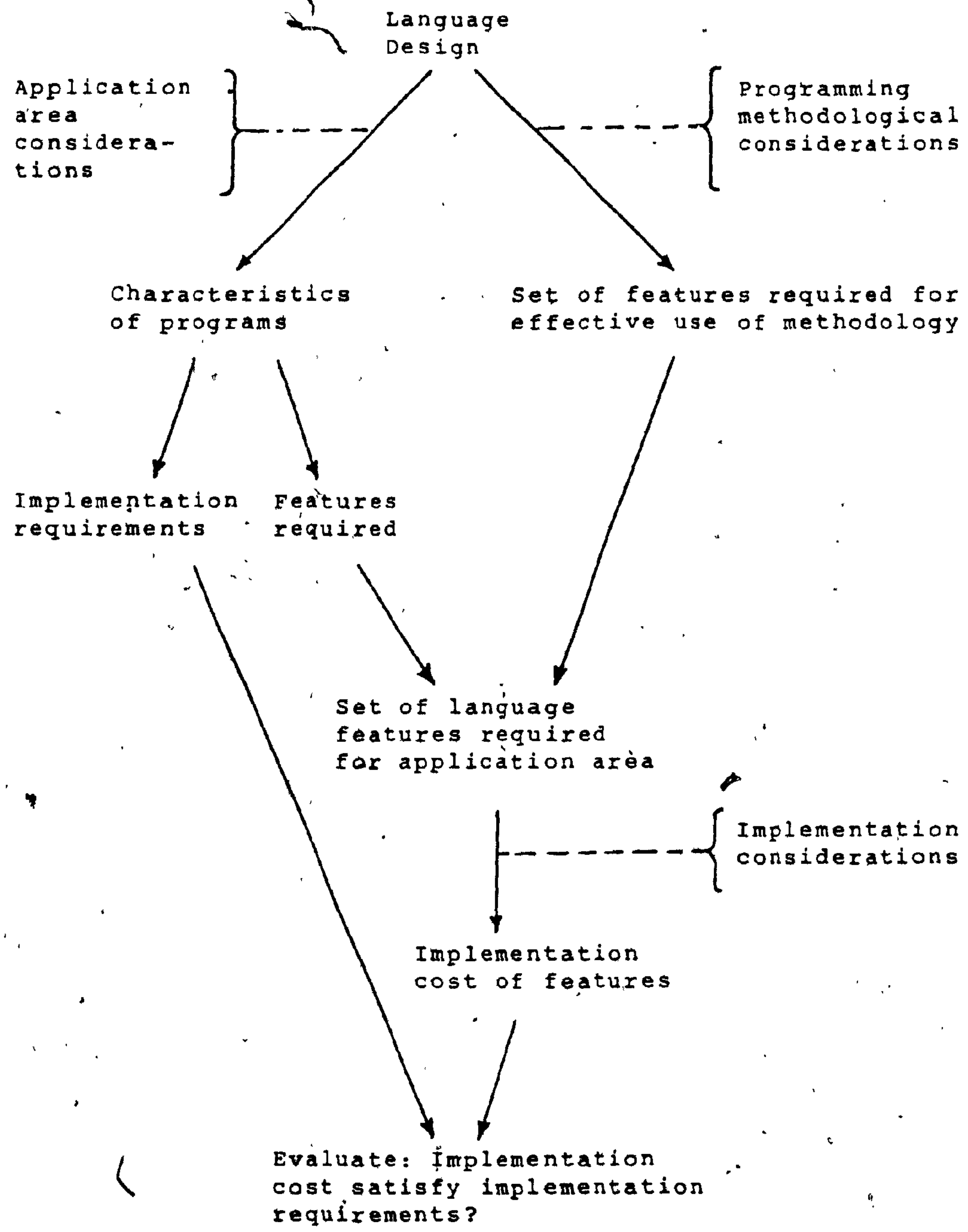

Language Design
Application area considerations
Programming methodological considerations
Characteristics of programs
Set of features required for effective use of methodology
Implementation requirements
Features required
Set of language features required for application area
Implementation considerations
Implementation cost of features
Evaluate: Implementation cost satisfy implementation requirements?

Figure 2.1 Considerations in Language Design

strong influence on the set of features to be included. Programming methodologies, such as structured programming, will dictate the inclusion (or sometimes the exclusion) of constructs, and influence the design of the application oriented features.

The implementation requirements of these features are themselves influenced by the application area, in aspects such as the required degree of implementation efficiency, or availability of run time support. The cost of implementing these features can then be determined. These implementation costs can then be evaluated using the implementation requirements of the application area to determine their suitability in the language.

This chapter is organised as follows. Section 2.1 discusses the importance of application area considerations in language design, and illustrates their influence with a few examples. Section 2.2 examines the impact of programming methodological considerations on language design. The methodology specifically considered is "structured programming" [Dijkstra-72a]. The major portion of this section examines the need for abstraction in programming, and its influence on language design. Issues surveyed in lesser detail are: sequencing mechanisms, data structuring facilities, facilities for top down development of programs, and facilities for hierarchially structuring programs. Section 2.3 discusses the implementation considerations in language design. Major implementation strategies and policies in language implementation which affect the characteristics of programs implemented in a language are discussed. Also investigated are the language design decisions which affect the choice of these implementation strategies and policies.

2.1 APPLICATION AREA CONSIDERATIONS

Current technology in programming language design and implementation tends to indicate that successful languages will continue to be those that are designed to suit a particular class of applications. This situation is indicated by:

1) The failure of general purpose languages in gaining widespread usage [Hoare-81]. Prominent examples are PL/1 and ALGOL-68. This situation continues in spite of the fact that PL/1 receives support from a major computer manufacturer, and that ALGOL-68 was designed by a team of internationally recognised experts.
2) The success of languages designed for particular applications. Examples are: FORTRAN, ALGOL-60 for scientific applications, and COBOL for business applications.
3) The absence of new concepts for design of general purpose programming languages [Seegmueler-76]. The concept of extensible languages has not fulfilled its initial promise for development of families of

languages.

It therefore appears that any serious language design effort must pay considerable attention to the class of applications which the language is to serve.

In light of the apparently infinite range of computer applications, the above arguments might suggest a corresponding proliferation of languages. Such a fine classification is, however, not necessary. Existing programming languages constitute a much coarser classification. Their features and implementation characteristics cover large classes of applications adequately, if not optimally. Three major application areas are apparent:

1) Simulation and modelling applications.
2) Real-time and systems applications.
3) Non real-time/systems and non simulation/modelling applications.

This categorisation is appropriate to applications traditionally covered by general purpose programming languages. Application areas excluded from this are non-procedural, and/or applicative languages such as query and list processing languages. Table 2.1 summarizes the difference in the characteristics and implementation requirements of these application areas.

The application area enables a language designer to characterize programs for the application. This makes it

	Simulation Applications	Real-Time/System Applications	Non Real-Time/System Non Simulation Applications
1) process features	required, programs generally consist of a fixed number of processes	required, programs generally consist of a dynamically varying number of processes	not required
2) degree of run time permitted	elaborate run time support is permitted	elaborate run time support should be avoided as far as possible	elaborate run time support permitted
3) relative importance of efficiency	important	very important	important
4) relative importance of preemption of system design decisions	important but may be preempted	very important, should be left to the programmer whenever possible	important but may be preempted
5) I/O facilities required	I/O statements	facilities for performing real-time I/O.	I/O statements
6) typical length of software lifespan	short	long	medium

Table 2.1 Programming language application area characteristics

possible to determine the type of structures required in the language, as well as the implementation requirements of programs written in the language. The application area therefore not only serves as a basis of a language design, but also serves to provide criteria to evaluate the design of a language.

As an example of the way in which the application area influences the design of features, consider programming languages such as FORTRAN and ALGOL-68. These languages are primarily designed for scientific applications, which come under the non real-time/system and non simulation/modelling group of applications, and therefore do not have facilities for temporal abstractions. Another example is the inclusion monitor-like features [Hoare-74], [Brinch Hansen-73a] in languages such as MODULA [Wirth-77a], and CONCURRENT PASCAL [Brinch Hansen-75a], [Brinch Hansen-77a]. These languages are primarily intended for real-time/systems applications, and support their need for temporal abstractions with language constructs for defining processes. A consequence of using process structures implies the need for synchronization, hence the inclusion of monitors in these languages.

As an example of how implementation requirements can influence the design of features in a language, consider again languages such as CONCURRENT PASCAL and MODULA. These languages do not permit features such as dynamic arrays, dynamic processes, and general recursive procedures (permitted to a very limited extent in MODULA). Implementation of these features require dynamic storage management. The absence of these features is attributed to the implementation requirement of real-time/system applications concerning elaborate run time systems.

2.2 PROGRAMMING METHODOLOGICAL CONSIDERATIONS IN LANGUAGE DESIGN

Traditionally, languages have been designed in an ad-hoc manner. Consideration was mainly given to the application and target machine of the language e.g. FORTRAN which was initially designed for scientific applications for a particular IBM machine (see [Wirth-73]). Very little attention was paid to the design of features that would enhance the quality of programs written in the language. New languages attempt to improve the quality of programs written in a language by including features deemed necessary on methodological grounds. Examples are: CLU [Liskov-75a], [Liskov-77], ALPHARD (see [Wulf-77]), CONCURRENT PASCAL, MODULA, ADA [Ichbiah-79a], MESA [Geschke-75], [Geschke-77], EUCLID [Lampson-77], MODEL [Johnson-76], [Morris-79], and ASBAL [Moss-77].

Software quality is generally estimated by properties such as portability, reliability, efficiency, understandability, human engineering, testability, and modifiability [Boehm-78]. It is vital that these properties should be reflected in the program text [Wulf-77], [Wulf-76],. The language used to code programs therefore plays an important role in determining the quality of the resultant product.

As an extreme example of the role of language, consider a program coded in two languages: assembly and a high level language. In addition to having a shorter development time (due to greater programmer productivity), the program written in the high level language will be more reliable, understandable, portable, and possibly more efficient and modifiable than the assembly language version of the program. Programming languages, however, are not the only factor contributing to the quality of programs. Properties such as portability are largely language dependent. Other properties such as efficiency, reliability, understandability, and modifiability, however, also depend on the factors such as program structure and modularity [SP-76]. These factors, primarily influenced by design strategies and coding desciplines, are some of the subjects addressed by programming methodologies.

Programming methodologies such as "structured programming" [Dijkstra-72a], address the general problem of producing quality software. A good (complete) methodology e.g., the chief programmer team, ideally addresses all the major issues in software development [Baker-73], [Mills-73]. This includes design strategies, coding and documentation standards, and management policies. When properly used, a programming methodology can be the single most important factor contributing to the development of quality software. This can be inferred from the success of systems such as: the "THE" system [Dijkstra-68b], [Mckeag-76], the "New York Times Information Bank" [Baker-72a], [Baker-72b], [Baker-73], and the VENUS operating system [Liskov-72a]. Two points must be noted in the use of methodologies. They do not guarantee programs which are well structured and modular. They do, however, increase the likelihood of producing well designed programs. To be fully effective, the methodologies must be supported by the language in which the program is coded. More precisely, a language must include those features which support the use of the methodology. This implies the importance of programming methodological considerations in language design.

This section considers some of the language features required for supporting the programming methodology "structured programming" [Dijkstra-72b]. The term structured programming as used by Dijkstra, Wirth [Wirth-71a], [Wirth-74a], and others [Denning-76a], [Gries-74], [Woodger-71] refers to a constructive method of producing correct programs. Although it is not a complete methodology, it addresses those issues in software design which largely influence language design. In particular it addresses the design and coding stages of software development with the following procedural and structural recommendations:

1) Strict adherence to the use of rigid (simple) sequencing statements in programming.
2) The use of hierarchy to control program complexity to manageable proportions.
3) The use of stepwise refinement in program development.
4) The proof of program correctness (informally) during programming development.

The first recommendation has had tremendous influence on the design of control structures in languages. Originally, it only addressed the design of sequencing mechanisms in sequential processes [Dijkstra-68c]. The debate following its introduction, however, has motivated corresponding discussions on the design of synchronisation primitives for concurrent processes [Hoare-74], [Brinch Hansen-73a], [Brinch Hansen-72a], [Campbell-74]. In addition, it has motivated discussion on the design of data structuring facilities [Hoare-72], [Hoare-78]. Sequencing mechanisms for sequential and concurrent processes are discussed in section 2.2.1.

The second recommendation requires that the structure of a program be reflected in its text. Some useful types of hierarchical program structures and their virtues are discussed by [Goos-75b], [Turner-80]. Language support for hierarchically structuring is discussed by [Goos-75a]. Examples of languages which partially support hierarchically structured programs are block structured languages such as

ALGOL-60 and PASCAL. Some deficiencies of these languages concerning global variables are discussed by [Wulf-73]. Various solutions to these problems are proposed in languages such as MODULLA, EUCLID, and ADA. More complete support is provided by languages such as CONCURRENT PASCAL, CLU, ALPHARD, SIMULA [Dahl-66], [Birthwhistle-70].

The procedural recommendation, advocating stepwise development of programs as a method for achieving hierachically structured programs, implies the need for two facilities in the languages:

1) Abstraction facilities.

2) Facilities for top down development of programs.

Various top down design methodologies are discussed by [Dijkstra-72b], [Wirth-74a], [Wirth-71a], [Baker-72a], [Baker-72b], [Baker-73]. An example of a language which supports top down design and implementation is CLU. Facilities for abstraction which are of direct interest in this thesis, are discussed in section 2.3.2.

Dijkstra's suggestion on proving programs correct during program development has motivated a corresponding development of language structures which are amenable to proofs of correctness using techniques such as those proposed by [Hoare-69]. Examples of language structures influenced by this recommendation are: the assert statement [Popek-77], the GOTO statement [Wulf-71], and aliasing [Popek-77].

2.2.1 SEQUENCING MECHANISMS

2.2.1.1 SEQUENCING MECHANISMS FOR SEQUENTIAL PROGRAMS

Many of the early discussions on the subject of sequencing mechanisms for structured programming suggest the use of three basic types of sequencing in programming, and the elimination of the GOTO statement as a means of controlling the flow of control in a program. The three mechanisms advocated are:

1) Concatenation, which is represented by placing the statements of programs in a sequence.
2) Selection, implemented by statements such as the IF-ELSE-THEN, and CASE.
3) Iteration, implemented by statements such as the WHILE, REPEAT, and FOR.

The main justifications offered in support of rigid adherence to this policy are:

1) Any program containing GOTO statements can be converted into an equivalent program without GOTO'S using these basic sequencing statements [Bohm-66].
2) These statements correspond to commonly used patterns of reasoning in problem solving i.e. identifying and placing the steps in a solution in an orderly sequence, enumerating and separately solving the cases of a problem and repeating a problem step until a satisfactory result is obtained [Denning-76].
3) These statements correspond to the some of the most

powerful proof techniques in mathematics, i.e. proof by considering each step in turn, proof by enumeration of cases, and proof by induction [Denning-76], [Dijkstra-72a].

4) Efficient compilation of programs is made easier in languages which do not have GOTO statements.

5) Each statement or group of statements in a GOTO free program can be represented as a single entry single exit block. This enhances the intellectual manageability of a program (see [Zelkowitz-78]).

These recommendations gave rise to the GOTO controversy [Leavenworth-72]. A recent discussion on the subject by Knuth [Knuth-77] shows that the question of GOTO is in fact peripheral to the problem of constructing structured programs. He has shown several commonly occurring examples such as: loops that need multiple exits, and the 'N and a half' loop problem, which can not be adequately expressed by the three simple forms of sequencing. He therefore suggests the inclusion of at least two additional type of statements to handle these cases: Zahn's situation indicator loop [Zahn-74], and Dahl's 'N and a half' loop (see [Knuth-77].

2.2.1.2 SEQUENCING MECHANISMS FOR CONCURRENT PROGRAMS

There is a great deal of similarity between synchronisation and sequencing. Elson [Elson-73] argues that "synchronisation and sequencing are logically the same phenomenon, though intuitively they may seem quite distinct." For example a mutual exclusion semaphore is used to enforce sequential access to data and resources shared by cooperating processes.

The first solution to the problem of synchronising processes, which was easy to use and did not involve busy waiting, was provided by Dijkstra [Dijkstra-68a]. He proposed synchronisation primitives (the so called P and V operations) which operated on special variables called semaphores.

Semaphores are adequate for expressing the major synchronisation requirements of processes. Their use for achieving mutual exclusion, however, is often questioned because they are unstructured [Brinch Hansen-72a], [Brinch Hansen-73a], [Hoare-74]. Structures such as conditional critical regions [Brinch Hansen-72a], and monitors [Hoare-74], [Brinch Hansen-73a] are therefore advocated to overcome these problems.

Monitors are easy to use and also produce programs with good structure. Problems have, however, been found in constructing monitors, as the synchronisation code of the monitor is mixed with the code accessing the shared resource. Problems have also been found in implementing monitors: efficiency problems [Kessels-77], problems related to nested monitor calls [Lister-77], and problems related to excessive serialisation [Campbell-74]. Some of these implementation problems are addressed by the manager concept [Jammel-77]. A more comprehensive solution to both construction and implementation problems is, however, provided by path expressions [Campbell-74]. Path expressions achieve a separation between the synchronisation code and the code accessing the shared resource. In addition, they allow a programmer to specify concurrent access to shared resources. Path expressions, however, are hard to formulate. Suggestions to improve their formulation have been made by Bekkers et al. [Bekkers-77]. Approaches similar to path expressions have been proposed by Laventhal [Laventhal-78].

