The Monitor and Synchroniser concepts in the programming language CLANG

Thesis
submitted by

ALAN GORDON CHALMERS

in fulfilment
of the requirements
for the degree

MASTER OF SCIENCE

Rhodes University
June, 1984

ACKNOWLEDGEMENTS

I would like to express my sincere thanks and gratitude to my supervisor, Pat Terry, for the encouragement given, and the enormous effort undertaken with the reading of the rough drafts of this thesis.

Thanks also to my friends and colleagues, David, for all the advice and inspiration provided, and Vaughan, for the constructive critiscism which arose from many a discussion.

I would also like to acknowledge the financial support of Rhodes University, and the Council for Scientific and Industrial Research.

My final thanks must go to my parents, without whom none of this would have been possible.

Table of contents

Chapter 1: Introduction

Sequence, n : Succession, coming after or next, set of things that belong next to each other on some principle of order, series without gaps.

Concurrent, a&n : Running together, as parallel lines; co-operating

The concise Oxford dictionary

An essential factor in the continuing use of computers is the development of software. This software for a particular application typically consists of one or more programs. There are two main types of program.

A sequential program consists of a list of statements that is executed sequentially; its execution is called a process.

A concurrent program specifies a set of interacting (or even totally independent) sequential programs that may be executed concurrently as parallel processes.

In the last 15 years the state of the art of concurrent programming has advanced significantly.

Advances in hardware have increased the availability of inexpensive processors, and thus made possible the construction of distributed systems and multiprocessors which were previously considered economically infeasible.

Along with these advances in hardware have come theoretical developments by such people as Dijkstra (1968), Brinch Hansen (1972, 1973) and Hoare (1974, 1978) which have led to new programming notations for the easy and explicit expression of concurrent process initialisation, communication and synchronisation.

Of particular interest to this thesis are those constructs developed for interprocess communication and synchronisation. These have included the low level construct of the Semaphore (originally developed by Dijkstra in 1968 [Dij68]) and two higher level constructs: the Monitor (developed independently by Brinch Hansen in 1973 [Bri73] and Hoare in 1974 [Hoa74]), and the Rendezvous (developed by Hoare in 1978 [Hoa78], implemented in the programming language Ada, and adapted in CLANG in 1983 by the author as the Synchroniser).

As the result of these developments in both hardware and software, the art of concurrent programming is no longer restricted to the designers and implementors of operating systems; it has now become possible to contemplate using concurrent programming for all kinds of applications: for example, database management systems, large scale parallel scientific computations and real-time, embedded control systems, to mention but a few.

However, to programmers schooled in sequential programs the change to the concurrent way of thought is fairly traumatic, especially when this entails the learning of a whole new language. The programmer then has to learn the involved syntax of this new language, while at the same time trying to grasp the concepts of concurrency.

Although the theoretical study of concurrent programming is well advanced, actual languages that implement concurrent features are not readily available; or if they are (eg. Modula-2, but cf. Chapter 4), their cost for procurement is generally quite high.

It is for this reason that experimental simple languages have been developed for the sole purpose of teaching students, at both the undergraduate and postgraduate level, the concepts of concurrency, without having the students floundering over the syntax of a complex new language.

One of the first of these such languages was an extended version of Wirth's Pascal-S, [Wir75], proposed by M. Ben-Ari in his book, [Ben82], published in 1982. (Pascal-S is a subset of the language Pascal.) Ben-Ari modified the subset and implemented concurrency based on the idea of processes launched by an explicit **Cobegin..Coend** construct. Although Pascal-S is designed to run on a single processor, the parallel execution of the processes in Ben-Ari's extension is simulated by context switching between processes after a small random number of steps. Semaphores are the only exclusion and protection mechanism provided in this extension.

Experience with Pascal-S here at Rhodes University leads us to believe that it would require a fair degree of modification to convert Ben-Ari's system for use on a micro-computer.

Ben-Ari's ideas together with the ideas from an independent, though similar, extension to Pascal-S by the author in 1982, [Cha82], led to the initial development of the language CLANG by Terry, [Ter83], as a possible vehicle for teaching concurrency to students.

Further extensions to CLANG to provide the **monitor** and **synchroniser** constructs form the practical basis for the present study.

The aims of this project have been to implement a computer language suitable for teaching both undergraduate and postgraduate students about concurrent programming with special reference to the high level constructs available for concurrent process synchronisation and communication. These constructs have to be readily distinguishable to the programmer and easy to use.

Another design goal for this extended version of CLANG is that it had to be able to run reasonably efficiently on small micro-computers (such as the Apple II) so as to ensure its widespread availability.

The experience gained in the design and implementation of the monitor and synchroniser has been used to assess the potential of similar constructs in other programming languages and to compare and contrast the advantages and disadvantages of these mechanisms with those in CLANG, both from the programmer's point of view and the ease of implementation.

The remainder of this thesis is arranged as follows:

Chapter 2: An introduction is given to the programming language CLANG, as most of the examples throughout the thesis are given in CLANG notation.

Chapter 3: A discussion is given of the problems involved with concurrency, namely the necessity for concurrent processes to synchronise and communicate in order to co-operate. This chapter also looks at one of the earliest solutions to these problems, the semaphore, and shows how the difficulties with the semaphore evolved into a need for higher level constructs such as the monitor and the synchroniser.

Chapter 4: The problems of the monitor concept are outlined, followed by an assessment of the monitor concept in the languages: Concurrent Pascal, Edison, Modula-2 and Pascal Plus. This is followed by a similar assessment of the monitor concept as implemented in CLANG, so that the differences can be contrasted. Also included is a description of how the monitor concept was actually implemented. The description is done by means of flow diagrams and worked examples; the listing of the code can be found in appendix B.

Chapter 5: The synchroniser concept is tackled in a similar manner as was the monitor concept in chapter 4. The languages assessed and contrasted with CLANG are Ada and CHILL.

Chapter 6: Conclusions are drawn from chapters 4 and 5 as to the potential of CLANG as a language for teaching concurrent programming. The merits of both the monitor and the synchroniser are debated in an endeavour to ascertain whether there is a need for either (or both) in a concurrent programming language.

Also under discussion in this chapter are other forms of interprocess synchronisation not implemented in CLANG.

Finally the question is raised:

> "Are there other (perhaps better) methods of expressing interprocess synchronisation and communication not yet discovered ?"

Chapter 2: Introduction to CLANG

"The name CLANG (standing for Concurrent **LANG**uage) was chosen as it had a pleasant ring to it"

P.D Terry [private communication]

CLANG is an experimental (very stripped down) Pascal-like language developed at Rhodes University by Terry, originally for teaching compiler design and implementation to undergraduate students.

It is based on ideas found initially in Wirth's "Algorithms + Data Structures = Programs" [Wir76], with ideas for simulating concurrency adapted from Ben-Ari's "Principles of Concurrent Programming" [Ben82] and "The Pascal-S Mark1.HAC compilers" by the author [Cha82]. Quite a lot of inspiration was obtained from Wirth's Pascal-S [Wir75].

The language supports the usual WHILE and REPEAT loops (including a REPEAT ... FOREVER infinite loop), FOR loops and the IF ... THEN ... ELSE construct, PROCEDURES and FUNCTIONS (which may be nested, and declared FORWARD). Concurrency is initiated using an explicit COBEGIN..COEND construct.

The main restriction is in the field of data typing. Essentially only one type is supported - INTEGER. Only simple integers and simple one-dimensional arrays may be declared and at present arrays may not be passed as parameters to subprograms. I/O is very simple, limited to the input of integers or characters and the output of constant strings, characters or integer expressions.

The advanced features of the monitor and synchroniser implemented in CLANG have enabled CLANG to be used as a language for teaching concurrent programming to postgraduate as well as undergraduate students.

The language is compiled into intermediate P-codes by a compiler written in highly standard Pascal. This P-code is then interpreted by a procedure which forms an integral part of the compiler program.

Concurrency is simulated on single processor machines, for which CLANG was developed, by letting each active process run for a small random number of p-code steps, before switching to the next ready process. (The random numbers are obtained by a call to an external procedure.) Programs using concurrency may be expected to behave differently each time they are run.

The high level constructs for concurrent process synchronisation and communication, available in CLANG and examples of their use may be found in appendix A: The User Manual.

Many of the remaining examples in this thesis are presented in CLANG, but any reader familiar with Pascal should have no difficulty in following the CLANG code.

The similarities between CLANG and Pascal can easily be shown by means of the following example.

<u>Example:</u>

A program to find the factorials of integers from 0 to 8 may be coded in Pascal as:

```
program FINDFACTORIALS(INPUT, OUTPUT);
 var N: INTEGER;

  function FACTORIAL(N: INTEGER): INTEGER;
   begin
    if N = 0 then FACTORIAL := 1
    else FACTORIAL := N * FACTORIAL(N-1)
   end;   (*FACTORIAL*)

  begin   (*FINDFACTORIALS*)
   READ(N);
   while (N > 0) and (N < 8) do
    begin
     WRITELN('The factorial of ',N,'=',FACTORIAL(N));
     READ(N)
    end
   end.   (*FINDFACTORIALS*)
```

An equivalent program may be coded in CLANG as:

```
program FINDFACTORIALS;
 var N;

  function FACTORIAL(N);
   begin
    if N = 0 then FACTORIAL := 1
    else FACTORIAL := N * FACTORIAL(N-1)
   end;   (*FACTORIAL*)

  begin   (*FINDFACTORIALS*)
   read(N);
   while (N > 0) and (N < 8) do
    begin
     writeln('The factorial of ',N,'=',FACTORIAL(N));
     read(N)
    end
   end.   (*FINDFACTORIALS*)
```

Chapter 3: Fundamental problems of Concurrency

communicate v.t. & i : Impart, transmit
synchronise v.t. & i : occur at the same time, be simultaneous, co-ordinate

mutual a. : by each to(wards) the other
exclusive a. : shutting out; not admitting of

The concise Oxford dictionary

In order to co-operate, concurrently executing processes must synchronise and communicate.

Communication allows the execution of one process to influence that of another. Methods of interprocess communication include the use of shared variables (ie. variables that can be referenced by more than one process) and the sending and receiving of messages.

The concurrent processes may be executing asynchronously and thus synchronisation is often necessary so that the processes may communicate safely. Synchronisation can be viewed as a set of constraints on the ordering of events.

For example: If a variable must be updated by one process before it can be used by another, then these two processes must synchronise so that they can co-operate properly.

The interleaving in time of the execution of concurrent processes often makes it desirable that the execution of a certain sequence of statements appears to be an indivisible operation.

Consider this example:

Suppose initially that the value of a shared variable 'X' is 0 and that both process I and process II execute a statement that increments X by 1.

ie. X := X + 1

It would be reasonable to expect the final value of X, on completion of process I and process II's concurrent execution, to be 2. However, this will not always be the case as assignment statements are not generally implemented as one indivisible operation and thus the value of X might be 1 or 2.

Although the two processes may not be executing exactly the same statement this anomalous behavior arises from the fact that both processes are accessing the same variable and so to avoid this, the assignment statement for the shared variable concerned must be "protected" so as to prevent its execution being interleaved in time. This "protection" must mean that while one process is executing the assignment statement, if another process also wishes to execute a similar statement on the same variable, then this other process must be delayed until such time as the first process has finished executing its statement.

A sequence of statements that must appear to be executed as an indivisible operation is called a **critical section.**

The term **"mutual exclusion"** refers to mutually exclusive execution of critical sections.

Thus in the above example the assignment statement would have to be guarded by some form of mutual exclusion mechanism to ensure its correct execution.

Note: If two (or more) processes have no variables in common then their execution need not be mutually exclusive.

One traditional solution for ensuring mutual exclusion to a resource (eg. variables, data structures etc.) which needs to be shared by several concurrent processes is via the use of semaphores [Dij68].

A semaphore is conceptually a non-negative integer valued variable on which two operations are defined:

P (ie. wait) and V (ie. signal)

Given a semaphore S:

P(S) will delay the process executing it until $S > 0$ whereupon $S := S - 1$ will be executed; the test and decrement are executed as an indivisible operation.

V(S) executes S := S + 1 as an indivisible operation.

To implement mutual exclusion each critical section is preceded by a P operation and later followed by a V operation on the same semaphore.

Another situation in which it is necessary to coordinate the execution of concurrent processes occurs when a shared resource is in a state inappropriate for executing a particular operation. Any process attempting such an operation should be delayed until the state of the resource changes as a result of other processes performing operations on the resource.

This type of synchronisation of processes we have termed <u>conditioned synchronisation</u>.

In implementing conditioned synchronisation using semaphores, shared variables are used to represent the condition, and a semaphore associated with the condition is used to accomplish the synchronisation (an example is given below).

Although the semaphore is quite an elegant low level primitive and can be used as a general tool for solving synchronisation problems, a concurrent system built solely on semaphores is courting disaster if even one occurrence of a semaphore operation is mistaken anywhere.

When using semaphores, a programmer might forget to incorporate all statements that reference shared resources into critical sections. This could destroy the mutual exclusion required within these critical sections.

Another difficulty with using semaphores is that both conditioned synchronisation and mutual exclusion use the same pair of primitives. This makes it difficult to distinguish the purpose of a given wait or signal operation. Since mutual exclusion and conditioned synchronisation are distinct concepts they should have distinct notations.

Even the correct usage of semaphores leads to obscure programs (cf. first example below). This is because it is the responsibility of the programmer to ensure that the critical section is accessed in mutual exclusion, by means of correct usage of semaphores.

Therefore it follows that if the facilities to ensure this mutual exclusion were implicit in in the programming language itself, the programmer would be relieved of the burden, and furthermore the potential for compile time error checking would be introduced.

The two high level constructs introduced into the language CLANG to facilitate easy interprocess synchronisation and communication, the monitor and the synchroniser, will be dealt with in detail in the following chapters.

The differences between these three constructs can easily be illustrated (in CLANG programs) using a simple classic example, the so-called warehouse problem. A warehouse can only store one item at any one time, and has to deal with requests from a producer and a consumer (processes) who wish continuously to deposit and remove items respectively. The problem is further complicated by the need to prevent the consumer attempting to remove a non-existant item or the producer trying to deposit an item in the warehouse that might already be full.

Firstly the warehouse implemented by means of semaphores.

```
program CLASSICEXAMPLE;
 const DEPOSIT = 1;
       REMOVE = Ø;
       OCCUPIED = 1;
       UNOCCUPIED = Ø;
 var INSIDE, SPACE, SHOP,
     MUTEX, EMPTY, FULL;    (*semaphores*)

  procedure WAREHOUSE(var ITEM, OPERATION);
   begin   (*WAREHOUSE*)
    wait(MUTEX);    (*wait for mutual exclusion*)
    INSIDE  :=  INSIDE  + 1;   (*no. in WAREHOUSE*)
    if OPERATION = DEPOSIT then
     begin
      if SPACE = OCCUPIED then (*can't deposit yet*)
       begin
        signal(MUTEX);   (*release exclusivity*)
        wait(EMPTY)
       end;
      SPACE := OCCUPIED;
      SHOP := ITEM;   (*deposit item*)
      signal(FULL)
     end
    else
     begin
      if SPACE = UNOCCUPIED then
       begin
        signal(MUTEX);   (*release exclusivity*)
        wait(FULL)    (*wait for a deposit*)
       end;
```

```
    ITEM := SHOP;    (*remove item*)
    SPACE := UNOCCUPIED;
    signal(EMPTY)
   end;
  INSIDE := INSIDE - 1;
  if INSIDE = Ø then
   signal(MUTEX)   (*release exclusivity*)
 end;    (*WAREHOUSE*)


procedure PRODUCER;
 const SWEET = 1;
 var ITEM;
  begin
   repeat
    ITEM := SWEET;    (*produce item*)
    WAREHOUSE(ITEM, DEPOSIT)
   forever
  end;   (*PRODUCER*)


procedure CONSUMER;
 var ITEM, MOUTH;
  begin
   repeat
    WAREHOUSE(ITEM, REMOVE);
    MOUTH := ITEM   (*consume item*)
   forever
  end;   (*CONSUMER*)


begin   (*CLASSICEXAMPLE*)
 INSIDE := Ø;
 SPACE := UNOCCUPIED;   (*warehouse initially empty*)
 MUTEX := 1;
 EMPTY := Ø;
 FULL := Ø;
 cobegin
  PRODUCER;
  CONSUMER
 coend
end.   (*CLASSICEXAMPLE*)
```

<u>Aside:</u> It can be seen from the above that with a program that makes sole use of semaphores great care must be taken to avoid disaster.

<u>eg.</u> if the two statements

```
signal(MUTEX);
wait(EMPTY)
```

in the procedure WAREHOUSE had been reversed

```
wait(EMPTY);
signal(MUTEX)
```

deadlock (<u>ie.</u> disaster) would have resulted

The same warehouse can be coded as a monitor as follows:

```
program CLASSICEXAMPLE;

 monitor WAREHOUSE;
  (**************************************************)
  (* The   procedures   DEPOSIT  and   REMOVE   are *)
  (* exportable from the monitor. This is signified *)
  (* by   prefixing   their  declaration  with   an *)
  (* asterisk.                                      *)
  (**************************************************)
  const OCCUPIED = 1;
        UNOCCUPIED = Ø;
  var SHOP, SPACE;
  condition FULL, EMPTY;

   procedure *DEPOSIT(ITEM);
    begin
     if SPACE = OCCUPIED then EMPTY.qwait;
     SPACE := OCCUPIED;
     SHOP := ITEM;   (*deposit item*)
     FULL.qsignal
    end;   (*DEPOSIT*)

   procedure *REMOVE(var ITEM);
    begin
     if SPACE = UNOCCUPIED then FULL.qwait;
     ITEM := SHOP;   (*remove item*)
     SPACE := UNOCCUPIED;
     EMPTY.qsignal
    end;   (*REMOVE*)

  begin   (*WAREHOUSE*)
   SPACE := UNOCCUPIED  (*warehouse initially empty*)
  end;    (*WAREHOUSE*)
```

```
procedure PRODUCER;
 const SWEET = 1;
 var ITEM;
  begin
   repeat
    ITEM := SWEET;    (*produce item*)
    WAREHOUSE.DEPOSIT(ITEM)
   forever
  end;   (*PRODUCER*)


procedure CONSUMER;
 var ITEM, MOUTH;
  begin
   repeat
    WAREHOUSE.REMOVE(ITEM);
    MOUTH := ITEM    (*consume item*)
   forever
  end;   (*CONSUMER*)



begin   (*CLASSICEXAMPLE*)
 cobegin
  PRODUCER;
  CONSUMER
 coend
end.   (*CLASSICEXAMPLE*)
```