2.2.2 INFLUENCE OF STEPWISE REFINEMENT

Abstraction plays an important role in programs developed using stepwise refinement to guide the design process. The fundamental activity occurring in programs developed in this manner is the recognition of abstractions [Wulf-76], [Wulf-77], [Liskov-72b]. A program, using stepwise refinement, is developed in stages. At each stage the program is written using just those data objects and operations which are ideally suited to solving the problem. Some or all of these objects may not be primitive in the language. These are called abstractions. The process terminates when the lowest level abstractions have been implemented in terms of the primitives of the programming language.

5

There is a definite need for programming languages to support the expression of abstraction. A problem noted in programs developed using stepwise refinement, and which are implemented by a language not supporting abstraction, is that the abstraction and refinement steps are absent in the final program. Languages supporting abstraction have several potential benefits: abstractions act as units of modularity, thereby enhancing program modifiability and understandability; they reduce the apparent complexity of a complete program by displaying the various levels of abstractions; and programs expressed in these languages are more self documenting than others [Linden-76], [Meertens-77], [Horning-76].

The decision to support abstraction requires the specification of forms of abstraction which will be useful in programming, and the identification of the subset of these which requires programming language support. Programs represent a sequence of actions on data in time. Each of these major components of programs, actions, data, and sequence in time, can be subject to abstraction. This defines the three principal forms of abstraction: procedural, data, and control [Sahasrabuddhe-76]. Since these are abstractions of the major components of programs, they must be supported by programming languages. Sections 2.2.2.1, 2.2.2.2, and 2.2.2.3 discuss these abstractions and the support they need from programming languages.

There are other useful forms of abstraction. For example Sahasrabuddhe [Sahasrabuddhe-76] credits Knuth [Knuth-77] with the discovery of a form of abstraction that he calls structural abstraction. Using structural abstraction, programs are developed for clarity, and then subjected to transformation to improve their behaviour. Knuth suggests that structural abstraction is best supported by program development systems rather than by programming languages [Knuth-77]. Examples of such systems are those proposed by [Cheatham-72], [Darlington-73].

Another form of abstraction which is extremely useful for expressing a certain class of programs is temporal abstraction. Using temporal abstractions, elements of sequencing necessary for lower levels of description are eliminated from more abstract levels where they are no longer needed [Sahasrabuddhe-76]. Temporal abstractions and their language support are discussed in section 2.2.2.4.

2.2.2.1 PROCEDURAL ABSTRACTION

Procedural abstraction is a term used to refer to any abstract operation implemented by some unspecified algorithm in a program. Two types of procedural abstraction can be distinguished by the manner in which they are invoked. Abstractions invoked directly by the program text are referred to as functions or procedures. Those invoked directly by the hardware (in response to interrupt conditions) are called interrupt handlers [Wegner-71], [Wegner-68], [Organick-71]. Programming languages generally support procedures and functions by allowing a programmer to define a name to refer to a group of statements implementing some abstract operation. Execution of this abstract operation is caused by referring to the name. Interrupt handlers are usually not supported by programming languages. They are not discussed in this section.

Investigators studying the use made of procedural abstractions distinguish two categories of usage in practice [Parnas-75], [Goos-77]. They are:

1) Those that return a result but neither use nor make any changes to the environment.
2) Those that sometime return a result and also use and make changes to the environment.

More than one invocation of these abstractions, when supplied with identical actual parameters, will result in the return of identical results in the first case but not in the second case. In programming languages, these categories of procedural abstraction facilities are generally referred to as functions and procedures (subroutines) respectively. There is another reason for distinguishing between functions and procedures. Since functions neither depend nor make changes to the environment, programs using functions are much easier to prove correct than those using procedures. This reason is often offered as justification in providing these two alternate types of procedural abstraction facilities [Popek-77].

One of the most desirable features required of the procedural abstraction facilities, and one that has been traditionally supported by programming languages is parameterization. Parameterization serves to enlarge the scope of applicability of a single definition of a procedure. The definer of a parameterized procedure specifies the type and number of parameters required for correct operation of the procedure. When the procedure is invoked, the invoker must supply the actual parameters or arguments of the procedure.

A language designer has to make several decisions in the design of a procedure parameterization facility of a language. The first class of decisions are primarily syntactic, being concerned with issues such as positional versus keyword parameter passing conventions [Hardgrave-76], and the use of default values [Ichbiah-79b]. These decisions, essentially dictated by taste, are also influenced by the application area. For instance default values and keyword parameter passing conventions are appropriate in JCL languages. Conclusive evidence indicating the superiority of any particular policy in less special purpose languages is not available.

The second design decision concerns the binding mechanism, or rules used to bind actual parameters to the formal parameters of a procedure. A survey of several binding policies commonly used in programming languages is available in [Waite-76a], [Aho-77]. This decision is of vital importance as it has a considerable impact on the efficiency and understandability of a program. For example, the complex run time behaviour sometimes displayed by "call by name" binding mechanisms makes them difficult to understand, whereas "call by value" parameters can be extremely inefficient for passing large data structures. Two essential properties that a set of parameter binding mechanisms must display are: ability to pass a value as an argument, and ability for a procedure to change the global environment (without resorting the use of global variables). Again, no evidence is available indicating the clear superiority of any particular set of parameter binding mechanisms. Taste is therefore an important factor in deciding on a set of binding mechanisms. Experience of successful languages such as PASCAL, however, provides a basis for making these decisions.

The third decision that must be made is the use of automatic coercion [Waite-76b] of actual parameters to formal parameters. Automatic coercion is usually rejected on the grounds of understandability, efficiency and reliability [Parnas-76]. The need for automatic type coercion, however, has been observed by [Geschke-75]. The present trend is to permit automatic coercion in certain limited contexts while other coercions must be specified by the programmer

[Geschke-75].

Lastly, a language designer must make decisions to deal with "aliases" in programs [Popek-77]. Aliasing is caused by overlap of parameters. This causes procedures to return unexpected results, and complicates proofs of correctness. New languages such as EUCLID include features to detect aliasing.

Recent developments in programming methodology have indicated the need for two additional features in a procedural abstraction facility: generic definitions, and overloading of procedure names. Parameterization considerably expands the scope of applicability of a single definition. Restricting parameters to data objects however, constrains the applicability of a procedure. It is not possible, for example, to define a single procedure to sort an array of reals and integers in PASCAL. Permitting types to be valid procedure parameters presents a way to define operations which are applicable to a wider variety of objects. Such procedures, called generic procedures, are discussed by [Demers-76], [Schuman-75], [Wegbreit-74].

Traditionally, programming languages either prohibit two procedures from having identical names, or hide procedures with the same name. The need for generic procedure and generic data definition and data abstraction features in programming languages has, however, created a requirement for a more flexible naming policy for procedure definition and identification. This policy should permit more than one procedure to have the same name without hiding each other. Such procedures, called overloaded procedures, are discussed by [Demers-76].

A language designer must also make decisions concerning the use of global variables in a procedure body. Procedures serve as units of modularity [Horning-76]. It is therefore desirable that a procedure body contain no free variables [Dennis-75]. The advantages of modularity, however, must be weighed against the increased overhead and inconvenience (consequently encouraging error) caused by a long parameter list. In essence, however, this is a trade off that is most effectively made by the programmer. Many new languages provide a compromise solution to this dilemma. They permit free variables in a procedure body, but require the programmer to explicitly list the free variables used by the procedure. An example of such a language is EUCLID.

2.2.2.2 DATA ABSTRACTION

Data objects in programs generally represent certain abstract quantities in the problem being solved. In practice there are only certain specific operations that can be meaningfully applied to these data objects. These operations are called the behavioural properties of the data objects. The data objects and operations may, however, not be present in the language to code the program, and may therefore have to be implemented (in the simplest case) in terms of primitive language structures. These implementation details are called representational properties of the data object [Shankar-80], [Nestor-76], [Liskov-74].

The programming methodology "stepwise refinement" suggests a two stage development of data objects used in programs. At the first stage, the programmer decides on the data objects and operations necessary to solve the problem. At this level he is solely concerned with the behavioural properties of the data objects. At the second stage, the programmer is concerned with the implementation of the data objects and the operations. Here, he is solely concerned with finding representations that will exhibit the desired behaviour. The term 'data abstraction' is generally used to describe this concept of design of data objects.

A programming language structure intended to support data abstraction should permit the use of data objects by naming them without any knowledge of their implementation. This requires that the structure used for implementing data abstractions should allow: the designer to define the representation as well as the operation of the data objects, the implementer to hide those representational details and operations which the user need not to know in order to use the object, and the implementer to force the user to manipulate the object using the operations provided and not by direct manipulation of its storage representation [Liskov-75a], [Shaw-80], [Shankar-80].

Traditional programming languages do not support data abstraction. Procedural abstractions are suitable for expressing abstract operations. They are, however, inadequate for representing data abstractions for two reasons: data abstractions require several operations or entry points which are hard to simulate with a single procedure, and it is difficult to express the representation of a data object using a procedure. The need for another program structure for supporting data abstraction is therefore apparent.

Programming language structures that form the basis of conventional data abstraction facilities are:

1) Class structures [Dahl-66], and

2) User defined type definitions [Wirth-71b], [Wirth-72].

The principle difference between these is the manner in which the data objects are created, and the rules which govern their lifetimes.

Class structures support data abstraction by permitting a programmer to define the data representation (i.e data structure) of the class object as well as the procedures that operate on objects of the class. These representational details and procedures are called the attributes of the class.

Class objects in SIMULA are created dynamically by special operations, and have a lifetime independent of the procedure that creates the object.

Type definition facilities support data abstraction by permitting a name to be associated with the application of a data structuring operation. This name serves to characterize the representation and define certain primitive access algorithms on object of the type. In addition, this name is used for:

1) Declaration and denotation of objects of the type.

2) Identification of generic objects.

3) Error detection by type checking.

Objects of the type are created when the procedure in which they are declared is called and cease to exist when the procedure is exited.

Neither class structures nor user defined type definitions provide complete support for data abstraction. Both approaches fail to provide means to enable the definer of a data abstraction to prevent a user from directly manipulating the representation of the data abstraction. In SIMULA's class structures, all the attributes of the class are visible to the user. In PASCAL, the fine structure of the type definitions is visible to all programs using the object of the type [Koster-76]. Useful data abstraction facilities can be obtained by adding to class structures and type definitions means for controlling access to the attributes of a class or to the fine structure of a type definition. Examples of data abstraction facilities based on class structures are CLU, and ALPHARD. Those based on type definitons are MODULA, ADA, MODEL, and MESA.

Consider for example the data abstraction facilities in CLU and MODULA. In CLU the data abstraction facility is called a "cluster". Clusters are similar to classes, except that they permit the definer to explicitly specify which of the defined objects defined by the cluster are available to the user. This allows the implementer to hide details of representation of the abstraction. The data abstraction facility in MODULA is the module. Modules permit the definer of an abstraction to explicitly state which objects defined by a module are available outside it. Furthermore, when type names are exported in this manner, the fine structure is not automatically exported. This prohibits the user from making use of the representation of object of the type.

Other features considered desirable in data abstraction facilities are overloading, generic definitions, and uniform references.

Overloading has already been discussed in section 2.2.2.1 on procedural abstraction. Its importance in languages supporting data abstraction is attributed to the flexibility it provides definers of different abstractions in providing symbolic names to the operations of data abstraction without the danger of causing name conflicts or hiding.

Generic data definition facilities, like generic procedure defintions, permit the definer of a data abstraction to expand considerably the scope of a single definition. Such definitions require that the language allow overloading and type or class definitions to accept parameters. An instance of a generic object is created when actual parameters are supplied to the formal parameters of a generic definition. An example of a language which allows such definitions is CONCURRENT PASCAL. More sophisticated generic definition facilities also allow type or class objects as parameters. Examples of such languages are CLU and ALPHARD. Correct operation of such definitions requires that the language possess facilities which allow the definer to specify acceptable actual parameters [Wulf-77], [Mitchell-78]. An example of a language which has such facilities is ALPHARD.

The property of uniform references gives an implementer of a data object freedom in determining the representation of an abstract data object. This freedom is achieved by incorporating in a language facilities for uniformly referring to attributes of an object, irrespective of its representation. The importance of this property was first recognised by [Ross-70]. A sophisticated implementation of such a facility is discussed by [Geschke-75]. It is available in varying degrees in languages such as MESA, and CLU.

2.2.2.3 CONTROL ABSTRACTION

The concept of a program as a sequence of actions in time on data suggests the need for mechanisms for sequencing arbitrary sets of actions. Some sequencing mechanisms appropriate for structured programming were discussed in section 2.2.1. Section 2.2.2.1 on procedural abstraction discussed means for sequencing user defined actions. These sequencing mechanisms are instances of control abstractions. The discussions on procedural and data abstraction facilities in sections 2.2.2.1, and 2.2.2.2 respectively, discussed facilities which permit a programmer to define a more convenient set of operations and data objects than those provided as primitives in the programming language. A corresponding discussion of control abstraction would involve consideration of a set of control primitives from which a wide variety of control structures could be derived [Prenner-71], [Prenner-73], [Bobrow-73]. The following discussion is limited to fixed set of control primitives defined by the language. This excludes the possibility of user definition. In general it is possible to conveniently solve problems in a particular application area with a small set of standard control abstractions.

Data abstraction facilities in a programming language accentuate the need for a new control structure for iteration. The purpose of many loops in programs is to iterate through a collection of objects, and to perform some action on the elements of the collection. Some deficiencies of WHILE, REPEAT and FOR loops used for this purpose are:

1) They specify too much, preventing the compiler from finding efficient representations.
2) They do not clearly express the purpose of the loop. Code for producing the element from the collection is mixed with the code which performs the action on the element.
3) When the collection is a data abstraction, these loops subvert the data abstraction, as it is necessary to allow the programs of such a loop to access the representation.

Alternate structures are therefore suggested for this purpose [Liskov-77], [Gries-77], [Wulf-77].

As an example of these structures consider the "generators" in ALPHARD. A generator allows a programmer to specify the order of traversal over a collection of objects, without raising any of the problems mentioned above.

A generator is used in conjunction with FOR statements, which invokes at appropriate points, five basic operations:

1) Start, which initialises the loop.
2) Done, to determine whether the loop is finished.
3) Value, to determine the value of the current element in the collection.
4) Next, to step to the next element in the collection.
5) Finish, which performs clean up operations on

termination of the loop.

These operations are provided by the definer of the generator.

Generators avoid all the problems mentioned above. Firstly, generators allow a programmer to separate the code used to produce the next element in the collection from the operation which must be performed on the element, thus clearly expressing the purpose of the loop. Secondly, since predefined collections such as array objects can be provided with predefined generators for iterating over them, it is possible for the compiler to implement them efficiently. Finally, the use of a generator to iterate over a collection of objects does not require the user to be aware of any representational details of either the collection or the generator, and therefore supports data abstraction.

2.2.2.4 TEMPORAL ABSTRACTIONS

A characteristic of programs constructed using the abstraction facilities discussed so far is that they consist of a single path of control. In such programs the state of the system at any time can be determined by examining its progress. Some programming applications, however, may require facilities for implementing programs consisting of multiple paths of control. The major motivations for such facilities are economic and methodological.

It is fairly easy to see the need for programs consisting of multiple paths of control for economic reasons. Such programs allow concurrent execution of individual paths of control, thereby making efficient use of system resources.

The need for programs consisting of multiple paths of control on methodological grounds is not immediately apparent. Programs constructed using procedural abstractions are suitable for solving programs which have a hierarchical structure [Fisher-72]. In essence, such problems are solved by decomposing the original problem into logically separate parts, and implementing each of them separately as procedures or functions. Some problems however do not have such a hierarchical structure. They may instead require several interdependant stages in processing, which in turn requires interleaved execution of multiple paths of control. It is possible to implement programs requiring multiple paths using techniques such as "program inversion" [Jackson-76a], and simulation of coroutine structures using procedures [Zelkowitz-78]. These techniques, however, do not preserve program structure and are therefore undesirable.

This section discusses two language structures: processes and coroutines, which can be used for representing temporal abstraction in a program.

Coroutines were introduced by Conway [Conway-63] to display a decomposition of a compiler in which the main components of the compiler had a symmetric rather than the traditional asymmetric calling relationship. These structures allow a programmer to define multiple paths of control in a program, and provide the programmer with means for controlling their execution. Transfer of control between these multiple paths is explicit.

Process structures were introduced into computer systems to distinguish between static programs and their dynamic behaviour and to be able to identify and understand the nature of coordination problems that arise when several programs are executed in parallel [Freeman-73]. In languages, these structures were introduced mainly to allow a programmer to define several paths of control in a program which can be executed concurrently. The process structure provides for the definition of each process independently of any other co-existing process. Tranfer of control between processes is transparent.

Many of the features desirable in process and coroutine structures are similar to features desirable in procedures. It is desirable, for example, to allow processes and coroutines to have parameters, to allow free variables within process and coroutine bodies, and to allow several activations of coroutine and process structures to exist simultaneously.

The main difference between them is in the semantics of their call statements. In a procedure call, control is transferred to the called procedure, the caller becomes passive and the called procedure is activated. A procedure call does not involve creation of a new path of control. Thus a program constructed using only procedures consists of a single path of control.

Coroutine structures require two types of call statements:

1) An initial call statement which creates a new path of control for the called coroutine, but does not involve a transfer of control.

2) A resume call statement which transfers control to a path. This path is resumed at its last point of interruption. The coroutine resumed becomes active and the caller becomes passive.