Finally the warehouse may be coded as a synchroniser:

```
program CLASSICEXAMPLE;

 synchroniser WAREHOUSE;
  (**************************************************)
  (* The  sequential  positioning  of  the   accept *)
  (* statements  for a DEPOSIT and a REMOVE request *)
  (* ensures that the  order  of  these  operations *)
  (* is correct.                                    *)
  (**************************************************)
  var SHOP;
  entry DEPOSIT(ITEM), REMOVE(var ITEM);

   begin
    repeat
     accept DEPOSIT(ITEM) then
      begin
       SHOP := ITEM
      end;
```

```
     accept REMOVE(var ITEM) then
      begin
       ITEM := SHOP
      end
    forever
   end;   (*WAREHOUSE*)


  procedure PRODUCER;
   const SWEET = 1;
   var ITEM;
    begin
     repeat
      ITEM := SWEET;   (*produce item*)
      WAREHOUSE.DEPOSIT(ITEM)
     forever
    end;   (*PRODUCER*)


  procedure CONSUMER;
   var ITEM, MOUTH;
    begin
     repeat
      WAREHOUSE.REMOVE(ITEM);
      MOUTH := ITEM   (*consume item*)
     forever
    end;   (*CONSUMER*)


begin   (*CLASSICEXAMPLE*)
 cobegin
  WAREHOUSE;  (*a synchroniser is an active process*)
  PRODUCER;
  CONSUMER
 coend
end.   (*CLASSICEXAMPLE*)
```

Chapter 4: The Monitor concept

> "... a collection of associated data and procedures is known as a monitor ... it is essential that only one [sub]program at a time actually succeed in entering a monitor procedure, and any subsequent call must be held up until the previous call has been completed."
>
> C.A.R. Hoare [Hoa74]

The need for a construct, whereby the programmer would be relieved of the tedium of explicitly ensuring mutual exclusion around a critical section, and the possibility of compile time error checking could be introduced, led to Brinch Hansen [Bri72], [Bri73] and Hoare [Hoa74] developing the idea of a monitor.

A monitor is a construct used local to a program. It is formed by encapsulating data structures, which may be shared by concurrent processes, with a set of procedures / functions which access those structures. A monitor may also incorporate other operations (such as initialisation code) which might be needed on the data structure, but which must be hidden from the processes which access the data structure.

A process has exclusive access to the shared data while it is executing a monitor procedure / function. This exclusivity is provided by the monitor itself and relieves the programmer of the burden of having to build his or her own exclusion code.

Monitors thus provide a high level construct for ensuring mutually exclusive access to a shared resource. However, monitors by themselves provide no means of conditioned synchronisation, and thus must be supplemented by such features as condition variables to facilitate this.

The queueing of processes is an essential factor of the monitor concept. If simultaneous access is requested to a monitor by several processes then some "fair" queueing must be effected at the "entrance" to the monitor to ensure that only one process has exclusivity to a monitor, and also to ensure that another process will be granted exclusivity as soon as it becomes available again. Similarly, with condition variables the queueing of processes is necessary if the data structure is not in the required state. Lastly, some form of "polite" queue may be necessary, when a process signals another process waiting (suspended) within the monitor, if this signalling process is to be suspended until the signalled process has completed its activities in the monitor.

Not all the procedures / functions of a monitor are available to the processes that wish to access the data structure; those that are, are typically flagged as such. In CLANG this flagging takes the form of prefixing the declaration of the procedures / functions concerned by means of an asterisk ('*') (cf. chapter 3, the warehouse coded as a monitor).

These flagged procedures / functions are then typically accessed by preceding the call with the name of the corresponding monitor as in:

monitorname.subprogramname

4.1 Problems associated with the monitor concept

Deadlock : utter standstill

Invariant : unchangeable, always the same

The concise Oxford dictionary

The monitor construct ensures that only one process may be active "inside" a monitor at any one time. This process is said to have exclusivity to that monitor.

When a process releases exclusivity to a monitor, that monitor is accessible to any concurrent process. This exclusivity may be released either as the result of the process finishing within the monitor and thus leaving it, or as a result of the process being suspended.

Should a process request exclusivity to a monitor and find it unavailable, then the process is suspended on an implicit "entry" queue associated with that monitor until such time as the exclusivity becomes available.

Conditioned synchronisation is not provided by the monitor construct itself, so if this is required within a monitor, additional constructs are necessary. These constructs typically consist of some sort of explicit condition queue on which a process can WAIT, ie. be suspended, until it is given the go-ahead to continue by some form of SIGNAL from another process.

Thus a process can be suspended "outside" a monitor waiting for exclusivity, or "inside" a monitor on some condition queue.

Due to the hierarchical structure of programs it is possible to call a monitor procedure / function from within another monitor declared after it. This is known as a nested monitor call.

For example

```
monitor MON1;

 procedure *PROC1;
  begin
   (*some statements*)
  end;  (*PROC1*)
  ...
 begin   (*MON1*)
  (*body of MON1*)
 end;    (*MON1*)


monitor MON2;
 procedure *PROC2;
  begin
   ...
   MON1.PROC1;    <--- nested monitor call
   ...
  end;   (*PROC2*)
 ...

 begin   (*MON2*)
  (*body of MON2*)
 end;    (*MON2*)
```

The nested monitor call implies that it is possible for a concurrent process to be holding exclusivity to several monitors when it is suspended. This possibility heralds a problem area relating to monitors. ([Lis77], [Had77], [Kee78], [Par78], [Wet78])

Before discussing the problem, we must introduce some new terminology:

A PLOXY point (standing for Possible Loss Of eXclusivitY) is a point in the code of a monitor where a concurrent process might be forced to suspend itself and, because of this, release exclusivity to the monitors it occupied.

Just how many exclusivities are released will be detailed shortly.

There are two types of PLOXY points:

A nested PLOXY point is a PLOXY point that results from a concurrent process executing a nested monitor call.

A conditioned PLOXY point is a PLOXY point that results from a concurrent process executing certain operations relating to a condition queue.

The degree to which a programming language tackles the question of which exclusivities are to be released by a concurrent process at a PLOXY point shows the potential of the language for solving the related problems of certain deadlock, potential deadlock, and loss of parallelism.

Certain deadlock

The condition queues are themselves shared data structures to be protected by the monitor. This implies that all the operations on a condition queue, eg. Waits and Signals, relating to a shared data structure must be local to the monitor which encapsulates it.

This is significant because, if a process on being suspended "inside" a monitor as the result of a WAIT operation on a condition, does not release exclusivity to that monitor, then deadlock will result, as no other process will be able to enter the monitor to perform the corresponding SIGNAL. (Similarly, if the language forces the signalling process to be temporarily suspended if there is a process waiting on that condition, then deadlock will again result if the exclusivity is not released by the signalling process.)

Potential deadlock

A process can acquire a set of monitor exclusivities by performing a series of nested monitor calls.

Potential deadlock can exist if a process, on being suspended at a conditioned PLOXY point, does not release all the exclusivities it is holding.

Should a process, after a series of nested monitor calls, be suspended in a monitor (say MON1) on a conditioned WAIT operation then it must remain suspended until the corresponding SIGNAL operation is forthcoming from another process. Suppose this other process could not call the monitor MON1 directly, but first needed to gain access to other monitors. If the exclusivity to these other monitors is still held by the suspended process, then deadlock will result as the SIGNAL can then never be performed.

Loss of parallelism

When a process requests exclusivity to a monitor and some other process is already busy inside that monitor then the requesting process is suspended until such time as the other process has released exclusivity to the monitor.

If, after a series of nested monitor calls, a process is suspended at a PLOXY point without releasing the held exclusivities then there is a potential loss of parallelism as no other process will be able to gain access to those held exclusivities for at least the duration of the suspension.

Differences in approach

It has been shown that failure to release exclusivities can lead to the problems of deadlock and loss of parallelism, but even the releasing of exclusivities can give rise to problems.

Released exclusivities should be reacquired before a reactivated process be allowed to continue.

(Aside: This is not really necessary as a process only needs the additional exclusivities when it leaves the monitor for which the granting of exclusivity caused reactivation. However for ease of implementation it is generally preferable for all the desired exclusivities to be reacquired before allowing the reactivated process to be available, once more, for scheduling.)

This can lead to a reactivated process remaining delayed waiting for the exclusivities that it released and which other processes might currently be holding.

Invariance of monitor variables

This possible delay with the release-and-reacquire approach is a fairly minor problem when compared with the need to establish the invariance of the monitor variables at the PLOXY points.

A process on reacquiring a monitor's exclusivity after being suspended might reasonably expect to find the values of many, if not most, of that monitor's variables in the same state as when the exclusivity was released. This, of course, might not always be the case, as once the exclusivity is released other process are free to gain access to that monitor and so alter the values of the variables.

However, it is not always desirable to ensure all the monitor variables are invariant. This can best be illustrated by means of examples.

The first example shows a case where it is important for the monitor's variables to be invariant, while the second example shows a case where it is desirable for at least some of the monitor's variables to be subject to alteration between the time of releasing the exclusivity and when it is reacquired.

Example 1

```
monitor MONIT1;
 var LOOP;

  procedure *A;
   begin
    LOOP := 0;
    while LOOP < 5 do
     begin
      LOOP := LOOP + 1;
      --- PLOXY point ---
      (*some operations*)
     end   (*while*)
   end;   (*A*)
```

```
procedure *B;
 begin
  LOOP := 6
 end;

 ...
```

Should a process (say process I) gain access to monitor MONIT1 and during the course of executing procedure A, be forced to release the exclusivity at the PLOXY point, on reacquiring the exclusivity the value of the variable LOOP may not be 1 as expected, but rather 6 if another process gained access to procedure B during the interim of process I's suspension. This might be totally unacceptable.