Unlike procedures therefore, a coroutine structure allows a program to consist of multiple paths of activity. Like procedures, however, there is only a single path which is active at a time.

Process call statements involve both creation and activation of a new path of control of the called process. Unlike procedures and coroutines however, both the called and the calling process remain simultaneously active.

Processes need means for synchronising their activities. The two main motivations for these facilities are: mutual exclusion, and interprocess communication [Dijkstra-68a], [Habermann-72]. Facilities for synchronising processes have already been discussed in section 2.2.1.2.

2.3 IMPLEMENTATION CONSIDERATIONS

Decisions made in language design influence the choice of implementation strategies or policies which in turn directly influence characteristics such as efficiency, run time support, portability, and availability of programs implemented in the language. Implementation requirements of major importance, especially in real-time/systems are: availability, run-time support, run-time efficiency, and portability. This section discusses language design decisions which influence the choice of implementation strategies and thus influence the characteristics of programs implemented in the language.

2.3.1 INFORMATION BINDING

Programs are composed of a sequence of symbols (or names), which are used to refer to particular objects, for example operators and data. The set of symbols used in a program can be classified into two categories: predefined symbols, and user defined symbols. Predefined symbols generally have fixed meaning which is defined by an implementation of the language. User defined symbols on the other hand, must have meaning assigned to them in a program. The term information binding refers to the binding of attributes which give meaning to the symbols in a program.

Elson [Elson-73] has classified the attributes required by the symbols used in programs into six categories: name-declaration, declaration-declaration, declaration-object, declaration-constraint, declaration-description, and location-value bindings, and distinguished five binding times: program creation, linkage-edit, load, call and the execution reference at which they can occur. Table 2.2 shows examples of the binding times of typical language contructs in some commonly used programming languages.

The time when information is bound plays a major role in determining:

1) The method to be used in program execution.
2) The choice of the storage management policy in implementing a language.

These impementation strategies (or policies) have a great impact on the run-time efficiency and run-time support required by programs implemented in the language. The following subsections examine the impact of these implementation strategies or policies on the characteristics of programs. In addition they highlight binding time decisions which affect the choice of strategy.

2.3.1.1 METHODS OF PROGRAM EXECUTION

Two alternate methods for executing programs implemented in a particular language are:

1) Interpretation: direct simulation of statements of the

BINDING \ TIME	Program creation	Linkage	Load	Call	Execution reference
Name-declaration	ALGOL names			APL names	
Declaration-declaration		FORTRAN common		Argument-parameter binding	
Declaration-object			FORTRAN data	ALGOL local data	PL/1 "based" data
Declaration-constraint	All data				
Declaration-Description	ALGOL names				APL NAMES
Location-value	Constants				

Table 2.2 Attribute binding times in some common programming languages (Elson-73).

programming language.

2) Compilation: translation and subsequent execution of a machine code version of the program.

The fundamental factor determining the method used to execute programs in a language is the binding time of symbols used in a program. A language construct according to Wegner [Wegner-68], Elson [Elson-73] is compilable if all its required information bindings are determined before the construct is encountered during program execution. On a larger scale, a language is compilable if a majority of the language constructs are bound before execution reference time.

Interpretation versus compilation is a typical example of the flexibility-efficiency trade-offs involved in language design. Flexibility of features requires postponement of bindings until execution reference time. This requires the programs to be interpreted. The method chosen to execute programs implemented in a language influences both the run-time efficiency as well as the run-time support required by the programs. Execution of a program via interpretation does not require a separate translation phase before execution can begin. In general, however, the combined time required for the translation and execution phases of program executed by compilation is much less than the time taken to execute the program via interpretation. In addition, since a translated program can be repeatedly executed without recompilation, the potential saving can be even greater [Wegner-68]. Execution of a program via interpretation requires an elaborate run-time system in the form of an interpreter. In addition to adding overhead to the execution of a program, this interpreter occupies space that could otherwise be utilized by the user's program.

2.3.1.2 STORAGE MANAGEMENT POLICY

The storage management policy used in the implementation of a language plays an important part in determining the efficiency of implementation of the language. Three commonly used storage management policies in increasing order of flexibility and implementation complexity and decreasing order of implementation efficiency are:

1) Static storage management policies.
2) Overlay storage management policies.
3) Dynamic storage management policies.

A survey of these storage management policies is contained in [Griffiths-76a], [Aho-77]. Static and overlay storage management policies are similar in the sense that in both these methods storage is allocated statically at compile time. The main difference between them is the manner in which space is allocated for the different subprogram units. In static storage management, space allocated for different subprogram units is disjoint. On the other hand space allocated for different subrogram units in overlay storage management policies is non-disjoint. In dynamic storage management policies space is allocated on demand at run-time. Two important classes of dynamic storage management policies are:

1) Stack based dynamic storage management.

2) Heap based dynamic storage management.

The main difference between these two policies is the manner in which allocated storage is released. In stack based dynamic storage management, units of space allocated are deallocated in a last-in first-out manner. This rule is not observed in heap based management policies.

Two important factors which influence the choice of the storage management policy to be used in an implementation of a langauage are: binding time, and rules of lifetimes of objects. Static and overlay policies require that the amount of storage to be allocated for objects in a program be known at compile time. This requires that bindings, such as declaration-object and declaration-description bindings which are often postponed until run-time in many compiled languages, be done at compile time. Postponement of these bindings until run time generally requires dynamic storage management. Rules governing the lifetime of objects are a major factor in choosing between heap and stack based dynamic storage management policies. In general, objects which have a lifetime longer than the subprogram (or unit of allocation) in which they are created require heap based storage management. On the other hand, objects whose lifetime is the same as the subprogram unit in which they are created can be allocated on a stack.

The choice of storage management policy plays an important part in determining the flexibility of features and implementation efficiency of a language. Later binding times permitted by dynamic storage management policies allow a greater freedom in design of features in a language, in addition to allowing optimal use of storage space. Static and overlay policies produce programs whose execution time performance is optimal, at the cost of less flexible features. Static management policies do not allocate storage optimally. Storage allocation in bottom up overlay policies [Bochmann-78] have performance between dynamic and static allocation policies.

2.3.2. SCHEDULING AND SYNCHRONIZATION

Languages that include process structures discussed in section 2.2.2.4 require varying degrees of run time support for features such as creation, destruction, scheduling and synchronization. For example, the process structures in PL/1 and CONCURRENT PASCAL require different degrees of run time support. In this section we investigate the run time support required by process structures in languages such as MODULA and CONCURRENT PASCAL.

The two aspects of process structures in these languages requiring run time support are: scheduling and synchronization of processes. A program in these languages consists of a set of concurrently executing sequential programs called processes, each of which runs on its own virtual processor. If the number of processors in the system is equal to the number of processes in the program, then each process can run on its own private physical processor. This is not usually the case. In practice the number of processes is far greater than the number of processors in the system. It is therefore necessary to have a mechanism by which a large number of processes can share the limited number of physical processors. This mechanism is called a scheduler. The scheduler's task is to interleave the execution of the processes, thus giving the appearance that each process has its own physical processor.

In languages having process structures, the algorithm and data structures used by the scheduler to determine the order of interleaving of processes is fixed by the implementation [Brinch Hansen-77b], [Wirth-77b]. These algorithms and data structures not only form a run time system for programs implemented in the language, but in addition represent decisions that are preempted by the implementation of the language.

2.3.3 INPUT/OUTPUT

Programming languages in general do not provide structures for programming peripheral devices. Instead, they generally provide sequential I/O statements for expressing the I/O requirements of a program. These languages therefore require an elaborate run time system which implements these I/O routines. These run time systems not only degrade the portability of programs in the language, they also make these programs extremely configuration dependent. For example, adding an I/O device to a system requires changes to the run time system.

Even languages which provide structures for programming peripheral devices, such as device monitors [Ravn-80] (a proposed extension to CONCURRENT PASCAL) or device processes in MODULA may require run time systems. Consider for example the device processes in MODULA. The representation of a device driver as a process requires run time support.

2.3.4 RUN TIME PROGRAM ORGANISATION

In compiled languages there are two ways in which the run time representation of the program can be organised:

1) reentrant.

2) non-reentrant.

These run time organisations of programs can be clearly expressed in terms of Wegner's [Wegner-68] data structure model of program execution.

In the data structure model of program execution, the execution of a program is represented by a series of snapshots called instantaneous descriptions. Each snapshot consists of:

1) A program part, which consists of the executable statements of the program to be executed.

2) A data part which represents the data on which the program operates.

3) A stateword representing the information in the processing unit of the computer.

In programs organised non-reentrantly, the program part of the instantaneous description is modified during its execution. Futhermore, each activation of a program requires creation of a new instantaneous description of a program. In programs organised reentrantly, the program part of the instantaneous description is not modified during program execution. Each instance of use of such a program, called an activation, requires creation of a stateword and data part of the instantaneous description. It does not require creation of the program part of the instantaneous description, thus allowing several instances of activations of a program to share a single program part.

Reentrant organisation allows programs to be organised much more efficiently than programs organised non-reentrantly. Such an organisation is most effective when a language has features such as recursive procedures, or has structures such as processes and coroutines.

CHAPTER 3

DESIGN OF RT-PASCAL

3.0 INTRODUCTION

This chapter discusses a framework and outlines a philosophy for the design of a set of language structures to support the implementation of real-time/system programs. Programs implemented in the language incorporating these structures are assumed to run on "small computers" in a "bare machine" environment. The basic framework for the design of structures in a language is provided by consideration of the application area. This framework is often augmented by other properties sought in programming languages. The philosophy suggests development of a functionally complete language which allows implementation of a system without recourse to the use of assembler language. It also suggests avoiding language structures which require run time support, and preempt system design decisions. This chapter applies this philosophy in making language design decisions concerning structures for multiprogramming, and for performing I/O. The structures specifically designed for these purposes are the "coprocess" and "interrupt handler" structures. They are introduced in this chapter and discussed in detail in the following chapters.

The chapter is organised as follows. Section 3.1 discusses considerations specific to the application area. It characterizes the features and implementation requirements of real-time/system programs, and highlights the special requirements of languages for implementing programs on small computers in a bare machine environment. Providing these features and satisfying the implementation requirements is the major design goal of RT-PASCAL. The other design goals of RT-PASCAL concern preemption of system design decisions, and reliability. These design goals, which form the framework or basis for the design of RT-PASCAL, are discussed in section 3.2, which in addition identifies language design decisions which significantly affect these design goals. Section 3.2 also investigates the approach taken by extant system implementation languages in making language design decisions concerning structures for multiprogramming and for performing real-time I/O. It then outlines our approach in making these language design decisions. Sections 3.4, and 3.5 discuss language issues in implementing multiprogramming systems and in performing real-time I/O. These sections also introduce two new language structures, "coprocesses" and "interrupt handlers" which are specially designed for these purposes.

3.1 APPLICATION AREA PROGRAM CHARACTERISTICS

The application area for which RT-PASCAL is designed is real-time/system applications which are to be implemented on small computer architectures as characterised by the present generation of mini and micro computers. By this statement we wish to cover single application or dedicated systems which are the primary applications of this class of computer architecture.

Wirth characterises real-time applications as multiprogramming applications in which an additional factor, execution time, is taken into account in program construction [Wirth-77c]. He describes a multiprogram as a set of sequential programs each of which can be executed concurrently. In conclusion, he suggests that languages intended for real-time applications should include: structures for sequential programming, structures for multiprogramming, and structures for performing real time I/O. Structures for sequential programming include data and sequential control structuring facilities as well as procedural, data, and control abstraction facilities. Structures for multiprogramming include structures such as processes and means for synchronising their activities. Real-time systems in general consist of a number of concurrent processes. Real-time control of I/O requires facilities for programming I/O devices. Examples of such facilities are device processes [Wirth-77a], device monitors [Ravn-80], and interrupt handlers

[Manacher-71].

Some authors distinguish between real-time and systems implementation languages on the basis that facilities for programming peripheral devices are not necessary for systems applications. For example CONCURRENT PASCAL (which is designed for systems applications) does not have facilities for peripheral devices. Various proposals [Ravn-80] to add these facilities to such language indicate their importance in systems applications.

The three major implementation requirements of real-time/systems applications are availability, implementation efficiency, and freedom from elaborate run-time support.

Implementation requirements concerning run time support and implementation efficiency are widely recognised. The need for these implementation requirements in system implementation languages has been discussed by [Lang-68], [Dreisbach-76] and [Freeman-73]. The application area addressed by RT-PASCAL emphasises the importance of these requirements. Firstly, the relatively low processing capabilities of the small target machines accentuate the need for implementation efficiency. Secondly, the class of applications, namely, dedicated or stand alone systems accentuates the need to avoid run time support.

The implementation requirement concerning availability is not often recognised. Real-time/system programs generally contain a set of time critical tasks. For reliable operation of the system it is important that these tasks be executed within a critical period of time. Languages intended for real-time/systems applications should not therefore include features that may cause a delay in executing these time critical tasks. This requirement, referred to as availability, is of vital importance in real-time/system applications.

3.2 DESIGN GOALS AND DESIGN PHILOSOPHY

The principal design objective of RT-PASCAL is to provide the requisite features and satisfy the implementation requirements of real-time/systems applications. We emphasise both functional completeness and conformity to the implementation requirements. By functional completeness we refer to the set of features required by this class of applications. This property allows coding of a complete system without recourse to assembler language thereby greatly enhancing the system portability of programs implemented in the language. The importance of the implementation requirements has already been discussed above.

The other design objectives of RT-PASCAL concern preemption of system design decisions, and language aids to reliability. In real-time/systems applications, design decisions can significantly affect the performance of a system. A system implementation language should therefore avoid preempting system design decisions. These decisions should be left to the system designer whenever possible. Examples of system design decisions in real-time/system applications are decisions concerning scheduling and synchronisation. Reliability of programs is affected by several issues. Languages should assist the development of reliable programs by avoiding as far as possible constructs which are prone to errors, and by introducing constructs which allow a language processor to detect errors. Examples of such constructs are pointer variables, and type definitional facilities respectively.

A variety of different approaches have been taken in designing real-time/systems implementation languages. These approaches are typified by BLISS [Wulf-71], ASTRA [Hegering-76], EUCLID, MODULA, CONCURRENT PASCAL, and most recently ADA. While each of these languages has features which support the design of real-time/systems applications, none contains the full set of characteristics sought in this work.

The two language design decisions addressed by this work which significantly affect our design goals are: decisions concerning structures for multiprogramming, and decisions concerning structures for programming peripheral devices. The way in which these two subjects are addressed classifies extant real-time/systems languages into two categories: those that provide exclusive use of a high level language for systems implementation, and those that require partial implementation of the system in assembly language.

Examples of languages in the first category are CONCURRENT PASCAL, MODULA, and ADA. These languages provide the structures necessary for systems implementation without any assembly code. These structures however, depend on the existence of a run time facility, which consists, in the case of MODULA and CONCURRENT PASCAL, of small [Wirth-77b] and medium [Brinch Hansen-77b] sized kernels respectively. The implementation of this run time system implies an extensive set of design decisions, which are therefore preempted from the system designer. Furthermore, this run time system cannot be implemented using the structures provided by the language. It has to be implemented in assembly language.

The second category of languages, which include ASTRA and EUCLID, do not require run time support and consequently do not preempt system design decisions. They do not, however, have structures to represent multiple paths of control in a program and structures to perform I/O. These languages therefore require the use of assembly language code for most real-time/system work.

The philosophy used in the design of the language structures presented here represents a third approach. As the language is required to run on a sequential machine architecture, all of its structures are essentially sequential. Structures are provided for expressing a program as a set of sequential control paths of control; and means are provided for controlling these control paths. In an implementation of a multiprogram, policy decisions concerning scheduling, synchronisation, and I/O processing are entirely in the hands of the programmer as it is in the case of languages in the second category and in assembly language work. These structures therefore provide the programmer with high level language constructs which reflect the operations concerned, and hide the details not required for the expression of the system functions.

A third language design decision which significantly affects our design goals is binding time. This question is not specifically treated in this work. It should be noted that the bare machine environment and our design goals concerning run time support suggest that the language should not imply dynamic storage management. Furthermore, our design goal concerning preemption of system design decisions suggests the need to provide the programmer with means for managing the allocation of storage. In this work we will discuss whenever possible an implementation in which storage is managed using Bochmann's bottom up storage management policy. This allows an efficient implementation which we expect will enable us to gain some practical experience on the utility of the concepts developed here. It should be noted that the choice of storage management policy affects the design of the structures addressed in this work. Their design therefore proceeds in two stages. In stage one, structures are developed without regard to their implementation. In stage two the design of the structures is examined with the view of implementing them in an environment in which storage is allocated statically.

3.3 STRUCTURES FOR MULTIPROGRAMMING

Implementation of a multiprogram requires a system implementer to provide an implementation of facilities for scheduling the execution of paths of control, and facilities for synchronising the activities of concurrently executing paths of control. Low level synchronisation required for implementation of these facilities is greatly dependent on the environment in which the multiprogram runs. In a uniprocessor environment, low level synchronisation can be achieved by interrupt control. A multiprocessor environment, however, also requires hardware support in the form of a Test and Set instruction or external logic which can, under program control, put a processor in a hold state. In this work, we provide constructs which can implement low level synchronisation in a uniprocessor environment. In other words, we provide facilities for interrupt control. No special constructs are provided for implementing low level synchronisation in a multiprocessor environment. In general, mini and microprocessors do not have any special functions for this purpose. It is therefore inappropriate to provide language structures which presume the existence of these functions.

Implementation of a multiprogram in a language in which there is no run time support implies that the software which implements multiprogramming must be explicitly programmed by the system implementer. This task is greatly facilitated if the system implementation language has structures which represent paths of control, and the operations which are characteristically required for implementing the structures for scheduling and synchronisation facilities of a multiprogram.

Implementation of the scheduling facility requires that the system implementation language possess means for representing paths of control in a program, and means for transferring control between them. Furthermore, the implementation of a scheduling facility for a multiprogram which is to run in the environment described above, is usually supported in hardware by a "timer interrupt". It is therefore essential that the implementation language possess structures for processing these interrupts. This language structure should have the capability of transferring control to a control path other than the one it interrupted.

Implementation of synchronisation facilities requires that the systems implementation language provide means for accessing (synchronisation) variables, which are in general shared among several (concurrently executing) paths of control. The principles of structured programming, especially those relating to data abstraction, suggest the implementation of certain (synchronisation) operations which are applicable to these variables. Some characteristics of these operations are: they are accessed by several control paths and therefore may be entered simultaneously, and their implementation generally involves transfer of control to a control path other than the one that called it. The former characteristic requires that the language allow the structure defining a control path to access globally defined structures, and that procedures be implemented reentrantly. The latter characteristic requires that a subsequent transfer of control to the original control path cause its execution to resume at the point where it relinquished control in the procedure.

Finally, implementation of scheduling and synchronisation facilities of a multiprogramming system in a language having the structures described above requires that the language have one additional facility: the ability to specify the execution of a procedure on inter path transfers of control. More precisely, it is necessary to provide an instruction by which transfers between control paths may be accompanied by invocation of a procedure. In these instructions control is transferred to the target control path. Its execution, however, is preceded by execution of the procedure.

RT-PASCAL satisfies all these requirements of a system implementation language which assumes no run time support. RT-PASCAL is a block structured sequential programming language in the tradition of PASCAL. In addition to PASCAL's structures, however, RT-PASCAL supports the implementation of a multiprogram by providing structures for defining programs consisting of multiple paths of control, facilities for transferring control between them, and means for processing the hardware interrupts in a system.

The structure which implements the sequential control path in RT-PASCAL is the "coprocess" structure. The name has been chosen to reflect its affinity to the process concept in [Wirth-77a], and [Brinch Hansen-75a], [Brinch Hansen-77a] with modifications which are reminiscent of characteristics of coroutines. The structure is developed in detail in chapter 4. The following paragraphs highlights the central idea of the coprocess concept.

The coprocess structure is essentially a sequential control structure. Like a coroutine, it allows the definition of a program consisting of several paths of control, and the means for transferring control between them. The main difference between the coroutine and coprocess structures is in the semantics of a procedure call. In the case of coroutines, transfer of control by a procedure to a control path other than the one that called it causes the procedure to be terminated. Subsequent transfer of control resumes the original control path at the statement following the procedure call. In the case of coprocesses and processes, however, the procedure activation is retained. Relinquishing control within a procedure does not cause the procedure to be exited, and subsequent transfer of control (explicit in the case of coprocesses, and implicit in the case of processes) resumes the original control path at the point where transfer of control occurred within the procedure.

The structure provided for processing the interrupts in a system in RT-PASCAL is the interrupt handler. The hardware interrupts in a system generally serve two functions: "device interrupts" act as mechanisms for making efficient use of system resources, and "timer interrupts" serve the scheduling function mentioned above. In its role as a language structure for processing device interrupts, the interrupt handler structure allows the programmer to specify return of control to the interrupted path of control after processing a interrupt. In its role as a language structure for processing timer interrupts, the interrupt handler structure allows the programmer to specify transfer of control to a control path other than the one that it interrupted. The interrupt handler structure is developed in detail in chapter 5. The following section on I/O highlights some of its essential concepts.

3.4 INPUT/OUTPUT

The absence of run time support assumed in RT-PASCAL implies that it cannot provide abstract operations such as READ and WRITE for performing I/O. This in turn implies that language structures must be provided for programming the peripheral devices in a system. A language structure essential to the programming of peripheral device is a facility for accessing the set of registers shared between the device and the CPU. Language structures for this purpose, which are applicable to a large class of mini and micro computers, have been discussed by Wirth [Wirth-77a]. They are not further discussed in this work. More important, especially from the point of view of making safe and efficient use of system resources, are language structures for processing and controlling the interrupts in a system. These facilities are provided by the "interrupt handler" structure and instructions which explicitly control the occurrence of interrupts in a system. Although an interrupt concept for programming peripheral devices is common in most computer architectures, its details of operation are often varied. The interrupt mechanism discussed in chapter 5 is applicable to a large class of mini and micro computer architectures. Chapter 5 develops in detail the interrupt handling facility for this class of interrupt structures. The following paragraph briefly presents the concepts of an interrupt in RT-PASCAL.

An interrupt in RT-PASCAL is viewed as a call to a procedure called an "interrupt handler". This procedure may be conceived as being invoked by the executing control path. As in the case of procedures, exiting an interrupt handler via a return statement returns to the interrupted control path. Unlike a normal procedure however, relinquishing control to a control path other than the one in whose path it is invoked, terminates the interrupt handler. These explicit transfers of control are necessary for handling the scheduling function served by timer interrupts. There are other minor difference in implementation between procedures and interrupt handlers. These differences are discussed in chapter 5.

CHAPTER 4

COPROCESS STRUCTURES

4.0 INTRODUCTION

This chapter develops a language structure called a "coprocess", which is well suited for implementing multiprogramming systems at the system implementation level. Some of the essential facilities required for implementing multiprogramming systems provided by this structure are: representations for sequential paths of control, and facilities for controlling their execution. Implementation of a language ~~incorporating this structure on a~~ sequential machine characterised by most present day computers does not require a run time system. The structure therefore provides the system implementer with a tool for implementing a multiprogram which neither requires run time support nor preempts system design decisions. In addition it goes a long way towards eliminating the use of assembly language in system implementation.

This chapter and the next make extensive use of the contour model of block structured processes for describing the semantics of function modules. Section 4.1 of this chapter therefore briefly discusses the essential characteristics of this model.

The rest of this chapter is organised as follows. Section 4.2 discusses language structures which can be used for implementing multiprogramming at the system design level. It examines the use of coroutine structuree for this purpose, and identifies an important area in which it is inadequate. It then introduces the coprocess structure, and demonstrates its adequacy in this area. Section 4.3 discusses the design of the coprocess structure. It first develops the features desirable in this structure, and then discusses its implementation in an environment in which storage is allocated statically.

4.1 THE CONTOUR MODEL OF BLOCK STRUCTURED PROCESSES

Information structure models [Wegner-68], [Wegner-71] characterize a process as a time invariant "algorithm" and a time varying "record of execution". More precisely, they describe a process as a sequence of "snapshots" or "instantaneous descriptions", each of which is composed of these two components: the time invariant algorithm and the record of execution, as shown in figure 4.1.

The contour model of block structured processes forms a subset of Wegner's information structure model of computational processes [Johnston-71]. The concept of a process covered by this model is restricted to block structured processes. More specifically this implies that both the record of execution and the algorithm have an

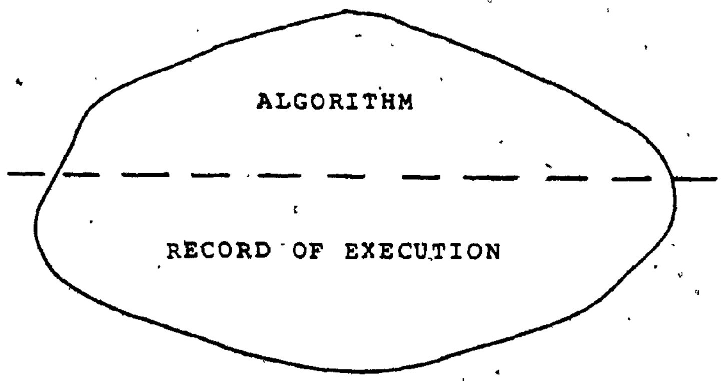


Figure 4.1 A "snapshot" in the information structure model of process execution

underlying structure that can be characterised as a nested contour structure as shown in figure 4.2.

The contour model, like the general information structure model, also has two components: the algorithm, and the record of execution. The algorithm consists of a fixed reentrant pure procedure, and the record of execution consists of the variable data cells. This record contains several "processors", which are cells that control the execution of the algorithm.

The term processor is used here in a logical sense. In this context it is defined as a data item rather than a hardware CPU. In an actual implementation a processor would need to reside in a CPU in order to execute. Stripped to the

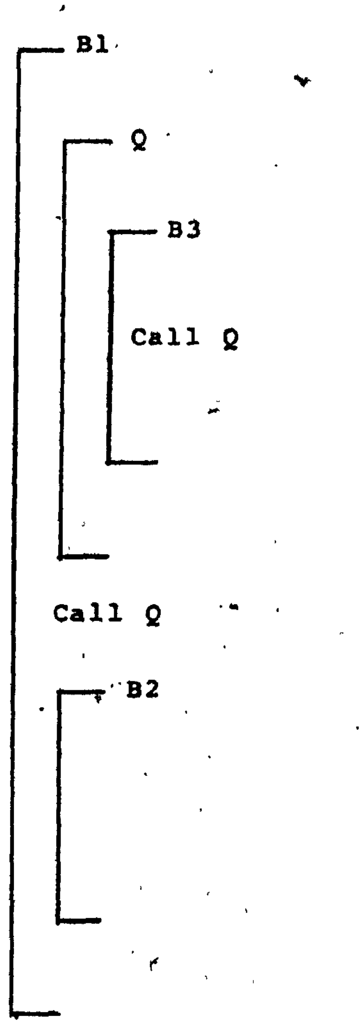


Figure 4.2a Block structured program. B1, B2, B3 are blocks Q is a procedure.

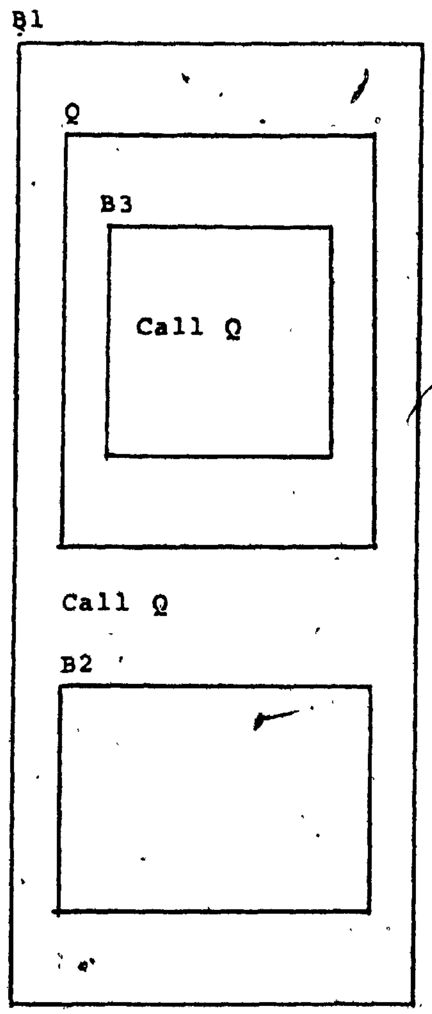


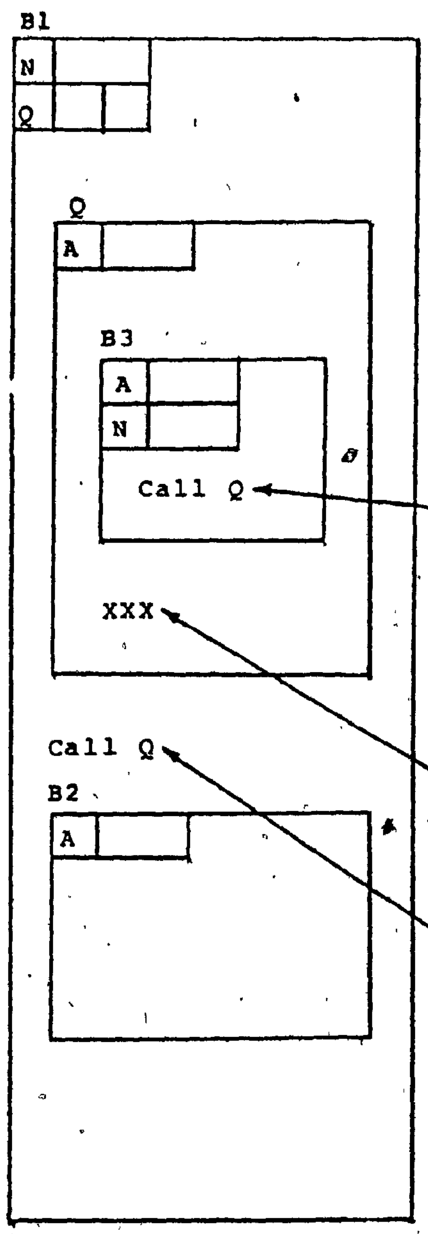
Figure 4.2b Nested contour structure of program in figure 4.2a.

essentials, a processor consists of an instruction pointer IP which points to an instruction in the algorithm, an environment pointer EP which points to an accessing environment containing the data cells in the record of execution, and a return pointer RP. This return pointer, RP, normally points to the processor which created it.

Every contour contains as an integral part of itself a DECLARATION ARRAY of named cells. Objects in the declaration array (analogous to local variables) are those that are declared within the contour. As an example, observe the upper left corner of each contour in figure 4.3. In an implementation of the contour model for representing the execution of programs, space not only has to be allocated for the declaration array but also for copies of the registers of the processor when it is not active.

Execution of the algorithm is achieved by having the instruction pointer scan the instructions in the algorithm step by step. Transfers of control involve a change in either the IP or both the IP and EP of the processor. When an identifier is encountered in the algorithm, the processor uses the environment pointer to build the address designated by the identifier in the record of execution.

The version of the contour model used in this thesis is an adaptation of the version used by Berry [Berry-71] to discuss the semantics of OREGANO. Although the version used here and Berry's version are semantically the same, small notational differences exist between them. For instance processors in Berry's version consist of an (IP, EP) pair, whereas in this version they consist of a triplet (IP, EP, RP). The RP in



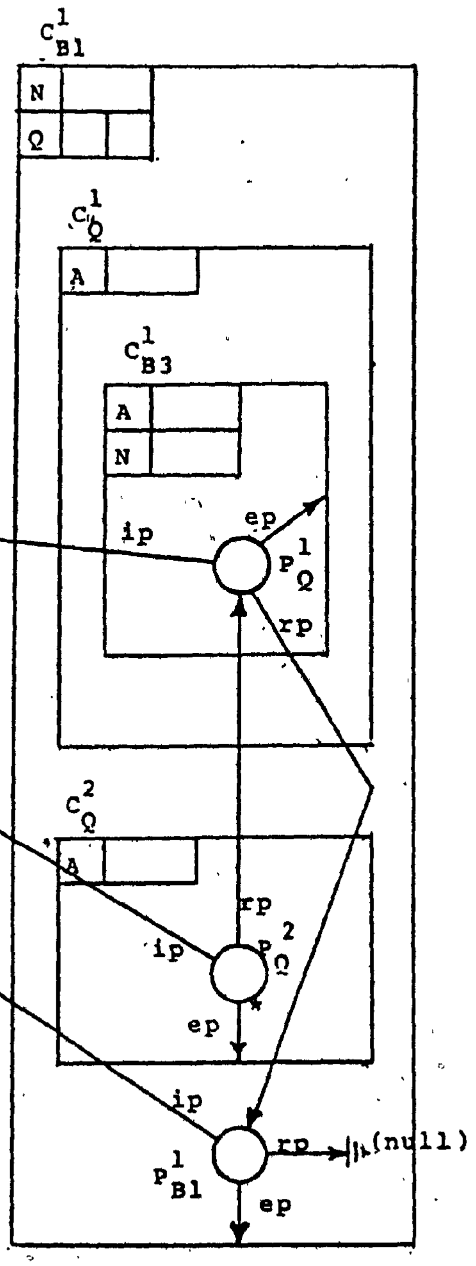


Figure 4.3a Algorithm

Figure 4.3b Record of execution

Example of a snapshot in the contour model. In figure 4.3b, processor P^1_{B1} has invoked procedure Q, and Q has been called recursively by block B3.
Note: The circles represent processors. The symbol '*' indicates a processor which is active.

Berry's version forms part of the contour. Other notational differences made for sake of clarity may be apparent to the reader.

There is a great deal of similarity between the stack model which has been extensively used to describe the semantics of block structured languages such as ALGOL-60 and PASCAL, and the contour model presented here. Of interest with regard to subsequent discussion is the manner in which they treat procedure activations. A procedure activation is called an activation record [Wegner-68] in the stack model and a contour in the contour model. The main difference between them is in the manner in which procedure activations are deleted on procedure exit. In the stack model the activation record is unconditionally deleted on procedure exit, whereas the contour model uses a reference counted storage management policy to deallocate procedure activations.

4.2 STRUCTURES FOR MULTIPROGRAMMING AT THE SYSTEM IMPLEMENTATION LEVEL

4.2.1 COROUTINE STRUCTURES

Like processes, coroutine structures provide a representation for multiple paths of control in a program. Unlike processes however, only one path of control is active at any particular time. Coroutines, therefore, do not specify concurrent execution of individual paths of control at the language level. Assuming the existence of a structure for processing timer interrupts, coroutines can be used to implement concurrent execution of individual paths of control at the system implementation level. Such concurrent execution is achieved by implementing a scheduler which will schedule more than one path of control simultaneously. Control is transferred to the scheduler either by a control path or by an interrupt handler. As in assembly language language work, ensuring that a coroutine or the scheduler is not entered by more than one processor is the responsibility of the programmer. Coroutines can therefore be used as a basis of implementing concurrent execution of control paths at the system implementation level.