Example 2

```
monitor MONIT2;
 var BUSY;
 condition FULL;

  procedure *A;
   begin
    while BUSY = Ø do
      FULL.qwait;     (*conditioned PLOXY point*)
    (*some operations*)
   end;   (*A*)

  procedure *B;
   begin
    BUSY := 1;
    FULL.qsignal    (*corresponding signal*)
   end;   (*B*)

  begin   (*MONIT2*)
   BUSY := Ø    (*initial value of BUSY*)
  end;    (*MONIT2*)
```

A process (say process I) executing procedure A of monitor MONIT2 would be suspended on the condition FULL as the value of the variable BUSY has been initialised to Ø. When another process (say process II) subsequently gains exclusivity to monitor MONIT2 and executes procedure B, the value of BUSY will be set to 1 and the corresponding qsignal desired by process I will be performed.

Process I will be reactivated, but if the variable BUSY is invariant, process I will still find its value to be Ø, and will thus again be suspended on the condition FULL.

Here it is desirable, in order to circumvent the infinite loop in procedure A, to have the variable BUSY susceptible to alteration.

These examples may seem somewhat contrived, but they do serve to illustrate the conflicting needs associated with monitor variable invariance.

The following sections will examine the degree to which each of the four languages, Concurrent Pascal, Edison, Modula-2 and Pascal Plus, all of which support monitor like facilities, deal with the aforementioned problems.

This information will then be contrasted with the way in which CLANG tackles the problems.

4.2 The monitor concept in other languages

Having originated over ten years ago, it is only natural that the monitor concept has been implemented to a lesser or greater degree in a number of languages.

This section will be concerned with four of these languages, and more particularly the extent to which they endeavour to overcome the problems outlined in the previous section.

Study of these languages has been greatly hampered by their unavailability (apart from Modula-2) for practical evaluation of certain questions relating to them.

The assessment of Concurrent Pascal, Edison and Pascal Plus has been done from a purely theoretical knowledge gleaned from the relevant articles and manuals published concerning them, [Bri75], [Bri77], [Col79], [Col80], [Har77]; [Bri81], [And83]; [Bus80], [Bus 82], [Wel79], [Wel80]. The author admits that it is possible that some of the conclusions may not be totally valid on some implementations.

Common features

Each of the four languages (and CLANG) supports a monitor like construct which entails:

(1) A language construct that encapsulates the data structure that may be "shared" by concurrent processes;

(2) Subprograms, such as procedures or functions, contained within this construct that will perform the desired operations on this data structure: These subprograms may be totally invisible "outside" the construct, or be flagged as being accessible;

(3) Variables which effectively exist at the global level of the program and which may or may not be flagged as exportable - possibly only in a "read only" capacity;

(4) The identifiers that are accessible outside the monitors are typically accessed by appending the identifier with the name of the monitor in which it was declared separated by a period - for example:

monitorname.identifier

(5) Initialisation code which will be performed on the variables of this "monitor like" construct before any processes attempt to access the data structure; and

(6) Some form of condition variable which can be used to provide conditioned synchronisation within the construct.

Any deviations from these basic principles will be highlighted, otherwise they will be assumed part of each language's constructs.

Example

The warehouse (as mentioned in chapter 3) may be coded as a monitor:

```
monitor WAREHOUSE;
 const OCCUPIED = 1;
       UNOCCUPIED = Ø;
 var SHOP, SPACE;
 condition FULL, EMPTY;

  procedure *DEPOSIT(ITEM);
   begin
    if SPACE = OCCUPIED then EMPTY.qwait;
    SPACE := OCCUPIED;
    SHOP := ITEM;   (*deposit item*)
    FULL.qsignal
   end;   (*DEPOSIT*)

  procedure *REMOVE(var ITEM);
   begin
    if SPACE = UNOCCUPIED then FULL.qwait;
    ITEM := SHOP;   (*remove item*)
    SPACE := UNOCCUPIED;
    EMPTY.qsignal
   end;   (*REMOVE*)

  begin   (*WAREHOUSE*)
   SPACE := UNOCCUPIED  (*warehouse initially empty*)
  end;    (*WAREHOUSE*)
```

4.2.1 Concurrent Pascal

Concurrent Pascal was developed by Brinch Hansen from 1975 to 1977. Being the first language to support the monitor concept it provided a vehicle for evaluating monitors as a system structuring device. The language has subsequently been used to write several operating systems eg. Solo [Bri76], [Pow78].

A monitor can only be initialised once, by an init statement, which allocates storage for the shared variables and performs the initialisation of these. After initialisation the shared variables of a monitor exist forever and are known as permanent variables.

The parameters and local variables of a monitor procedure only exist while it is being executed and are known as temporary variables.

A monitor procedure can only access its own temporary and permanent variables. These variables are not accessible to other system components. Other components may only call procedure entries (which are those procedures that are explicitly designated as visible from "outside" the monitor).

Only monitors and constants can be permanent parameters of processes and monitors, which ensures that processes can only communicate via monitors.

In Concurrent Pascal conditioned synchronisation is achieved by means of a standard type QUEUE. A variable of type QUEUE may only be declared as a permanent variable within a monitor type.

The operations that can be performed on a variable (say) Q of type QUEUE are:

empty(Q) : True or false depending whether the queue is empty or not.

delay(Q) : The calling process is delayed on the queue Q and loses its exclusive access to the given monitor's data structure. The monitor can then be accessed by other processes.

continue (Q) : The calling process returns from the monitor procedure in which the continue operation was performed. If another process is waiting on the queue Q, that process will immediately resume execution from its point of delay. The resumed process again has exclusive access to the monitor's data structures.

Tackling the problems

To prevent deadlock of monitor calls and to ensure that access rights are hierarchical, the prevention of a system type calling its own procedure entries is enforced.

Concurrent Pascal uses the (so called) current monitor release approach. A process will only release exclusion on the current monitor in a chain of nested monitor calls when it performs a delay operation. Similarly, for the continue operation only the exclusivity to the monitor in which the operation is performed is released (in any case the signalling process has to return from the monitor immediately).

No attempt is made to release any exclusivities should a process become blocked by a nested monitor call.

This approach has simplicity to recommend it, but as mentioned in section 4.1, the problems of system's response degradation and potential deadlock are raised.

By only releasing the current monitor in its chain of exclusivities a process can at least be guaranteed the invariance of the permanent variables of the monitors whose exclusivitiy was not released. However, there does not appear to be any provision for guaranteeing the invariance of the permanent variables of the monitor whose exclusivity is released.

Additional advantages / disadvantages

Concurrent Pascal's facilities for conditioned synchronisation have several flaws: basically a variable of type QUEUE is not a queue.

The standard type QUEUE may be used within a monitor type to delay and resume the execution of a calling process within a procedure entry. However, at any time no more than one process can wait on a single queue. (Aside: Nowhere in the literature, [Bri75], [Bri77], [Col79], [Col80], [Har77], is any mention made as to what will happen if two (or more) processes attempt to delay on the same queue - presumably some sort of run time error will result).

This means that any multiprocess queue will have to be explicitly defined by the programmer as an array of single process queues [Bri77].

eg. type MULTIQUEUE = array (. 0..qlength -1 .) of QUEUE

where qlength is the upper bound on the number of concurrent processes in the system.

The continue operation on a variable of type QUEUE makes the calling process return from its monitor call. This implies that any further statements following the continue will be ignored.

```
eg. "statements"
     if LENGTH = Ø then continue(Q);
     "further statements - ignored"
    end;    "of monitor procedure"
```

This problem can be minimised by careful positioning of the continue operation, but again the emphasis is on the programmer to undertake this, (although there is no way the signalled process can directly influence the signalling process).

4.2.2 Edison

The programming language Edison is a second design effort by Brinch Hansen, based on his five years of experience with the languages Pascal and Concurrent Pascal. Its design goals were not to introduce new ideas, but to combine the proven concepts into a language that is simpler than Pascal, yet more powerful than the combination of Pascal and Concurrent Pascal. Edison has been available since July 1980.

Edison does not include the monitor construct as such; instead a monitor can be constructed by the programmer as a module in which the procedure bodies consist of a single when statement.

The Edison module has data abstraction facilities like those of the general monitor. To implement mutual exclusion, Edison makes use of a simplified version of the conditional critical region (originally proposed by Hoare in 1972 [Hoa72]).

The form of the when statement in Edison is:

```
-- when --- BOOLEAN CONDITION --- do --┬- STATEMENT ┬--- end --
                                       \---- ; -----/
```

Conditioned synchronisation can be achieved by careful choice of the BOOLEAN CONDITION as part of a when statement.

Tackling the problems

The problems of which exclusivities to release to prevent potential deadlock and the invariance of monitor variables do not occur, as in Edison the technique used to control the execution of the critical phases of when statements, is one of global exclusion, ie. The execution of all when statements takes place strictly one at a time.

A process executes a when statement in two phases:

(1) Synchronising phase: The process is delayed until no other process is executing the critical phase of a when statement.

(2) Critical phase: The Boolean condition is evaluated. If the value TRUE is returned then the statements contained within the when statement are executed and the execution of the when statement is completed. If the value FALSE is returned, then the process returns to the synchronising phase.

Each synchronising phase of a process only lasts a finite time provided that the critical phases of all other concurrent processes terminate.

If several processes need to evaluate (or re-evaluate) the scheduling conditions simultaneously, the implementation must guarantee that they do so one at a time in some "fair" order (eg. first-in-first-out).

There is thus no implicit manner available in Edison to prevent several processes operating on a shared variable simultaneously with generably unpredictable results. However, by restricting the operations on shared variables to well defined disciplines under the control of modules and when statements it is possible for the programmer to formulate concurrent statements that make predictable use of such variables.