The use of coroutines as a program structure has not been exploited by language designers. To our knowledge only a few languages such as BLISS, OREGANO [Berry-71], and SIMULA include coroutine structures. The major purposes of coroutine structures in these languages are to provide the language with a program structuring facility, or to provide the language with a structure which can be used in simulation applications. We have therefore observed certain deficiencies in the use of coroutines for implementating multiprogramming systems.

In particular we refer to the effect of a "resume" statement in a procedure which has been called by a coroutine. We are unable to discern from the limited information we have on BLISS the precise effect of such a statement. In OREGANO and SIMULA however, it is clear that this statement causes the coroutine specified in the "resume" statement to be continued and the procedure to be exited. Subsequent resumption of the original coroutine will cause this coroutine to be resumed in the statement following the procedure call.

As an illustration consider the skeleton program shown in

```
1         Program EX41;
2             var R0, R1, R2: path-descriptor;
3             procedure Q;
4             begin
                  ...
5                 resume(R2);
                  ...
6             end;
7             coroutine X;
8             begin
                  ...
9                 call Q;
                  ...
10            end;
11            coroutine Y;
12            begin
                  ...
13                resume (R1);
                  ...
14            end;
15        begin
              ...
16            start-up(X,R1,EX41);
17            start-up(Y,R2,EX41);
18            resume(R1);
              ...
19        end.
```

Example 4.1

example 4.1. In line 16 and 17 the main program initializes

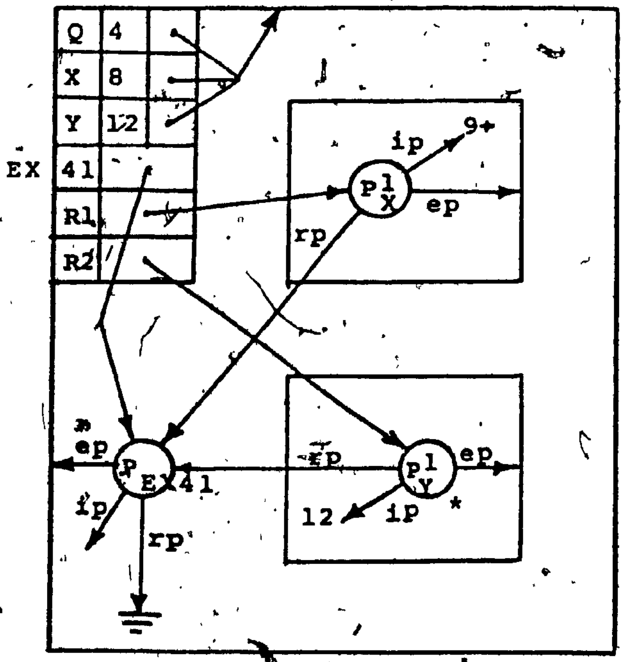


Figure 4.5 State of the record of execution after procedure Q resumes processor P_Y^1 in line 5 of example 4.1.

state of record of execution at this point is as shown in figure 4.5. Observing figure 4.5 it is apparent that the resume call in line 5 not only causes control to be transferred in coroutine Y but also causes the deletion of the contour and processor of the procedure Q. The effect of this deletion is that, when the coroutine X is resumed in line 12, control is transferred to the statement following the call to procedure Q, i.e. to the statement following in line 8.

Contrast these aspects of coroutine semantics with those of processes, illustrated with the example 4.2, expressed in

```
 1      program EX42;

 2         monitor A;
 3             var C : condition;
               ...

 4             procedure Q;
 5             begin
                 ...
                 C.wait;
                 ...
 7             end;

 8         begin
               ...
 9         end;

10         process X;
11         begin
               ...
12             call Q;
               ...
13         end;

14         process Y;
15         begin
               ...
16         end;

           ...
           (*other process declarations*)
           ...

17      begin (*main program*)
           ...
18         start(X);
19         start(Y);
           ...
20      end.
```

Example 4.2

CONCURRENT PASCAL. In this case, as before, the main program initializes two processes X and Y (in lines 15 and 16

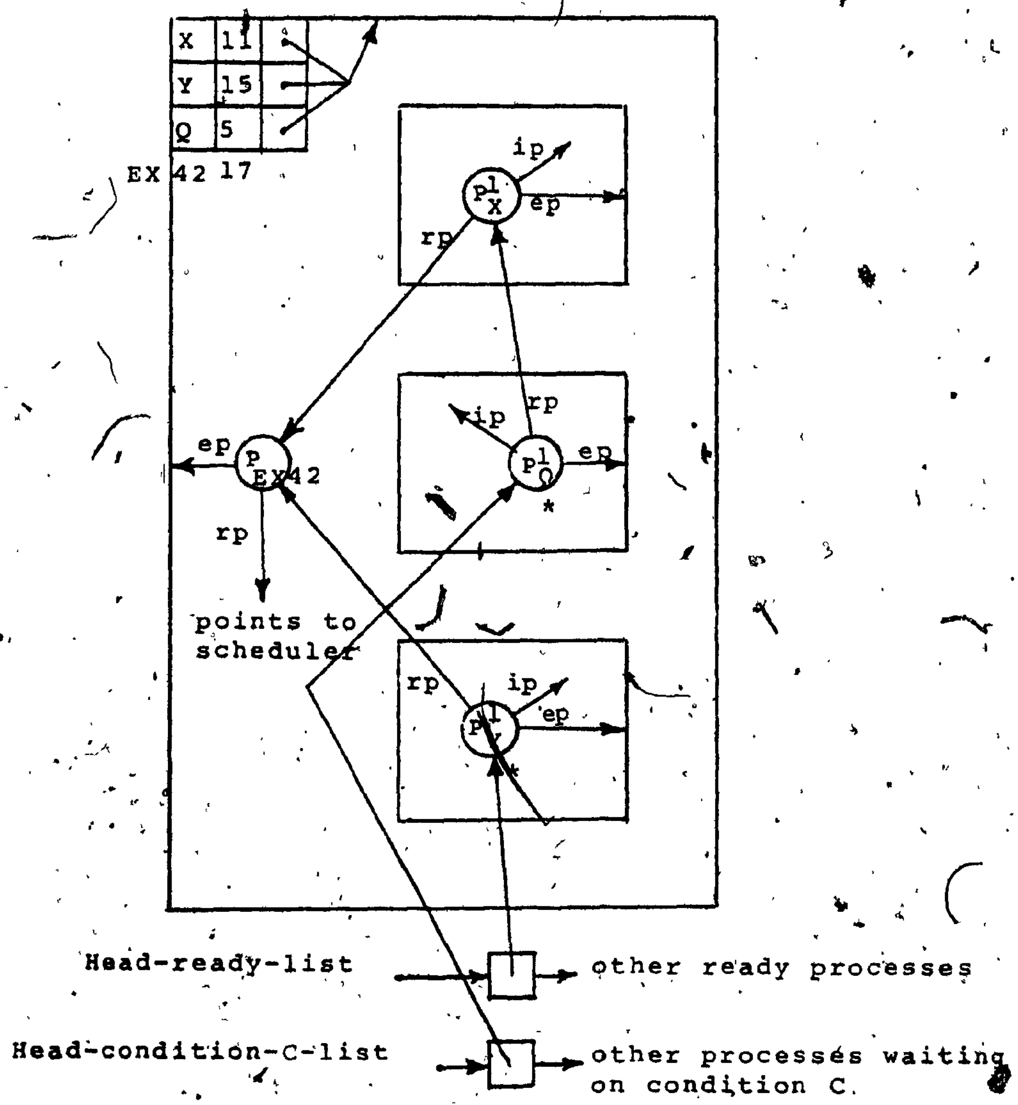


Figure 4.6. State of the record of execution when process P waits on condition C in line 6 of example 4.2.

respectively). When the main program completes execution, the scheduler will transfer control to the process X. During its execution process X calls the monitor procedure Q (in line 12). When process X executes the WAIT operation (in line 6), let us assume that it has to wait for condition C to be satisfied. It is reasonable to expect that the implementation of the WAIT operation not only makes process X wait, but also transfers control to another process. The state of execution record at this point is as shown in figure 4.6. Observe that when condition C is satisfied and control is transferred again to process X, it will continue execution at the statement following the WAIT statement within the procedure (in line 6).

Examining these two examples we make the following observations:

1) Both programs have similar structure.
2) The WAIT operation in the second example can be viewed as executing an operation analogous to a resume, following some housekeeping operations.
3) Observing the effects of the RESUME and WAIT statements in figures 4.5 and 4.6 respectively, we notice that the contour and processor of the procedure Q is deleted in the case of the coroutines and is retained in the case of the processes. This deletion is attributed directly to the semantics of the coroutine structures.

We therefore conclude that although coroutine structures form a basis for implementing multiprogramming at the system implementation level, their semantics are not adequate for implementing synchronization primitives (which in general include transfer of control outside the coroutine) at the system implementation level.

4.2.2 COPROCESS STRUCTURES

With reference to the discussion of the previous section, it is apparent that coroutine structures provide some of the facilities required for implementing multiprogramming system at the system implementation level. In addition, it showed that their semantics are inadequate for implementing synchronization primitives in the language. Since it is desirable to have the capability of implementing synchronization primitives at the system implementation level we introduce the concept of a "coprocess".

Some of the properties of coprocess structures are similar to those possessed by process and coroutine structures. Like coroutines and processes, coprocess structures provide representations for multiple paths of control in a program. Unlike processes, coprocesses do not specify concurrent execution of individual paths of control at the language level. Coprocess structures however, like coroutines, can be used to implement concurrent execution of individual paths of control at the system implementation level. Finally, like processes and unlike coroutines, coprocesses can be

```
program EX43;
    var R1, R2: path-descriptor;
    procedure Q;
    begin
        ...
        resume(R2);
        ...
    end;
    coprocess X;
    begin
    ...
    call Q;
    ...
    end;
    coprocess Y;
    begin
        ...
        resume (R1);
        ...
    end;
begin
    ...
    start-up(X, R1, EX43);
    start-up(Y, R2, EX43);
    resume(R1);
    ...
end.
```

Example 4.3

represented by a sequence of dynamically nested activation records which are retained even when control is transferred outside the coprocess.

As an illustration of the essential difference between coroutine and coprocess structures, consider example 4.1, in which we replace the coroutines X and Y by coprocesses X and Y respectively as shown in example 4.3. The effect of the

RESUME statement in procedure Q (in line 5), is that control is transferred to coprocess Y. The state of the record of execution at this point is as shown in figure 4.7.

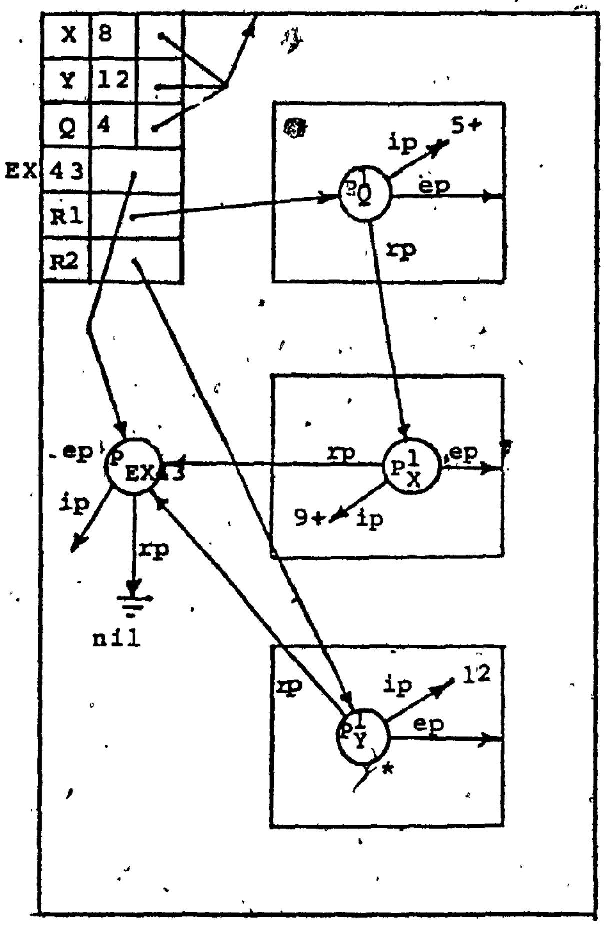


Figure 4.7 State of the record of execution after procedure Q transfers control to coprocess Y in line 5 of example 4.3.

Unlike the case of coroutines, the contour and processor of procedure Q is not deleted. Hence when coprocess X is resumed by coprocess Y (in line 13), control is transferred to the statement following the resume statement in procedure Q, and not at the statement following the call to procedure Q as happens in the case of coroutines.

We observed from example 4.2, discussed in the previous section that implementation of synchronization primitives in a high level language requires that execution of resume operations occurring within procedures should not cause termination of the procedure. Coprocess structures which provide this facility are therefore better suited than coroutines for implementing multiprogramming systems at the system implementation level.

4.3 DESIGN OF COPROCESS STRUCTURES

4.3.1 CONCEPTUAL DESIGN

Coprocess semantics can be viewed as an extension of coroutine semantics. It is therefore logical to discuss their semantics with respect to those of coroutines.

Like coroutines, coprocesses have two types of call statements:

1) A START-UP call statement, which creates a contour and a processor, and initializes the control path to refer to the processor of this instance of the coprocess.

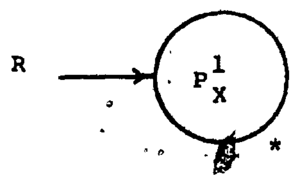


Figure 4.8a: State before processor P^1_X referred to by path descriptor R calls processor Q.

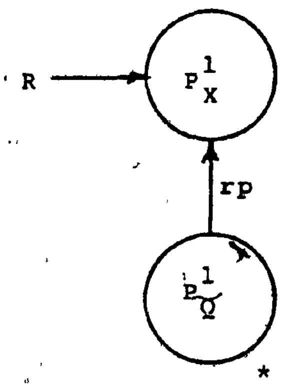


Figure 4.8b State after processor P^1_X referred to by path descriptor R calls processor Q.

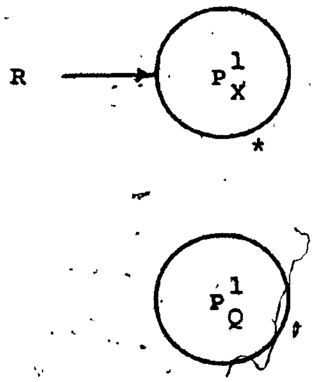


Figure 4.8c State after procedure Q is exited.

Note: As before circles represent processors. These figures show only the dynamic nesting of the processors.
The symbol '*' indicates processor which is active.

2) A RESUME call, which transfers control to the coprocess whose path is specified in the call.

The main difference between coroutines and coprocesses is in the semantics of the procedure call and return statements. In the case of coroutines, a procedure call statement involves the creation, initialization, and transfer of control to the processor of the called procedure. It does not update the path descriptor which controls the execution of the coroutine invoking the procedure. A return statement merely involves returning control to the dynamically nested processor. As a side effect, the contour of the returning procedure may be deleted if there is no external reference to the contour or to any element in the contour. These semantics of procedure call and return statements are illustrated in figures 4.8a, 4.8b and 4.8c.

Now compare the semantics of procedure call and return statements in the case of coprocesses. In this case the procedure call not only involves initialization, creation and transfer of control to the processor of the called procedure, but also involves updating the control path which controls the execution of the coprocesses to refer to the processor of the called procedure. These semantics are illustrated in figures 4.9a and 4.9b. These semantics allow resumption of a coprocess at its last point of exit, even when this is within a dynamically nested procedure. This is possible because a

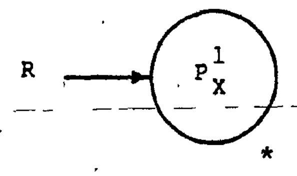


Figure 4.9a State before processor P_X^1 referred to by path descriptor R calls procedure Q.

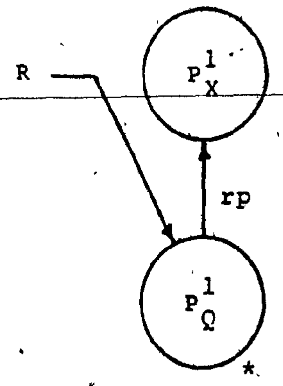


Figure 4.9b State after processor P_x^1 referred to by path descriptor R1 calls procedure Q.

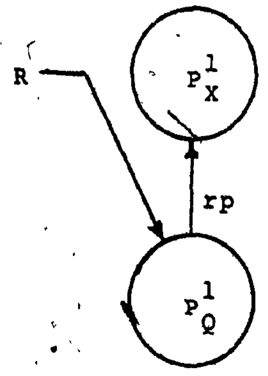


Figure 4.9c State after procedure Q exits via a resume instruction.

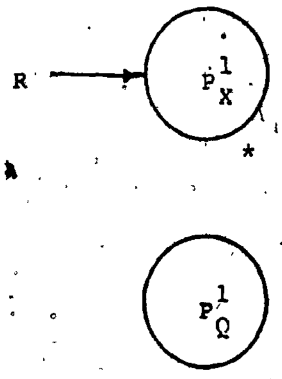


Figure 4.9d State after procedure Q exits via a return instruction.

reference to the procedure activation is retained on exit from the procedure via a resume statement. A consequence of the procedure call semantics is that the return statement not only involves transfer of control to the dynamically nested processor but also involves an update of the control path. These semantics of the resume and return instructions (occurring in a procedure) are shown in figure 4.9c and 4.9d.

The above discussion motivates a change in the processor structure of the contour model discussed in section 4.1. This change is motivated mainly by the need to update the path descriptor of the control path controlling the execution of the coprocesses. More specifically, it is necessary for a processor executing a call or return statement to have a reference to the path descriptor controlling the execution of a coprocess, to perform the updates required. We therefore add to the processor of the contour model, the field PD, which refers to the path descriptor controlling the execution of a coprocesses.

For correct operation of coprocesses, it is necessary that there exist only a single reference to any node (processor) in a control path. This requires us to prevent situations where more than one path descriptor refers to a node in a control path. Such situations arise when path descriptor variables are assigned values as shown in figure 4.10a, and when coprocesses are started-up as shown in figure 4.10b. Observing figure 4.10b it is apparent that the effect of the

```
procedure P(var S : SEMAPHORE);
var R : reference path descriptor;
begin
   disable;
   if S.STATE = 0
   then begin
      enque;
      R := SELECT;
      enable;
      resume(R);
   end
   else begin
      S.STATE := 0;
      enable;
   end;
end;
```

Example 4.4

1) The contour for the coprocess specified in the start-up operation is created and its parameters initialized.

2). A hidden path descriptor variable is created and initialized to refer to the processor created in step 3.

3) The processor for the coprocess is created and initialized as follows:

   1) IP, EP are initialized as in the case for procedures.

   2) RP is initialized to nil.

   3) PD is initialized to refer to the path descriptor variable created in step 2.

   4) RPD is initialized to refer to the path descriptor variable referred to by the reference path

descriptor variable in the start-up (i.e. RPD2).

4) The reference path descriptor variable specified in the start up statement is initialized to refer to the path descriptor variable created in step 2.

5) The processor executing the start up operation continues execution.

The form of the coprocess resume instruction is as follows:

resume(RA, P(arg-list))

where

RA is a reference path descriptor variable which indirectly refers to the processor to which control is to be transferred by the resume statement.

P is the name of an optional procedure which is to be executed in the path of the control path indirectly referred to by RPD, and arg-list is the list of arguments of the procedure.

The semantics of the coprocess resume statement are as follows:

1) If a procedure is specified then
   1) create and initialize a contour for the procedure (as in the case of procedures).
   2) Create a processor for the procedure and initialize it as follows:
      1) IP, EP and RPD are initialized as in the case of procedures.
      2) PD is initialized to refer to the path

descriptor specified in the resume statement.

3) RP is initialized to refer to the processor indirectly referred to by the reference path descriptor variable specified in the resume statement.

3) The hidden path descriptor variable referred to by the reference path descriptor variable specified in the resume statement is updated to point to the newly created processor.

2) Control is transferred to the processor indirectly referred to by the reference path descriptor variable.

The form of the coprocess return statement is as follows:

Return(p)

where

P is the name of an optional procedure which is to be executed in the path of the control path, to which control is transferred by the return operation. In other words, this provides the option of executing an exit procedure P, upon termination of a coprocess.

The semantics of the coprocess return statement are as follows:

1) If a procedure is specified in the return statement then

1) Same as 1.1 in the case of the resume statement.

2) Create a processor for the procedure and initialize it as follows:

1) IP and EP are initialized as in the case for procedures.

2) PD is initialized to refer to the hidden path descriptor variable which is referred to by the RPD field of the processor executing the resume instruction.

3) RP is initialized to refer to the processor indirectly referred to by the RPD field of the processor executing the return instruction.

4) RPD field is initialized to refer to the hidden path descriptor variable, which is referred to by the RPD field of the processor executing the return instruction.

3). The hidden path descriptor variable referred to by the RPD field of the processor executing the return statement is updated to refer to the newly created processor.

2) The processor executing the return instruction is set to a terminated state, and control is transferred to the processor indirectly referred to by the RPD field of the processor executing the return instruction.

The semantics of the procedure call statement are as follows:

1) Create and initialize the contour and the parameters of the called procedure.

2) Create a processor for the called procedure and initialize it as follows:

1) IP and EP are initialized to the procedure value (a procedure value is represented as an (IP, EP) pair).

2) RP is initialized to refer to the processor executing the call instruction.

3) RPD and PD are copied from the corresponding fields of the processor executing the call instruction.

3) Transfer control to the processor of the called procedure.

The semantics of the procedure return statement are as follows:

1) Update the path descriptor pointed to by the PD field of the processor executing the return statement to refer to the processor pointed to by the RP field of the processor executing the RETURN statment.

2) Transfer control to the processor pointed to by the RP field of processor executing the RETURN instruction.

4.3.2 ENVIRONMENTAL CONSIDERATIONS

The previous section makes no assumption on the environment in which the coprocess concept may be implemented. Obviously the coprocess concept may be implemented in different environments. The purpose of this section is to investigate the implications of implementing coprocesses in an environment in which storage is allocated using Bochmann's bottom up overlay storage management policy. Some restrictions on language features have already been discussed by [Bochmann-78]. This section discusses some additional restrictions which are particular to the coprocess concept and identifies some the structures discussed above which cannot be implemented using this policy.

The first and most obvious restriction is that coprocess instances should not be created dynamically. Since it is desirable to allow more than one instance of a coprocess in a program, coprocess instances should be created statically

```
program EX45;

    var PD1, PD2 : ref path descriptor
    data start-up(X, PD1, EX45);
    data start-up(X, PD2, EX45);

    coprocess X;
    begin
        ...
    end;

begin
    ...
end.
```

Example 4.5

using declaratory statements, as shown in example 4.5.

The second restriction we place on coprocesses is that they are not permitted to be statically nested within procedures. The need for this restriction becomes apparent when one considers the fact that coprocess instances are created statically and procedure instances are created dynamically in the overlay storage management policy. Creation of a coprocess instance includes initialisation of the processor of the coprocess. This, however, includes initialisation of its environment, which itself includes the environment of the procedure within which it is nested. It is therefore impossible to statically initialize the EP of the coprocess which is statically nested within a procedure.

The third restriction we place on coprocesses is that value parameters of a coprocess should only be compile time constants, and reference parameters should not be variables local to a procedure. As all instances of coprocesses are statically created, the need for these restrictions is obvious.

A feature which cannot be implemented using the bottom up storage management policy is the (optional) invocation of a procedure on transfers of control between control paths. These procedures must be allocated non-overlapping (disjoint) storage in a coprocess segment.

CHAPTER 5

INTERRUPT HANDLERS

5.0 INTRODUCTION

This chapter is concerned with the design of language structures for processing the interrupts in a machine. There is a great deal of similarity between an interrupt and a procedure call. Conceptually the occurrence of an interrupt may be viewed as a procedure call initiated by hardware. This conceptual view not only enhances our understanding of interrupts, but also provides us with guidelines for determining the language structures required for processing the interrupts in a system. The conceptual view of an interrupt does not, however, provide an adequate basis for making all the design decisions involved in the design of language structures for processing interrupts. Design of these structures must take into consideration the specific interrupt structure of the target machine, as well as the implementation requirements of these structures. In the following, these considerations are used as the basis for the design of high level language structures for processing and controlling the interrupts in a system.

The structures developed in this chapter for processing, and controlling the interrupts in a system, are similar in many respects to the structures in ESPL [Manacher-71]. Those presented here, however, are less powerful. This, we believe, makes them more consistent with the stringent requirements for performing real-time I/O, and well suited to the application area addressed by RT-PASCAL. The structures developed in ESPL are designed for more general application.

The chapter is organised as follows. Section 5.1 discusses interrupts from a conceptual point of view, and makes some preliminary observations concerning the language structures required for processing them. Section 5.2 discusses an interrupt architecture which is applicable to a large class of conventional mini and micro computer architectures, which are the primary target machines of RT-PASCAL programs. In addition, it discusses the implementation requirements of language structures of processing the interrupts in a machine. Section 5.3 discusses the design of language structures for processing interrupts for the class of machines discussed in section 5.2. Special attention is paid to the implementation requirements in the design of these structures.

5.1 CONCEPTUAL VIEW OF INTERRUPTS

Conceptually, an interrupt may be viewed as a procedure call to a programmed procedure called an interrupt handler. This conceptual view is independent of the precise interrupt structure of a machine, and also independent of the particular implementation requirements of the interrupt processing facility. The actions which occur on the acknowledgement of an interrupt closely resemble the actions that occur on execution of a procedure call statement. As in the case of procedures, an interrupt causes creation and initialisation of a contour and processor for the interrupt handler, followed by transfer of control to the newly created processor.

A characteristic which distinguishes an interrupt from a normal procedure call is that it occurs asynchronously, and that it is directly invoked by hardware. A consequence of this characteristic is that the interrupt handler is invoked directly in the path of the executing control path. In order to maintain the sequential order of the interrupted path, these interruptions should be transparent to the interrupted control path. It is therefore necessary to restore the state of the interrupted control path on exit from an interrupt handler.

Since interrupt handlers are similar to procedures, it is reasonable to assume that the semantics of return and resume instructions occurring in them are similar to those occurring within procedures. Assuming this, a return instruction will restore the state of the record of execution of the interrupted control path. The resumê instruction, however, causes problems. These problems can be eliminated by requiring that the hidden path descriptor variable be updated to refer to the interrupted processor before transfer of control.

The need for these updates can be eliminated by modifying the initialisation of the processor of an interrupt handler. These modifications are: initialise RPD and PD of the processor of the interrupt handler to nil, and transfer control without updating the hidden path descriptor variable in whose path the interrupt handler is invoked. It is interesting to note that these modifications also make the implementation of an interrupt handler much more efficient.

It is apparent that these modifications necessitate corresponding modifications in the return and resume instructions occurring within an interrupt handler. In a return instruction, this involves returning control to the processor referred to by the RP field of the processor executing the return instruction. In a resume instruction, control is transferred to the processor indirectly referred to by the path descriptor variable specified in the instruction. Minor extensions to these semantics are also necessary to deal with cases in which a resume statement specifies the execution of a procedure in the path of the resumed coprocess. The necessary modifications are similar to those discussed in chapter 4, and therefore are not discussed in this chapter.

It is desirable to allow interrupt handlers to call procedures. The procedure call should have semantics similar to interrupt handlers. In particular, exit of such a

```
program EX51;

    var R1 : reference path descriptor;
        COND : boolean;
    data start-up(X, R1, EX5X);

    interrupt handler IH;
    begin
        ...
        if COND then resume(EX5X);
        ...
    end;

    coprocess X;
    begin
        ...(A)...
        ...
        ...(B)...
    end;

begin
    ...
    resume(R1);
    ...
end.
```

Example 5.1

procedure via a resume statement should terminate the procedure and the interrupt handler. This requires a modification in the procedure call semantics discussed in chapter 4. These modifications concern the initialization of the procedure's processor. They are similar to the modifications discussed above in connection with interrupt handlers and are therefore not discussed any further.

As an illustration of the call and exit semantics of an interrupt handler, consider example 5.1. Let us assume for illustration purposes that there are two occurrences of the

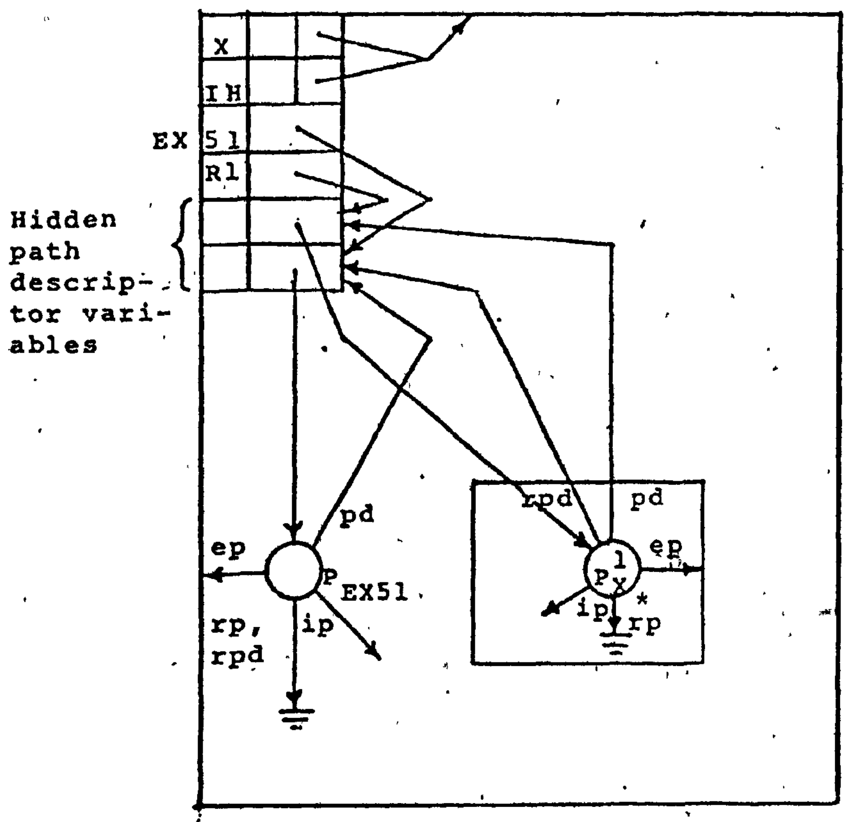


Figure 5.1 State of the record of execution before occurence of the interrupt which causes interrupt handler IH to be invoked. The value of the field ip of processor P^1_X is 9 when the interrupt occurs at point A and 10 when it occurs at point B in example 5.1.

interrupt which causes interrupt handler IH to be invoked. Furthermore let us assume that these interrupts occur when coprocess X is executing at points (A) and (B) in example 5.1, and that the interrupt handler is exited via a return

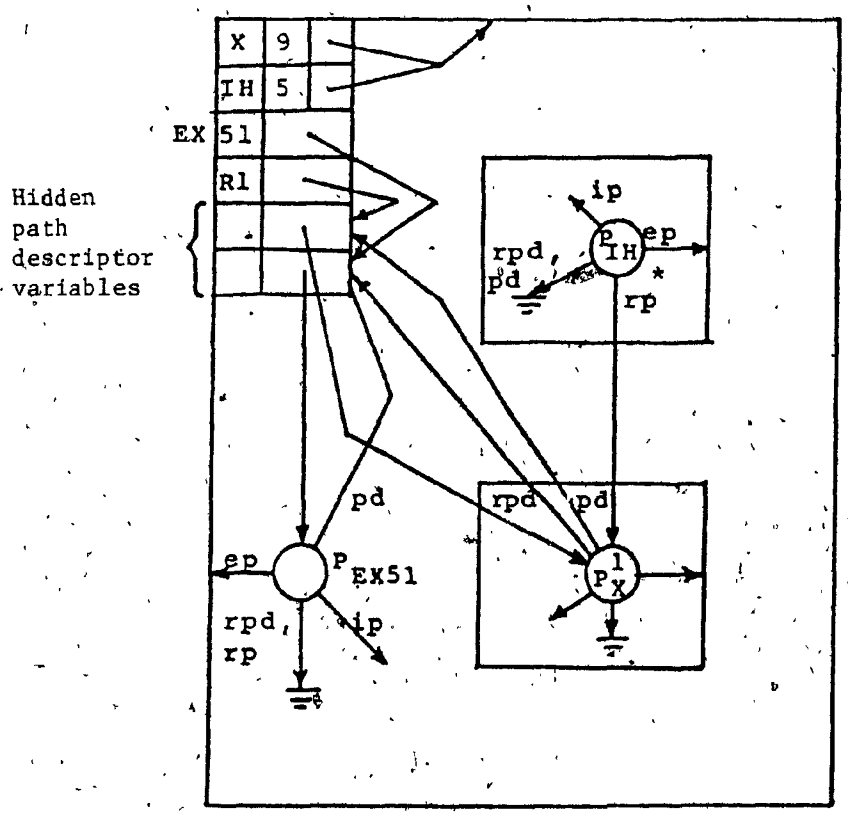


Figure 5.2 State of the record of execution after the interrupt handler IH is invoked.

statement in first instance, and via a resume statement in the second instance.