In Edison the concepts of modularity, concurrency and synchronisation have been separated. This admittedly results in a more flexible language, being based on fewer concepts. Also it is still possible to achieve the same security as in (say) Concurrent Pascal, by the user adopting a programming style that corresponds to the processes and monitors of Concurrent Pascal.

> eg. A "monitor" can be constructed using the simpler concepts of modules, variables, procedures and when statements.

However the responsibility to ensure this security is placed squarely on the shoulders of the programmer and there are few or no safeguards to prevent the programmer breaking the structuring rules and so writing meaningless programs with a very erratic behaviour.

4.2.3 Modula-2

The programming language Modula-2 is a descendant of its direct ancestors Pascal and Modula. It was developed in 1979 by Wirth, [Wir83], and includes all the aspects of Pascal with the extensions of the module concept and multiprogramming. It has been developed as a general, efficiently implementable, systems programming language. Modula-2 was released for general usage in March 1981 and is small enough to allow efficient program development on 8-bit microcomputers.

Modula-2 forgoes the high level multiprocessing concepts in favour of lower level coroutines. Coroutines are procedures which execute independently but not concurrently and which communicate by transferring control to one another (rather than by call-return). In a coroutine transfer, the transferring coroutine becomes inactive and the transferred coroutine resumes execution.

Thus the process (ie. coroutine) monopolises the processor until such time as it wishes to relinquish it. This process switch occurs when:

(a) a new process is initiated

(b) a SEND or WAIT operation is executed.

Process switching is thus, in its simplest form, entirely under the control of the programmer and any suggestion of concurrency would appear to be absent. In this case no monitor construct seems necessary. Modula-2, however, does have the saving grace in that the occurrence of hardware interupts can interfere with the execution of a process and be made to effect a process switch. This can be used to launch pseudo-concurrent processes, for example by allowing each process to run for a certain amount of time before a process switch is caused by a clock interrupt [Sew84].

The specifications for the language Modula-2 do not contain any concurrent features, but the library module facilities allow these to be created by the programmer. In his book on the language, [Wir83], Wirth suggests one such library module for implementing conditioned synchronisation and so effecting a process switch. This, together with the definition of a process ring (also to be contained in the library module), and the specification of a priority in the heading of a module to control the interrupting of the executing process (an intrinsic feature of the language), will result in a "monitor like" construct.

This suggestion is available in the Volition Systems' implementation of Modula-2 (which is used at Rhodes University) and so will be assessed here in the enviroment of concurrent processes being simulated by switching processes on the receipt of an interrupt.

Wirth's suggestion consists of the following operations which can be performed on a condition (say) S:

WAIT(S) - appends the calling process at the end of the list designated by S. A process switch is effected and any process that is ready to run may gain control of the processor.

SEND(S) - takes the first element off the list designated by S and transfers control of the processor from the sending process to that process.

Tackling the problems

The specification of the priority in a "monitor" module heading is vital in ensuring mutual exclusion of that monitor's code. The reason for the priority is that the sequential execution of any monitor procedure can only be disrupted by the occurrence of an interrupt having a priority in excess of that assigned to the particular monitor module. Thus a sufficiently high module priority precludes the interruption of the execution of any monitor procedure.

If the priority specified is not sufficiently high to prevent an interruption of the process in that module (and a consequent process switch) there are no safeguards to ensure the mutual exclusion of the monitor data.

Assuming the priority assigned to a module is high enough, if the code within that module is extensive the loss of parallelism amongst the processes could be significant as no other process could proceed until the one currently executing is either suspended or exits the monitor and renders the program once more susceptible to interrupt.

This simplistic method of exclusion does remove the problem of nested monitor calls being unsuccessful, but again the loss of parallelism must be emphasised.

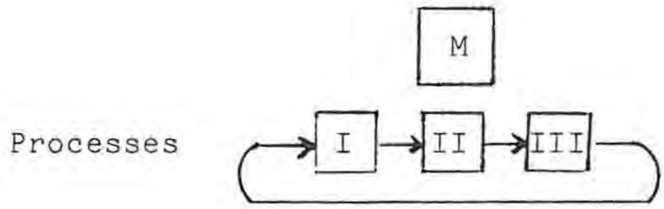
Implementing conditioned synchronisation by means of the WAIT and SEND operations is fraught with dangers.

Although the WAIT operation will cause a process switch and thus effectively cause the process performing such an operation to "release" all its exclusivities, a process performing a SEND operation is NOT assured of regaining the exclusivities it transferred to the reactivated process as soon as this process subsequently releases these exclusivities.

A break down of the mutual exclusion of a monitor module can easily occur.

Consider the following example

Suppose a system consists of a monitor M and three processes.



If the process scheduling is done via a counter-clockwise cyclic scan (a reasonable assumption), then the use of WAITs and SENDs in the following possible sequence of events can cause the break down:

Process I enters the monitor M and performs a WAIT(S) operation on some condition S in that monitor.

The resulting process switch means that process II starts to execute. Process II now performs some operations until interrupted by the clock, causing a process switch to process III.

Process III gains access to the monitor M and somewhere inside the code performs a SEND(S) operation thus transferring control to the now reactivated process I.

When process I subsequently releases exclusivity to monitor M, either by a subsequent wait operation or by leaving the monitor and being interrupted, it is process II that gains the use of the processor.

There is now nothing to stop process II from entering monitor M, despite the fact that process III is still "suspended" somewhere in there.

This situation is clearly contrary to the idea of a critical section.

Note: It is not the scheduling algorithm that is at fault, but rather the carte blanche way in which a process releases exclusivity without concern for any process which might still be "temporarily suspended" waiting for that exclusivity.

No attempt is made to ensure the invariance of any monitor variables from the time the exclusivity to a monitor is released by means of the WAIT or SEND operation, and when it is subsequently reacquired.

The WAIT and SEND operations "work" in the limited enviroment provided in the basic specifications of Modula-2, but as soon as any form of concurrency (albeit pseudo concurrency) is introduced by means of arbitrary process switching instead of only at specific points, Wirth's suggestion falls well short of what a programmer might expect in order to construct reliable concurrent programs.

The library facilities of Modula-2 do, however, give the user the opportunity for developing other, perhaps better, methods for expressing synchronisation and communication between concurrent processes.

One such extension to Modula-2, to provide a more Hoare-like monitor has been implemented, albeit imperfectly (because of other limitations in Modula-2), by a colleague D. Sewry [Sew84].

In Sewry's extensions, concurrency is simulated by allowing each process a small portion of processor time and using an interrupt driven process switch by means of an internal clock. (All this work was done on the Sage IV microcomputer which has an internal clock.)

Mutual exclusion is achieved by compelling all monitor module procedures, that are to be visible for access outside the monitor, to execute, as their first statement, a call to a routine to effect the gaining of exclusivity, and as their last statement a call to a routine to effect the releasing of exclusivity.

(A similar extension was undertaken with U.C.S.D. Pascal by Boddy, [Bod83], [Bod84].)

Should another process be busy in the monitor, the process attempting to gain access will be suspended until such time as the exclusivity becomes available.

Conditioned synchronisation is provided by the following modified operations on some condition S:

WAITCONDITION(S) - suspends the process performing the operation on a waitcondition queue designated by S. The process releases exclusivity to the monitor.

SENDCONDITION(S) - if there is no process waiting on the condition S then this operation has no effect, otherwise the process performing the operation is suspended on a "polite" queue and the process at the head of the condition queue regains exclusivity and continues execution. The process that performed the SENDCONDITION will remain suspended until the reactivated process releases exclusivity to the monitor.

Note:

When the exclusivity to a monitor is released, it is not immediately made available to any process, but rather the "polite" queue and then the queue for processes waiting for exclusivity, are scanned for any suspended process. The first one found is granted the exclusivity. If no processes are waiting, only then is the exclusivity made available for any subsequent requests.

The main restriction in Sewry's extensions is that only one monitor is allowed per program - this does away with the problems relating to nested monitor calls.

There are no facilities to ensure the invariance of the monitor variables during a process' period of suspension.

What the extensions do show is the existing potential of Modula-2 for allowing a programmer explicitly to develop his or her own constructs for allowing synchronisation and communication between concurrent processes.

4.2.4 Pascal Plus

The language Pascal Plus was developed in 1979 by Welsh and Bustard under the guidance of Hoare. While maintaining Pascal as a subset, Pascal Plus contains major extensions in the fields of data abstraction and concurrency. Its design objective was to provide tools which would encourage a programmer to construct well engineered solutions to problems on hand.

One of the major advances in Pascal Plus over the other languages in the fields of processes and monitors, is that in Pascal Plus processes and monitors may be defined as a type, which allows "instances" of these types to be declared, [Bus80].

Another new feature included is that of initialisation and finalisation code of a monitor, separated by what is termed an inner statement (denoted by '***'). The inner statement of a monitor also readies any processes, declared local to it, for concurrent execution. It is the inner statement in the main program which, when executed, activates all processes "simultaneously". By this stage all the intialisation code of the monitors will have been done. Once activated all processes proceed "simultaneously" (depending on whether the implementation is on a single- or multiprocessor system) until they all terminate.

When all the processes have stopped the code following the inner statement in the main program is executed followed by the finalisation code for all the monitors. The drawback of launching concurrency in this fashion is that for concurrency to be launched more than once per program the inner statement in the main program must be contained in some sort of loop and if certain process are not to be launched every time additional "fiddles" will have to be inserted into their code to achieve this.

Those identifiers of a monitor which need to be visible from outside the monitor are known as starred identifiers and their declarations are preceded by an asterisk ('*'). All unstarred identifiers are invisible and thus inaccessible outside the monitor in which they were declared.