The states of the record of execution before interrupt handler IH is invoked in the two cases, are as shown in

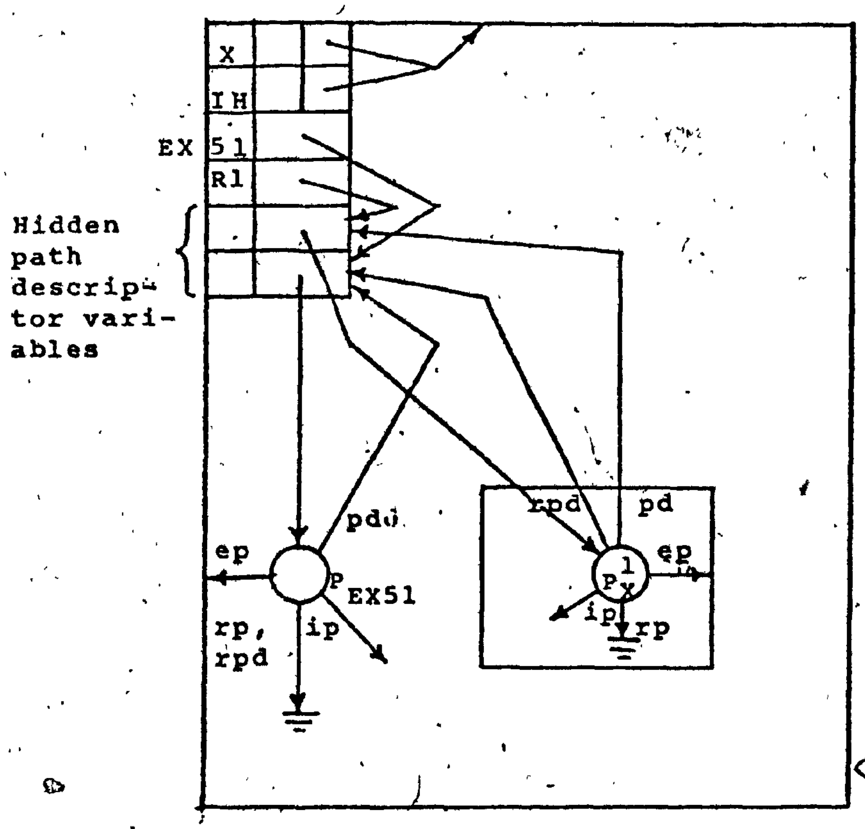


Figure 5.3 State of the record of execution after the interrupt handler IH is exited. Processor P_X^1 is active if IH exits via a return statement, Processor P_{EX51} is active if IH exits via a resume(EX51) statement.

figure 5.1. Here, we have assumed that the control path referred to by R1 has not called a procedure. The only difference between the records of execution in the two cases is that their IP's are different. The state of the record of execution after the interrupt handler IH is invoked is shown in figure 5.2. Note in this figure that the RPD and PD fields of the interrupt handler's processor are set to nil, and that the hidden path descriptor variable of coprocess X is not updated. Finally, the state of the record of execution after the interrupt handler is exited via the return instruction in the first instance and via the resume instruction in the second instance, is as shown in figure 5.3. In either case the state of the record of execution is the same. The only difference between them is that coprocess X is active in the first instance, whereas the main program is active in the second instance. Note also that figure 5.3 is identical to figure 5.1. The only difference is that coprocess X is active in figure 5.1, whereas in figure 5.3 this is not necessarily true. Here the main program is active when IH exits via the resume statement and coprocess is active when it exits via the return statement.

The above conceptual view of an interrupt allows us to make some preliminary but important observations regarding the language structures for processing the interrupts in a system. Firstly, it suggests that the language provide representations for interrupt handlers. Secondly, since interrupt handlers are invoked directly by hardware, it suggests the need for language restrictions for preventing them from being invoked by software. Finally it suggests the need for language structures for binding interrupts to interrupt handlers.

5.2 INTERRUPT MODEL AND IMPLEMENTATION CONSTRAINTS

Most real-time/systems implementation languages usually avoid introducing language structures for processing the interrupts in a machine. This is because of the great diversity of interrupt mechanisms in conventional computer architectures, which makes it difficult to devise structures which can be efficiently implemented on all machines, and also makes it difficult to devise a generally applicable set of language structures. This section therefore discusses an interrupt model which models the behaviour of a large class of mini and micro computers which are the principal target machines of RT-PASCAL programs. The implementation requirements for language structures for processing and controlling interrupts are more stringent than the implementation requirements of other language structures. This section also discusses the special implementation requirements of these structures.

Our model, which in many respects is an adaptation of Manacher's model [Manacher-71], can be summarised by the following points:

1) An interrupt consists of:

   1) A transfer of control caused by an external device connected to the computer, to a routine called the interrupt handler (or interrupt program, or interrupt routine).

   2) No loss of information occurs in the transfer,

thus making it possible to later return to the interrupt program.

2) A machine may have several different interrupts. Associated with each interrupt is a fixed location in memory, called the interrupt location. This location serves to identify the interrupt handler to which control is to be transferred on receipt of the interrupt. No assumption is made on the number of interrupts connected to a single interrupt: several devices may be connected to a single interrupt as in the M 6800, or an interrupt may be dedicated to a particular device as in the PDP 11.

3) The machine has the following registers and instructions:

   1) Associated with the machine is a one bit register called the interrupt enable register, for controlling all interrupts. Instructions exist for enabling and disabling all interrupts.

   2) Associated with each interrupt is a one bit register called the specific interrupt enable register (or mask register). Instructions exist for enabling and disabling these specific interrupts.

   3) Associated with every interrupt is a priority, which is set by the hardware of the machine. These priorities are basically tie breaking mechanisms: higher priority interrupts are

acknowledged before lower priority interrupts.

4) Associated with every interrupt is a one bit interrupt request register. An I/O device requests an interrupt by enabling this register. The register is reset when the interrupt is acknowledged.

4) A request for an interrupt is acknowledged if and only if the following four conditions are simultaneously satisfied.

1) The interrupt request register of the interrupt is set.

2) The specific interrupt enable register associated with the interrupt requested is enabled.

3) The interrupt enable register of the machine is enabled.

4) The interrupt request has the highest priority among all the other interrupts requested which simultaneously satisfy conditions 4.1, 4.2 and 4.3.

5) An interrupt request is acknowledged by the following sequence of events:

1) the interrupt register and the interrupt request register are disabled.

2) The state of the ongoing computation is saved.

3) Control is transferred to the interrupt handler, whose identity is obtained from the interrupt location associated with the interrupt.

The most important considerations in the design of language structures for processing the interrupts in a system are efficiency and availability. The nature of an interrupt makes it a time critical task, that is a task which requires a response within a critical period of time. This implies that the la[illegible]e structures designed for processing interrupts should be efficiently implementable. More specifically, the call and return semantics of an interrupt handler should be as efficient as possible. The time critical nature of an interrupt also makes availability an important consideration. Dynamic allocation of contours of interrupt handlers may cause unacceptable delays in the execution of an interrupt handler. It is therefore imperative that storage for the contours of interrupts be allocated statically.

5.3 DESIGN OF INTERRUPT HANDLING FACILITIES

The conceptual view, the interrupt model and the implementation requirements provide a basis for the design of structures for processing the interrupts in a system.

It is clear from the conceptual view that a language should provide means for defining interrupt handlers, and means for preventing these structures from being invoked by software. Means for defining interrupt handlers may be provided in a fashion similar to the way in which languages allow the definition of procedures. These structures may be prevented from being invoked by software by not providing them with a call statement. Other restrictions are necessary. Firstly, unlike procedures which may have parameters, interrupt handlers should not have parameters. Parameters are generally used to transfer information from a point of call to the calling procedure. Since our interrupt model does not transmit information, parameters are prohibited. Secondly, since interrupt handlers are allocated storage statically, it is necessary to restrict their declaration within procedures, which in RT-PASCAL are dynamic.

The conceptual view of interrupts also makes it apparent that a language must provide means for binding interrupts to interrupt handlers. This raises several issues. Firstly, decisions must be made concerning their binding time. Although general dynamic binding may not be necessary, it may be desirable to permit dynamic binding of interrupts to interrupt handlers on entry to a procedure. We will see later that such bindings raise efficiency problems, especially when considered in conjunction with dynamically nested interrupts. We therefore restrict bindings of interrupts to interrupt handlers to be static. Secondly, decisions must also be made concerning whether an interrupt handler may be bound to several interrupts. Since storage for interrupt handlers must be allocated statically, such bindings should be prohibited. Each interrupt handler should therefore be bound to a single interrupt. Finally, decisions must be made concerning the precise syntax of the bindings of interrupts to interrupt handlers. Consideration of the interrupt model suggests that such bindings may be performed by associating in the declaration of an interrupt handler the interrupt location which uniquely identifies an interrupt. As an illustration of such bindings, consider example 5.2, which defines an interrupt handler INT-HDLR for an interrupt whose interrupt location is <00FFH> (where H states that the location is in

```
interrupt handler INT-HDLR <00FFH>;
         ...
         (*local declarations of interrupt handler*)
         ...
begin
         ...
         (*body of interrupt handler*)
         ...
end;
```

Example 5.2

hex).

It is apparent from the interrupt model that it does not permit dynamic nesting of interrupt handlers, since it disables the interrupt enable register as soon as an interrupt is acknowledged. It is, however, desirable to permit dynamic nesting of interrupt handlers, to permit reasonable response times for extremely time critical interrupts. Hence the interrupt register should be enabled just before entry to the interrupt handler (that is after housekeeping tasks to maintain control have been performed).

The possibility of having dynamically nested interrupt handlers raises efficiency problems, if dynamic bindings of interrupts to interrupt handlers are permitted on entry to a scope. As an illustration of the efficiency problems,

```
program EX52

   interrupt handler CRIH1 <00F1H>;
   begin
       ...
   end;

   interrupt handler PRIH <00F3>;
   begin
       ...
   end

   procedure P;
       interrupt handler CRIH2 <00F1H>;
       begin
           ...
       end;
   begin
       ...
       ...(B)
       ...
   end

begin
   ...
   call P
   ...(A)
end
```

Example 5.3

consider the skeleton program shown in example 5.3.

In example 5.3 there are two interrupt handlers CRIH1 and CRIH2 for the interrupt <00F1H>. Interrupt handler CRIH1 must be bound to interrupt <00F1H> in the scope of the main program, whereas interrupt handler CRIH2 must be bound to the interrupt <00F1H> when control enters procedure P.

The problem of efficiency arises because interrupt <00F3H> can occur at either point (A) or point (B) in the program. There is no problem if it occurs at point (A), since the interrupt handler bound to interrupt <00F1H> is CRIH1. If however it occurs at point (B), then on entry to the interrupt handler PRIH, it will have to bind interrupt <00F1H> to interrupt handler CRIH1, since by block structure scope rules CRIH1 is active in PRIH. Furthermore on exit from the interrupt handler it will have to restore the original binding i.e. it will have to restore the binding of <00F1H> to CRIH2.

The only general way to handle this problem is to save the original bindings of all interrupt handlers, replace them with new bindings on entry to a interrupt handler, and restore the original bindings on exit from the interrupt handler. This undoubtedly is inefficient. We therefore prohibit all forms of dynamic bindings of interrupts to interrupt handlers. This rule therefore specifies that interrupts may only be bound to an interrupt handler in the outermost scope of a program.

Dynamically nested interrupt handlers also raise problems concerning recursive or circular entry of interrupt handlers. Recursive or circular entry of an interrupt handler is undesirable for the following reasons:

1) Recursive or circular entry of an interrupt handler implies the need for dynamic storage management of

activation records of interrupt handlers. We have seen above however, that storage for interrupt handlers must be allocated statically.

2) Global data accessed by recursively or circularly entered interrupt handlers is subject to corruption since it cannot be protected from simultaneous access by several activations of an interrupt handler.

Hence interrupt handlers must be protected from recursive or circular entry.

Recursive or circular entry of an interrupt handler can be easily prevented by disabling the specific interrupt register on entry to the interrupt handler, and only enabling it on exit (via a return statement) from the interrupt handler. Since in general it may be desirable to prevent a whole class of interrupts when a particular interrupt handler is active, we adopt a generalisation of the above scheme.

Our method for preventing recursive or circular entry of an interrupt handler is essentially based on the method used to prevent recursive or circular entry of an interrupt handler in ESPL. As in ESPL, we introduce the notion of an interrupt level which specifies which interrupts can interrupt the execution of an interrupt handler servicing a particular interrupt. The idea is in essence a software concept, whose underlying hardware is the mask register discussed above (or the processor priority in the PDP-11), and is quite independent of the hardware concept of interrupt priority

[Manacher-71].

In this scheme, each declaration of an interrupt handler is accompanied by a declaration of an interrupt level I, where 0 <= I <= (N-1) , and N <= the number of interrupts in the system. Recursive or circular entry of interrupt handlers is prevented by ensuring that an interrupt handler at level I can be interrupted by an interrupt handler at level J, if and only if J < I.

It is essential to provide the programmer with means for controlling the interrupts of a machine. For example, an interrupt is needed when an ongoing computation and an interrupt handler share an interrupt buffer. Again, we adopt a solution originally proposed in ESPL. We propose two instructions: ENABLE(LEV-LIST), and DISABLE(LEV-LIST), where LEV-LIST in both cases is a list of level numbers. These instructions enable or disable all the interrupts associated with a particular level number specified in the instruction. In the special case where no level numbers are specified in LEV-LIST, these instructions enable or disable all interrupts.

We conclude this chapter with the following summary. Interrupt handlers are declared as follows:

    interrupt handler IH <n <l ;

where

IH is the name of the interrupt handler

n is the interrupt location (or interrupt) to which the interrupt handler is bound.

1 is the level. An interrupt handler at level I can be interrupted by an interrupt handler at level J if and only if I<J.

The following are the restrictions on interrupt handlers:

1) They cannot be invoked by software.
2) They cannot have parameters.
3) They are bound statically to interrupts and they cannot be bound to more than one interrupt.
4) They cannot be declared within procedures.

Programmer control over interrupts is provided by the following instructions:

1) enable(lev-list) which enables all the interrupts associated with the levels specified in lev-list.
2) disable(lev-list) which disables all the interrupts associated with the levels specified in lev-list.

Interrupt handlers and procedures called by interrupts have semantics similar to procedures discussed in chapter 4.

The differences in the procedure call semantics are as follows:

1) The fields RPD and PD of the processor created by the call instruction are set to nil.
2) Control is transferred to the newly created processor without updating the hidden path descriptor variable of the interrupted control path.

The semantics of the return statements are as follows:

1) Transfer control to the processor referred to by the

RP field of the processor executing the return instruction.

The structures for processing interrupts discussed in this chapter provide all the facilities required for performing real-time I/O and dealing with timer interrupts. Static allocation of interrupt handler activations and restrictions on their bindings to interrupts allow efficient implementations without violating the constraints on availability. Finally, dynamic nesting of interrupt handlers permits efficient response to time critical interrupts.

CHAPTER 6

CONCLUSION

This thesis has presented the design of structures for the implementation of dedicated real-time/system application programs on small systems in a bare machine environment. The structures are specifically designed to provide a sequential language with facilities for implementing multiprograms, and for performing I/O. These structures are the "coprocess", and the "interrupt handler" respectively. The structures developed have properties much sought in real-time/systems implementation languages. In particular, they neither preempt system design decisions, nor require run time support. Incorporation of these structures in a sequential language makes it functionally complete (with regard to implementation of real-time/system programs). This allows implementation of a complete system without the use of assembly language.

Compared to languages such as CONCURRENT PASCAL, implementation of a system in a sequential language incorporating these structures, places greater responsibility on the programmer. This, however, is more than compensated by the absence of preempted system design decisions, which in real-time/system applications have a great impact on system performance. In addition, these structures do not require run time support, thereby greatly enhancing the portability of programs. Compared to languages such as EUCLID, which also do not preempt system design decisions, these high level language structures provide facilities for implementing real-time/system programs without recourse to the use of assembly language. This adds to the understandability and portability of the system.

The design of the structures was based on language issues in the implementation of multiprograms and facilities for performing real-time I/O. Implementation of a multiprogram required that the language have structures for defining and controlling paths of control, in addition to facilities for dealing with "timer interrupts". Furthermore, safe and structured implementation of a multiprogram using these structures required that the language optionally allow inter path transfers of control to be accompanied by invocation of a programmer specified procedure. Implementation of facilities for performing I/O required that the language possess structures for programming peripheral devices. Specifically required in a sequential language were structures for dealing with interrupts in a system. This included facilities for defining interrupt handlers which could be invoked on occurrence of corresponding interrupts, and facilities fr binding interrupts to interrupt handlers. Other facilities were noted. The need to obtain reasonable response to time critical interrupts required that the language allow dynamic nesting of interrupt handlers. The need for low level synchronisation required that the language possess structures for controlling the interrupts in a system. Finally, the dual purpose served by interrupts in a system required that interrupt handlers posess capabilities which allowed them to return either to the interrupted control path, or to a path other than the one it interrupted. These facilities are incorporated in the structures developed in this thesis.

The significant contribution of the new structures, the coprocess and the interrupt handler, is the ease of implementing multiprogramming systems on a bare machine. They are particularly suited for implementing the functions of a real-time kernel, and they provide the primitive functions for performing real-time I/O using high level language structures.

REFERENCES

[Aho-77] A.V. Aho, J.D. Ullman. Principles of Compiler Design. Addison-Wesley Pub. Co., 1977.

[Baker-72a] F.T. Baker. Chief Programmer Team: Management of Production Programming. IBM Systems Journal, Vol. 11, No. 1, jan 1972.

[Baker-72b] F.T. Baker. System Quality through Structured Programming. "Proceedings AFIPS, Vol. 41, part 1, pp.339-343, 1972.

[Baker-73] F.T. Baker, H.D. Mills. Chief Programmer Teams. Datamation, Vol. 19, No. 12, Dec. 1973.

[Bekkers-77] Y. Bekkers, J. Briat, J.P. Verjus. Construction of a Synchronisation Scheme by Independent Definition of Parallelism. Proceedings of the IFIP Working Congress on Constructing Quality Software, May 1977, P.G. Hibbard, S.A. Schuman, (Eds.), North Holland Pub. Co., 1978.

[Berry-71] D.M. Berry. Introduction to Oregano. Proceeding Symposium on Data Structures in Programming Languages. Sigplan Notices, Vol. 6, No. 2, Feb. 1971.

[Birthwhistle-70] G. Birthwhistle SIMULA Begin. Pub.No. S-31, Norsk Regnecentral, Oslo, 1970.

[Bobrow-73] D.G. Bobrow, B. Wegbreit. A Model For Stack Implementation of Multiple Environments. CACM, Vol. 16, No. 10, Oct. 1973.