Extensive facilities are provided in Pascal Plus for conditioned synchronisation. This is achieved through a standard monitor called condition, the underlying workings of which are hidden from the user and only the interface given below is visible. Associated with each instance of CONDITION is an ordered queue on which processes may be temporarily suspended until they are able to continue.

```
monitor CONDITION;
 type RANGE = 0..MAXINT;

  procedure *PWAIT(PRIORITY:RANGE);
   (* suspends  the  process calling it on  the  condition
      queue  with  the priority specified by PRIORITY  - a
      low  value  indicates a high  priority  status.  The
      suspended process is positioned behind all processes
      with a higher or equal priority status.             *)

  procedure *WAIT;
   (* suspends  the  process calling it on  the  condition
      queue  with a default priority of 'MAXINT  div  2' *)

  procedure *RETURN;
   (* restores a process to a condition queue after it has
      been activated temporarily                          *)

  procedure *SIGNAL;
   (* activates  the  process at the head of  a  condition
      queue. If the queue is empty a SIGNAL has no effect. *)

  function *EMPTY:BOOLEAN;
   (* returns  TRUE  if  the  condition  queue  is  empty;
      otherwise false.                                    *)

  function *LENGTH:RANGE;
   (* gives  the  number  of  processes  suspended  on  a
      condition queue                                     *)

  function *PRIORITY:RANGE;
   (* returns  the  priority value of the process  at  the
      head of the relevant condition queue.               *)
```

Tackling the problems

The designers of Pascal Plus believed that the monitors in a program represent a potential bottleneck, and so every precaution is taken to ensure that a process is never delayed unnecessarily while executing monitor code. This has led to the scheduling descision that a process in a monitor has a high priority, overriding its run priority and thus a process' execution of code in a monitor is allowed to run to completion without it losing control of the processor.

The literature, [Bus80], [Wel80], states that mutual exclusion in Pascal Plus can be implemented in one of two ways:

(1) On single processor machines, or multiprocessor machines where very little time will be spent executing monitor code, a global exclusion mechanism is used.

(2) On multiprocessor machines where monitor code might be more involved, a separate exclusion mechanism semaphore is maintained for each monitor.

(The author presumes the type of implementation is dependent on the machine on which Pascal Plus is running.)

In the case of (1) the problems of deadlock and the invariance of monitor variables relating to the nested monitor call do not arise, (but see below for conditioned synchronisation), as the global exclusion ensures that once one process is executing in a monitor; no other processes are allowed to gain exclusivity to any monitors and thus no nested monitor call can be unsuccessful.

Should a nested monitor call be unsuccessful in case (2), the process performing such a call is made to wait but does <u>NOT</u> release the exclusivity to any monitor it might already be holding. Again the problem of invariance of monitor variables does not arise and potential deadlock does not occur.

However, with both cases (1) and (2) the problem of loss of potential parallelism is prevalent. The seriousness of this problem depends (in case (1)) on how long a process is busy inside the monitors, or (in case (2)) on how long a process' nested monitor call is blocked and how many exclusivities that process might hold.

With regard to conditioned synchronisation the following rules relating to monitor exclusivities apply:

The PWAIT, WAIT and RETURN operations cause the release of all exclusivities to monitors which the process performing the operation might hold.

On performing a SIGNAL operation the signalling process transfers the exclusivity to the monitor (in which the operation takes place) to the process at the head of the relevant condition queue. If the SIGNAL operation is the last operation in a monitor procedure / function then the signalling process can leave the monitor and continue to run, otherwise the signalling process is delayed until the signalled process subsequently releases exclusivity to the monitor, either by leaving it or again being suspended via one of the conditioned wait operations. Should the signalled process in turn perform a SIGNAL then it is delayed in the same way.

When a SIGNAL is complete, the exclusivity to the monitor concerned is transferred back to the process that issued the signal. The resumed process then continues executing from the point where the signal was performed.

If there is no process on the condition queue then the SIGNAL operation has no effect.

With the conditioned synchronisation operations there does not appear to be any form of guarantee for the invariance of any monitor variables. The problem does not only occur in the monitor in which the conditioned synchronisation takes place, but also in any monitors to which the signalled process might have made successful nested monitor calls prior to being suspended.

Additional advantages / disadvantages

Variables of monitors which are declared as starred identifiers may be inspected outside the monitor by more than one process at a time. These variables may not be altered outside the monitor, but only via a procedure or function of the monitor in which they were declared (ie. inside the monitor). Thus several processes may be inspecting a monitor variable while another may be modifying it. The hardware ensures that the inspecting processes do not get a meaningless value, but it could be either the value just before, or just after the modification.

Once a process has entered a monitor by invoking a monitor procedure / function it is then free to call any other procedure or function of that monitor or even the initial procedure / function recursively.

If deadlock occurs, the main program is reactivated and the execution of the monitor bodies is completed. This has the advantage of recovery after deadlock, but it does mean that a program could finish running and produce spurious results propogated due to the processes' non-completion.

Apart from the non-guaranteeing of the invariance of the monitor variables at conditioned PLOXY points, Pascal Plus, more than any of those languages assessed so far, provides the mutual exclusion and conditioned synchronisation facilities necessary to ease the programmer's task of controlling concurrent process synchronisation and communication.

Many of the constructs in CLANG have been based on those of Pascal Plus.

4.3 Using and implementing monitors in CLANG - practical details

This section will examine how CLANG deals with the problems associated with the monitor concept. The examination will consist of two parts:

(1) A look at the features available in CLANG for the programmer to overcome these problems.

(2) A description, illustrated by worked examples and flow diagrams, on how the features shown in (1) were implemented.

The exact syntax of the monitor and its associated constructs may be found in the Appendix A: The User Manual, and a full listing of the Pascal code making up the CLANG compiler and interpreter may be found in Appendix B.

Monitors in CLANG may only be declared in the main block. This differs from Pascal Plus which allows monitor declarations to be nested inside other monitors. (Note: This is different from a nested monitor call.)

There appears to be no need to declare monitors local to another monitor block. This can be shown from one definition of a monitor:

> "A monitor is declared to ensure mutually exclusive access to a critical region dealing with the shared data structure" [Cha83].

Only one concurrent process should be active inside a monitor at any given time. If the nesting of monitor declarations was allowed, the nested monitor declaration would provide mutual exclusion in an area of code in which mutual exclusion is already guaranteed.

Conditioned synchronisation in CLANG is achieved via condition variables and the operations which can be performed on them.

Condition variables may only be declared local to monitors and are not variables in the "true" sense, but rather implicit queues on which the operations qpwait(PRIORITY), qwait, qsignal, queue and qlength can be performed. These operations are essentially as those in Pascal Plus.

Tackling the problems

As CLANG has been designed as a possible language for teaching concurrency, every effort has been made to address the problems associated with the monitor and condition variables.

CLANG has been developed to run on single processor machines, concurrency being simulated by "sharing" the processor by allowing each process to run for a small random number of steps before effecting a process switch. A process can be in one of four states:

running - actually executing with control of the processor

ready - waiting to be scheduled

suspended - waiting for some event that will return it to the ready state, and

terminated - finished execution

To minimise the loss of potential parallelism on a single processor machine it is necessary for a process, on being suspended, to release all the exclusivities it might hold.

The suspended process may not proceed until it has received the go-ahead to continue by another process, releasing the desired exclusivity or performing the necessary qsignal, and then has reacquired all those exclusivities it released on suspension.

This does mean there might be a delay before a reactivated process is restored to the ready state, but, as will be shown in the section on implementation, the use of priorities ensures that this delay time is kept to a minimum.

What happens when a process performs a qsignal operation on a condition variable is slightly more involved.

If there is no process waiting for the signal then a qsignal operation has no effect.

If there is a process waiting then the signalling process is temporarily suspended and the signalled process is reactivated. The signalling process will remain suspended until the signalled process releases the exclusivity to the monitor concerned, either by leaving it, or by performing a subsequent qwait or qpwait(PRIORITY) operation.

The signalling process is being "polite" in allowing the signalled process to continue and therefore the signalling process should be allowed to continue execution as soon after the signalled process has released the exclusivity as possible. Thus on being temporarily suspended, the process performing a qsignal operation does not release all the exclusivities that it might hold, but rather, only those which the signalled process needs to return to the ready state and continue executing.

Even with the exclusivities transferred to it by the signalling process, the signalled process may still not have all its required exclusivities to continue. How the signalled process acquires those exclusivities will be explained in the section on implementation details; it suffices to say here that a signalled process has the highest priority for acquiring its desired exclusivities when other processes release them.

If the signalled process in turn performs a qsignal before it releases the exclusivity of the monitor concerned, then it is also temporarily suspended on the "polite" queue, in front of the process which originally signalled it, which implies it will regain the exclusivity before this process.

Invariance of monitor variables - methods

When a process regains the exclusivities it released on being suspended, it might expect to find certain of the variables internal to those monitors in a certain state.

The release-and-reacquire approach to monitor exclusivities, adopted in CLANG, has made the ensuring of invariance of monitor variables necessary.

As was shown in examples 1 and 2 of section 4.1, there are conflicting desires when it comes to what monitor variables need to be invariant.

CLANG tackles this problem by dealing with each of the two types of PLOXY point individually. The motivation for this is:

> A programmer will be unable to predict with any certainty at a nested PLOXY point whether the nested monitor call will be successful or blocked. On the other hand, a conditioned PLOXY point is planned by the user to provide points of synchronisation between concurrent processes.

At a nested PLOXY point

Due to the unpredictable outcome of a nested monitor call, should a process attempt a blocked nested monitor call, CLANG ensures that the variables of the monitors whose exclusivities are released, will contain the same values when those exclusivities are regained, regardless of how many other processes may gain access to those monitors in the interim. This guaranteeing of invariance is implicit.

CLANG is seen as a teaching language, and as it is sometimes desirable to demonstrate the effects of not ensuring the invariance of monitor variables, a compiler directive has been provided to override this invariance (cf. Appendix A: The User Manual).

At a conditioned PLOXY point

Any process performing either a qwait or the qpwait(PRIORITY) operation on a condition variable is suspended and releases exclusivity to any monitors that it might currently be holding. Similarly any process performing a qsignal operation on a condition variable is temporarily suspended should there be some other process waiting for that signal. Although the signalling process might not transfer all its exclusivities to the signalled process, provision still has to be made for invariance in those that are.

As these conditioned PLOXY points are planned to provide synchronisation between concurrent processes, CLANG introduces two explicit standard procedures SAVE(parameters) and RESTORE. The standard procedure SAVE(parameters) allows the programmer explicitly to state (as the parameters to the SAVE) which variables of the monitor he or she wishes to make invariant (if any).

Note: This choice is limited to the monitor in which the conditioned PLOXY point occurs; any variables in the other monitors that the suspended process holds will be saved implicitly as per the nested PLOXY point (see above).

The RESTORE instruction re-establishes the monitor's variables to their expected values.

Due to the asynchronous nature in which the concurrent processes are executed in relation to each other, it is possible to execute a program without ensuring any invariance of monitor variables and still achieve the desired results.

However, without monitor variable invariance it is not possible to guarantee that the next time the program is run the results will again be achieved.

Additional advantages / disadvantages

Monitor variables that are declared as starred identifiers may be inspected, but their values may not be altered, from outside the monitor in which they were declared. Thus several processes may be inspecting a monitor variable while another may be "inside" the monitor altering its value. The value that the inspecting processes obtain could be either the value before or after the alteration.

A program's global variables may be inspected, but their values may not be altered, from within a monitor.

There are no safeguards to prevent concurrent processes updating the global variables "simultaneously". The onus is on the programmer to ensure that this does not happen. In CLANG there is no distinct PROCESS type, processes are just procedures called from within the Cobegin..Coend construct, so it is impossible for the compiler to ascertain at compile time whether a procedure is accessing a global variable from one of a set of concurrent processes or not.

Implementing monitors in CLANG - illustrated details

In the implementation developed to date, the language CLANG is compiled into intermediate P-codes by a compiler written in Pascal. (This is UCSD Pascal, with use made of as few "extensions" as possible.) This P-code is then interpreted by a procedure which forms an integral part of the compiler program. (The P-code set is based on that given in [Wir76].) Extensive use is made of Pascal's POINTER and SET facilities in implementing the various queues associated with monitors and condition variables.

The language CLANG has come a long way since its original inception as a program for teaching compiler construction. Several earlier versions of CLANG exist, such as CLANG6 by Terry [Ter83], and in these it is possible to construct monitors explicitly by means of semaphores, similar to the way shown by Ben-Ari [Ben82], or by the means shown by Boddy [Bod83] and [Bod84]. However these earlier versions do not attempt to tackle any of the problems associated with the monitor concept (cf. section 4.1) and are best suited for teaching the concepts of "simple" concurrency.

This section will be concerned with the latest version of CLANG, (CLANG 21.2C), and gives a description, in conjunction with flow diagrams (which should be studied with the accompanying notes) of the implementation of:

(1) monitors,

(2) condition variables, and

(3) invariance of monitor variables.

(1) Monitors

The whole crux of the implementation of monitors is the assigning of a unique number, by the parser, to each monitor as it is declared. This enables easy identification at run time as to which monitor is being referenced.

During run time there is a set, AVAILABLEMONITORS, which holds the unique numbers of those monitors to which no process currently has exclusivity.

It is necessary to establish exactly when a process requests exclusivity to a monitor and when it is releasing it.

Exclusivity is only required when a process wishes to call a starred procedure or function of a monitor. This is easily detected during compile time:

ie. monitorname.subprogramname

and thus at the P-code level a call to a starred procedure or function is preceded by a P-code which requests exclusivity to a particular monitor (identified by means of its unique number) before the calling process can continue.

Similarly a process releases exclusivity to a monitor on exit from a starred procedure or function, and so the return instruction is preceded by a P-code which will inform the other processes of the particular exclusivity being released.

A process on being suspended is also compelled to release the exclusivities it is currently holding. The possibility of suspension due a blocked nested monitor call cannot be established during compile time and so is dealt with implicitly at run time. Only if a process is suspended will the exclusivities be released. How this is achieved will be detailed shortly.

Each active process in CLANG has its own entry in a (circularly linked) Process Descriptor Table (PTAB). Contained within this process descriptor table are a number of fields which hold information relevant for the execution of each process.

Three new fields were introduced into PTAB for use in connection with monitors.

```
PTAB : array [PTYPE] of
         record
           ...
           EXCLUSSET, HELDSET: set of 1..MONMAX;
           NOOFELEMENTS: Ø..MONMAX;
           ...
         end;   (*PTAB*)
```

EXCLUSSET - holds the unique numbers of those monitors whose exclusivity the process has yet to relinquish.

HELDSET - is used to establish whether a process, on being reactivated after being suspended, has in fact reacquired all its necessary exclusivities so as to be allowed to continue execution.

NOOFELEMENTS - holds the number of exclusivities a process released on being suspended.

The implicit queue for each monitor is achieved by an array, MONITORQUE, of dynamic structures, indexed by the unique numbers of the monitors.

The delicate and involved nature of the proceedings when a process requests or releases exclusivity require these operations to be indivisible. This is achieved by making the request for exclusivity and the release of exclusivity single P-codes. (A process switch in CLANG can only occur after a single P-code has been completely interpreted.)

Requesting exclusivity

The diagramatic representation of the actions performed when a process requests exclusivity are shown in figure I.

Notes relating to figure I

(1) A process is able to determine whether another process already has exclusivity to the requested monitor by checking whether the monitor's unique number is in AVAILABLEMONITORS or not.

(2) An examination of the contents of the EXCLUSSET field for a process will establish whether or not a process has exclusivity to other monitors.

(3) The process is queued on the implicit monitor queue with a priority worked out by means of:

(the maximum number of monitors allowed per program) + 1
  - (the number of exclusivities held by the process)

ie. MONMAX + 1 - NOOFELEMENTS

where MONMAX = 15 in the implementation of CLANG under discussion.

The reason for this choice of priority on the implicit monitor queue as opposed to a simple first-come-first-served

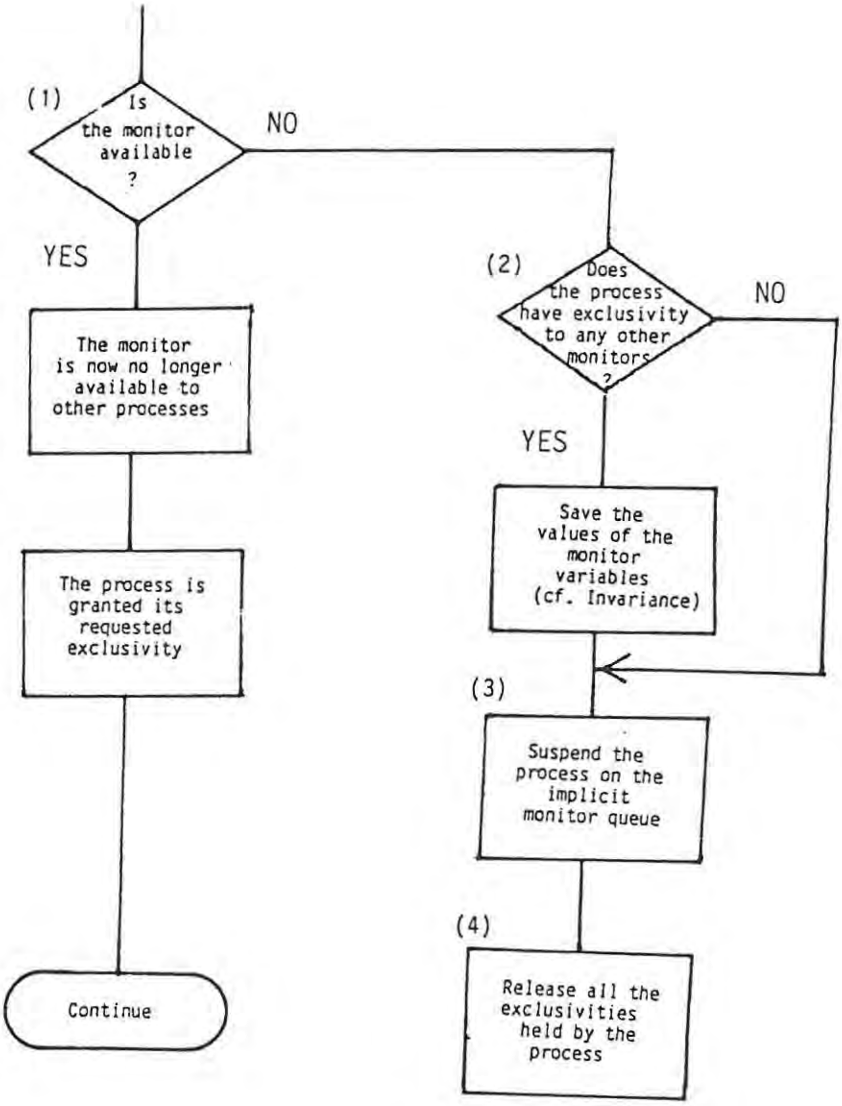


figure I: Requesting exclusivity

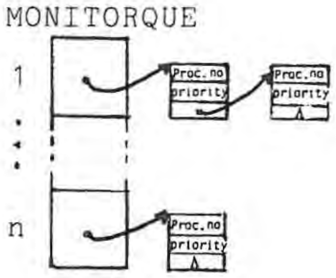
strategy was based on the logic that the more exclusivities a process has, the more will be released for other processes to access, when that process finishes execution.