[Bochmann-78] G.V. Bochmann. Compile Time Memory Allocation for Parallel Processes. IEEE Transaction on Software Engineering, Vol. SE-4, No. 6, Nov 1978.

[Boehm-78] B.N. Boehm, J.R. Brown, H. Kasper, M. Lipow, G.J. McCleod, M.J. Merit. Characteristics of Quality Software. North Holland Pub.Co, 1978.

[Bohm-66] C. Bohm, G. Jacopini. Flow-Diagrams, Turing Machines, and Languages with only two formation rules. CACM, Vol. 9, No. 5, May 1966.

[Brinch Hansen-72a] P. Brinch Hansen. A Comparison of Two Synchronisation Concepts. Acta Imformatica, Vol. 1, No. 3, 1972.

[Brinch Hansen-73a] P. Brinch Hansen. Operating System Principles. Prentice Hall, 1973.

[Brinch Hansen-75a] P. Brinch Hansen. The Programming Language CONCURRENT PASCAL. IEEE Transactions on Software Engineering, Vol. 1, No. 2, June 1975.

[Brinch Hansen-77a] P. Brinch Hansen. Concurrent Pascal Report. In, The Architecture of Concurrent Programs, chapter 8, Prentice Hall, 1977.

[Brinch Hansen-77b] P. Brinch Hansen. Concurrent Pascal Machine. In, the Architecture of Concurrent Programs, chapter 9, Prentice Hall, 1977.

[Campbell-74] R.H. Campbell, A.N. Habermann. The Specification of Process Synchronisation by Path Expressions. Proceedings of an International Symposium held at Rocquencourt, Apr. 11, 1974.

[Cheatham-72] T.E. Cheatham Jr., B. Wigbreit. A Laboratory for the Study of Automatic Programming. AFIPS SJCC-72, 1972.

[COD-76] Conference on Data: Abstraction, Definition, and Structure. March 22-24, Salt Lake City, Utah. Sigplan Notices, Vol. 11, No. 4, Apr. 1976.

[Conway-63] M.E. Conway. Design of a Separable Transition Diagram Compiler. CACM, Vol. 6, No. 7, pp.396-408, 1963.

[Dahl-72a] O.J. Dahl, E.W. Dijkstra, C.A.R. Hoare. Structured Programming. Academic Press, USA, pp.220, 1972.

[Dahl-72b] O.J. Dahl, C.A.R. Hoare. Hierarchical Program Structures. See [Dahl-72a].

[Dahl-66] O.J. Dahl, K. Nygaard. SIMULA: An ALGOL-based Simulation Language. CACM, Vol. 9, No. 9, Sept. 1966.

[Darlington-73] J. Darlington, R.M. Burstall. A system which automatically improves programs. IJCAI73, Stanford University, Stanford, Calif., 1973.

[Demers-76] A.J. Demers, Encapsulated Data Types and Generic Procedures. See [DIPL-76].

[Denning-76] P. Denning. A Hard Look at Structured Programming. In, Structured Programming, D. Bates (Ed.), Infotech State of Art Report, 1976.

[Dennis-75] J.B. Dennis. Modularity. In, Software Engineering: An Advanced Course, F. L. Baurer (Ed.), Springer Verlag, New York, 1975.

[Dijkstra-68a] E.W. Dijkstra. Cooperating Sequential Processes. In, Programming Languages, F. Genuys (Ed.), Academic Press, 1968.

[Dijkstra-68b] E.W. Dijkstra. The Structure of "THE" Multiprogramming System. CACM, Vol. 11, No. 5, May 1968.

[Dijkstra-68c] E.W. Dijkstra. Goto Statement Considered Harmful. CACM, Vol. 11, pp.147-148, Mar. 1968.

[Dijkstra-68d] E.W. Dijkstra. Complexity Controlled by Hierarchical Ordering of Function and Variability. Conference sponsored by NATO Science Committee, Held at Garmisch, Ger., Oct. 7-11, 1968.

[Dijkstra-72a] E.W. Dijkstra. Hierarchial Ordering of Sequential Processes. In, Operating System Techniques. C. A. R. Hoare, R. H. Perrot (Eds.), Academic Press, 1972.

[Dijkstra-72b] E.W. Dijkstra. Notes on Structured Programming. See [Dahl-72a].

[DIPL-76] Design and Implementation of Programming Languages. DOD sponsored workshop, Ithaca 1976.

[Dreisbach-76] T.A. Dreisbach, L. Weissman. Requirements for Real Time Languages. See [DIPL-76].

[Elson-73] M. Elson. Concepts of Programming Languages. Science Research Association Inc., 1973.

[Fisher-72] D.A. Fisher. A Survey of Control Structures in Programming Languages. Sigplan Notices, Vol. 7, No. 11, Nov. 1972.

[Freeman-73] P. Freeman. Software System Principles. Science Research Association Inc., 1973.

[Geschke-75] G.M. Geschke, J.G. Mitchell. On the Problem of Uniform References for Data Structures. Proceedings International Conference on Reliable Software, Apr. 1975.

[Geschke-77] G.M. Geschke, J.H. Morris, E.H. Satterthwaite. Early Experience with MESA. CACM, Vol 20, No. 8, Aug.1977.

[Goos-77] G. Goos, V. Kastens. Programming Languages and the Design of Modular Programs. Proceedings of the IFIP working conference on construction of Reliable Software, May 1977. P.G. Hibbard, S.A. Schuman (Ed.), North Holland Pub. Co., (1978).

[Goos-75a] G. Goos. Language Characteristics: Programming Language as a Tool in Writing System Software. Software Engineering: an Advanced Course. F.L. Bauer (Ed.), Springer Verlag, New York, 1975.

[Goos-75b] G. Goos. Hierarchies. Software Engineering: an Advanced Course. F.L. Bauer (Ed.), Springer Verlag, New York, 1975.

[Gries-74] D. Gries. On Structured Programming. CACM, Vol. 17, No. 11, Nov. 1974.

[Gries-77] D. Gries, N. Gehani. Some Ideas on Data Types in Higher Level Languages. CACM, Vol. 20, No. 6, june 1977.

[Griffiths-76a] M. Griffiths. Run Time Storage Management. Compiler Construction: An Advanced Course. F.L. Bauer, J. Eickel, (eds.), Springer-Verlag 1976.

[Habermann-72] A.N. Habermann. Synchronisation of Communicating Processes. CACM, Vol. 15, No. 3, Mar. 1972.

[Hardgrave-76] W.T. Hardgrave. Positional Versus Keyword parameter communication in programming Languages. Sigplan Notices, Vol. 11, No. 5, May 1976.

[Hegering-76] H.G. Hegering, D. Schneider, A. Schwald, G. Seegmueller. Systems Programming Elements of the Language ASTRA. In, Software Systems Engineering, B. Randell, (Ed.,) Online 1976.

[Hoare-69] C.A.R. Hoare. An Axiomatic Basis for Computer Programming. CACM, Vol. 12, No. 10, Oct. 1969.

[Hoare-72] C.A.R. Hoare. Notes on Data Structuring. See [Dahl-72a].

[Hoare-74a] C.A.R. Hoare. Monitors: An Operating System Structuring Concept. CACM, Vol. 17, No. 10, Oct. 1974.

[Hoare-78a] C.A.R. Hoare. Data Structures. In Current Trends in Programming Methodology, Vol. 4, R.T. Yeh (Ed.), Prentice Hall, 1978.

[Hoare-81] C.A.R. Hoare. Turing Award Lecture.. CACM, Vol. 24, No. 2, Feb. 1981.

[Horning-76] J.J. Horning. Some Desirable Properties of Data Abstraction Facilities. See COD-76.

[Ichbiah-79a] Preliminary ADA Reference Manual. Sigplan Notices, Vol. 14, No. 6, June 1979.

[Ichbiah-79b] J.D. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, O. Roubine, B.A. Wichmann.. Rationale for the Design of the ADA Programming Language. Sigplan Notices, Vol. 14, No. 6, June 1979.

[Jackson-76a] M.A. Jackson. Data Structures as a Basis for Program Design. In Structured Programming, Infotech State of the Art Report, 1976.

[Jammel-77] A. Jammel, H.G. Stiegler. Managers Versus Monitors. Proceedings IFIP congress 1977.

[Johnston-71] J.B. Johnston. The Contour Model of Block Sturctured Processes. Proceedings Symposium on Data Structures in Programming Languages. Sigplan Notices, Vol. 6, No. 2, Feb. 1971.

[Johnson-76] R.T. Johnson, J.B. Morris. Abstract Data Types in the MODEL Programming Language. See [COD-76].

[Kessels-77] J.L.W. Kessels. An Alternative to Event Queues for Synchronisation in Monitors. CACM, Vol. 20, No. 7, July 1977.

[Knuth-77] D.E. Knuth. Structured Programming with GOTO Statements. Current Trends in Programming Methodology, R.T. Yeh (Ed.), 1977.

[Koster-76] C.H.A. Koster. Visibility and Types. See[COD-76].

[Lampson-77] B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell, G.J. Popek. Report on the Programming Language EUCLID. Sigplan Notices, Vol. 12, No. 2, Feb. 1977.

[Lang-68] C.A. Lang. Languages for Writing Systems Programs. Conference Sponsored by NATO Science Committee, held at Garmisch, Ger., Oct. 7-11 1968. P. Naur, B. Randell, J.N. Buxton (Eds.).

[Laventhal-78] M.S. Laventhal. Synthesis of Synchronisation Code for Data Abstractions. Doctoral thesis, Massachusetts Institute of Technology, 1978. (MIT/LCS/TR-203).

[Leavenworth-72] B.M. Leavenworth. Programming with(out) the GOTO. Proceedings of the ACM Annual Conference, Aug. 1972.

[Linden-76] T.A. Linden. The Use of Abstract Data Types to Simplify Program Modification. See [COD-76].

[Liskov-72a] B.H. Liskov. The Design of the Venus Operating System. CACM, Vol. 15, pp.144-149, Mar. 1972.

[Liskov-72b] B.H. Liskov. A Design Methodology for Reliable Software System. Proceedings AFFIPS, Vol. 41, part 1, pp.191-199, 1972.

[Liskov-74] B.H. Liskov, S. Zilles. Programming with Abstract Data Types. Sigplan Notices, Vol. 9, No. 4, Apr. 1974.

[Liskov-75a] B. Liskov. An Introduction to CLU. New Directions in Algorithmic Languages, S.A. Schuman (Ed.), 1975.

[Liskov-77] B. Liskov, A. Snyder, R. Atkinson, C. Schaffert. Abstraction Mechanisms in CLU. CACM, Vol. 20, No. 8, Aug. 1977.

[Lister-77] A. Lister. The Problem of Nested Monitor Calls. Operating Systems Review, Vol. 11, No. 3, july 1977.

[Mckeag-76] R.M. McKeag. THE Multiprogramming System. See [SOS-76].

[Meertens-77] L. Meertens. Program Text and Program Structure. Proceedings of the IFIP Working Conference on Construction quality software, May 1977. P.G. Hibbard, S. A. Schuman (Eds.). North Holland Pub. Co., (1978).

[Mills-73] H.D. Mills. On the Development of Large Reliable Programs. Proceedings, 1973 IEEE Symposium on Software Reliability. New York, Apr-May, 1973.

[Mitchell-78] J.G. Mitchell, B. Wegbreit. Schemes: a High Level Data Structuring Concept. In Current Trends in Progrmming Methodology, R.T. Yeh (Ed.), Vol. 4, 1978.

[Morris-79] J.B. Morris. Data Abstraction: a Static Implementation Strategy. Proceedings on ACM Conference on Programming Languages, 1979.

[Moss-77] J.E.B. Moss. Abstract Data Types in Stack Based Languages. MIT/LCS/TR-190, Massachusetts Institute of Technology, Masachusetts, 1977.

[Nestor-76] J.R. Nestor. Models of Data Objects and Data Types. See [DIPL-76].

[Organick-71] E.E. Organick, J.G. CLEARY. A Data Structure Model of the B-6700 Computer System. Proceedings Symposium of Data Structures in Programming Languages. J.T. Tou, P. Wegner (Eds.), Sigplan Notices, Vol. 6, No. 2, Feb. 1971.

[Parnas-75] D.L. Parnas. A Technique for Software Module Specification with Examples. CACM, Vol. 15, No. 5, May 1975.

[Parnas-76] D.L. Parnas, J.E. Shore, D. Weiss. Abstract Types Defined as Classes of Variables. See [CQD-76]].

[Popek-77] G.A. Popek, J.J. Horning, B.W. Lampson, J.G. Mitchell, R.L. London. Notes on the Design of EUCLID. Proceedings ACM Conference on Language Design for Reliable Software, march 1977.

[Prenner-71] C.J. Prenner. The Control Structure Facility of ECL. Sigplan Notices, Vol 6, No. 12, Dec. 1971.

[Prenner-73] C.J. Prenner. Extensible Control Structures. Sigplan Notices, Vol. 8, No. 9, Sept. 1973.

[Ravn-80] A.P. Ravn. Device Monitors. IEEE Transaction on Software Engineering, Vol. SE-6, No. 1, pp 49-53, jan. 1980.

[Richard-76] F. Richard, H.F. Ledgard. A Reminder to Language Designers. See [DIPL-76].

[Ross-70] D.T. Ross. Uniform Referents: An Essential Property for a Software Engineering Language. In Software Engineering, Vol. 1, Tou J.T. (Ed.), Academic Press, 1970.

[Sahasrabuddhe-76] H.V. Sahasrabuddhe. Types of Abstraction in Programming. Software Systems Engineering, B. Randell (Ed.), Online 1976.

[Schuman-75] S.A. Schuman. On Generic Functions. New Directions in Algorithmic Languages 1975, S.A. schuman (Ed.).

[Seegmueller-76] G. Seegmueller Language Aspects in Operating Systems. Lecture Notes in Computer Science, Vol. 46, Springer Verlag, 1976.

[Shankar-80] K.S. Shankar. Tutorial: Data Structures, Types and Abstraction. Computer, Vol. 13, No. 4, april 1980.

[Shaw-80] M. Shaw, W.A. Wulf. Towards Relaxing the Assumptions in Languages and their Implementations. Sigplan Notices, Vol. 15, No. 3, March 1980.

[SOS-76] R.M. KcKeag, R. Wilson. Studies in Operating Systems. Academic Press, 1976.

[SP-76] Structured Programming. Infotech State of the Art Report, D. Bates (Ed.), 1976.

[Turner-80] J. Turner. The Structure of Modular Programs. CACM, Vol. 23, No. 5, May 1980.

[Waite-76a] W.M. Waite. Relationship of Languages to Machines. In Compiler Construction: an Advanced Course, F.L. Bauer, J. Eickel, eds., Springer Verlag, 1976.

[Waite-76b] W.M. Waite. Semantic Analysis. In Compiler Construction: an Advanced Course. F.L. Bauer, J. Eickel, eds., Springer Verlag, 1976.

[Wegbreit-74] B. Wegbreit. The Treatment of Data Types in ELI. CACM, Vol. 17, No. 5, May 1974.

[Wegner-68] P. Wegner. Programming Languages, information Structures and m/c Organization, McGraw Hill pub. Co., 1968.

[Wegner-71] P. Wegner. Data Structure Models for Programming Languages. Proceeding Symposium on Data Structures in Programming Languages, Sigplan Notices, Vol. 6, No. 2, Feb. 1971.

[Wirth-71a] N. Wirth. Program Development By Step-Wise Refinement. CACM, Vol. 14, No. 4, Apr. 1971.

[Wirth-71b] N. Wirth The Programming Language PASCAL. Acta Informatica, Vol. 1, No. 1, 1971.

[Wirth-72] N. Wirth. The Programming Language PASCAL and its Design Criteria. In, High Level Languages, Infotech state of the Art Report, Vol. 7, 1972.

[Wirth-73] N. Wirth. Systematic Programming: An Introduction. Prentice Hall, 1973.

[Wirth-74b] N. Wirth. On the Design of Programming Languages. Proceedings of the IFIP Congress 1974.

[Wirth-74a] N. Wirth. On the Composition of Well Structured Programs. ACM Computing Surveys, Vol. 6, No. 4, Dec. 1974.

[Wirth-77c] N. Wirth. Towards a Discipline of Real-Time Programming. CACM, Vol. 20, No. 8, August 1977.

[Wirth-77b] N. Wirth. Design and Implementation of Modula. Software Practice and Experience, vol. 7, No. 1, 1977.

[Wirth-77a] N. Wirth. MODULA: A Language for Modular Multiprogramming. Software Practice and Experience, Vol. 7, No. 1, 1977.

[Woodger-71] M. Woodger. On Semantic Levels in Programming. Proceedings of the IFIP Congress, 1971.

[Wulf-71a] W.A. Wulf. Programming without the GOTO. Proceedings of the IFIP congress, 1971.

[Wulf-71b] W.A. Wulf, D.B. Russel, A.N. Habermann. BLISS: A Language for Systems Programming. CACM, Vol. 14, No. 12, Dec. 1971.

[Wulf-73] W.A. Wulf, M. Shaw. Global Variables Considered Harmful. Sigplan Notices, Vol. 8, No. 2, Feb. 1973.

[Wulf-76] W.A. Wulf. Structured Programming in the Basic Layers of an Operating System. In, Language Hierarchies and Interfaces, G. Goos, J. Hartmanis (Ed.), Springer Verlag, 1976.

[Wulf-77] W.A. Wulf. Languages and Structured Programs. In, Current Trends in Programming Methodology, Vol. 1, R.T. Yeh (Ed.), Prentice Hall Pub. Co., 1977.