This priority also has important consequences for minimising storage requirements necessary for ensuring monitor variable invariance, as will be shown later.

This priority strategy does mean that those processes with few or no exclusivities released will take slightly longer before they are granted the requested exclusivity, but the possibility of indefinite overtaking is prevented as will be shown in the example given below.

Processes with the same priority will be queued on a first-come-first-served basis.

<u>Aside:</u> The information that needs to be stored on the monitor queue is: the process number (<u>ie.</u> the suspended process' entry into the process descriptor table), the process' priority and a pointer to the next process on the queue (if any).
This can be represented graphically as:



(4) The actions taken when releasing exclusivities due to a process becoming suspended are the same as when a process releases exclusion by leaving a monitor, save that for a process being suspended the actions must be performed for every exclusivity that the process holds, whereas for a process which exits a monitor, the actions are performed for exclusivity to that monitor alone.

Releasing exclusivity

When exclusivity to a monitor is released, the exclusivity is not simply added to the set of available monitors, but rather, a check is first performed as to whether any process might already be queued waiting for the exclusivity. If there is a process waiting, then that process is granted the exclusivity and then it must endeavour to regain all its released exclusivities so that it may continue execution.

The actions undertaken when execlusivity to a monitor is released can be viewed diagramatically in figure II.

Notes relating to figure II

(1) A process is added to the "polite" queue (GETFIRST) as a result of a qsignal operation on a condition variable. This will be shown in the section on implementing condition variables.

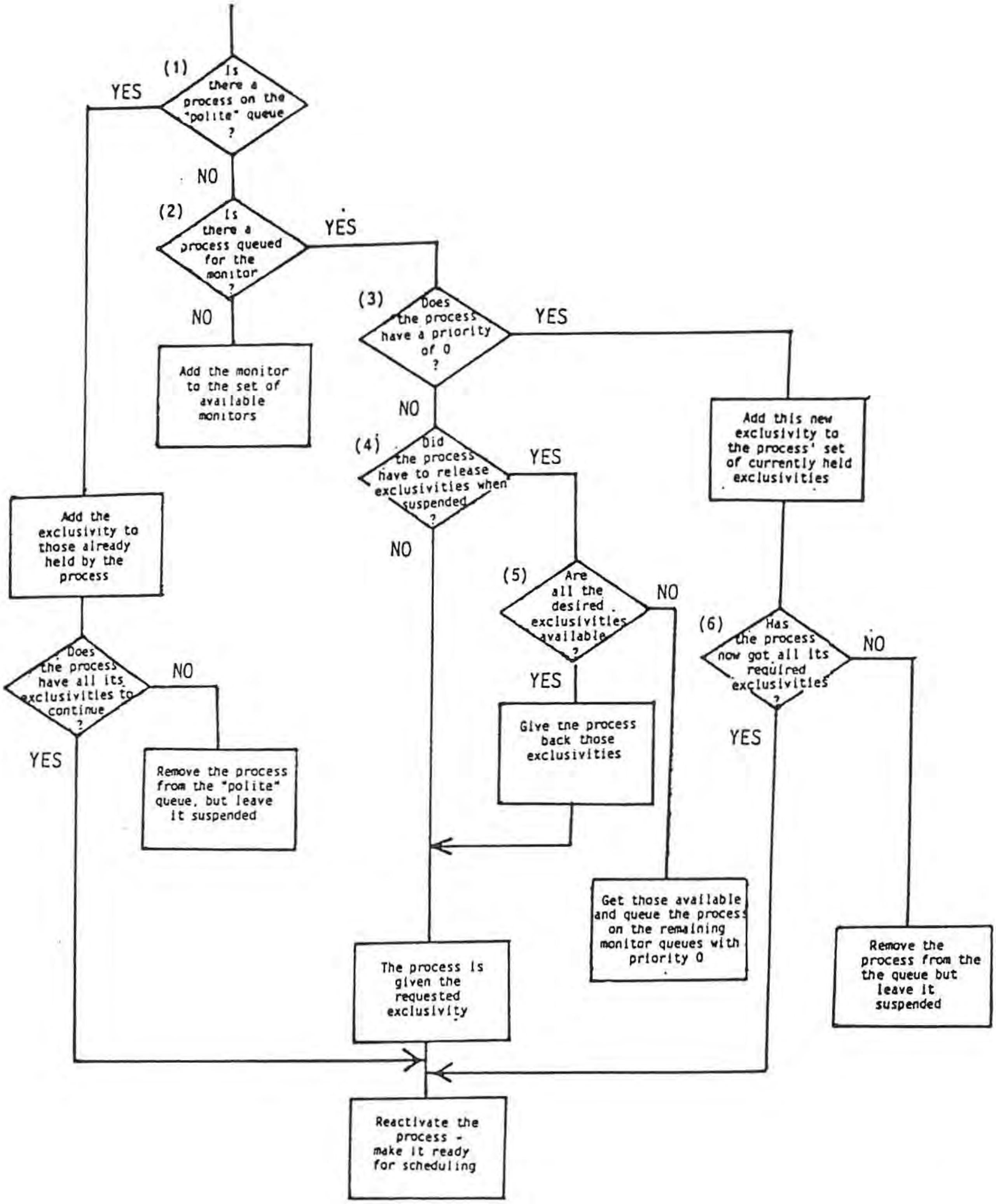


figure II: Releasing exclusivity

(2) The check is performed by examining the monitor queue indexed by the unique number of the monitor whose exclusivity is released. If this queue is empty then there is no process waiting to gain that exclusivity.

(3) The priority of Ø for a process at the head of the monitor queue needs some explaining.

The formula for assigning priorities to a process on being suspended (cf. note 3 relating to figure I), allows processes to be assigned priorities in the range 1 to 16 - a low priority value indicating a high priority status. A priority of Ø is the highest priority a process can have on the monitor queue. This priority is only assigned to a process when, having already obtained the exclusivity for which it was originally suspended, it may still not proceed but must be queued (with this priority of Ø) waiting to reacquire all its necessary exclusivities, that other processes currently hold.

The top priority thus ensures that such a process is delayed for a short a time as possible.

(4) This can easily be seen by examining the contents of EXCLUSSET. If EXCLUSSET is empty then no exclusivities were released when that process was suspended.

(5) Once a process is reactivated it must reacquire all the exclusivities it released on being suspended. An examination of EXCLUSSET and AVAILABLEMONITORS will reveal whether all the desired exclusivities are available or not. If they are then the process can reacquire them and be restored to the ready state. For all those exclusivities that are unavailable the process is suspended on the relevant monitor queue with a priority of Ø.

(6) Every time a reactivated, but still delayed, process reacquires one of its necessary exclusivities, this exclusivity is added to the HELDSET field of that process. When HELDSET = EXCLUSSET that process has reacquired all its necessary exclusivities and need be delayed no longer.

Additional Notes

If more than one process is suspended on a monitor queue with a priority of Ø then these processes will be queued on a first-in-first-out basis.

The release of exclusivity is an indivisible operation and cannot be interrupted by other processes wishing to release exclusivities or suspend themselves on monitor queues.

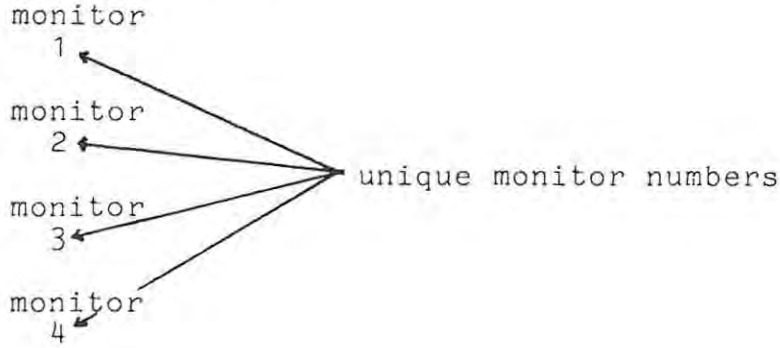
In CLANG it is possible to call a starred procedure or function recursively or from another procedure or function declared within the same monitor. These subsequent calls are performed without requesting exclusivity again (which would result in deadlock) and without releasing exclusivity to the monitor until returning to the point from where the original monitor procedure or function was called. (This is achieved by the SKIP field in a process' descriptor table.)

<u>Detailed example:</u>

This worked example is designed to show the workings of the various queues relating to monitor exclusivity. It is hoped that by careful study of this example in conjunction with the flow diagrams (figures I and II) the techniques used in implementing monitors in CLANG will become clear to the reader, thus making the understanding of actual Pascal code, supplied in Appendix B, that much easier.

Consider the case of four monitors and five processes declared in a program.

Initially the set up is:



Note: The hierarchical nature of monitor declarations restricts which monitors may be called from which other monitors.

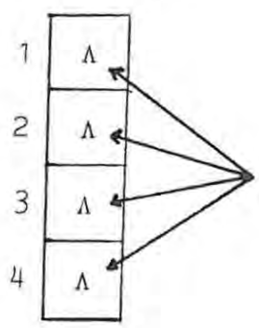
eg. It is possible to call a procedure / function in monitor 1 , 2 and 3 from within monitor 4, but no procedure / function in any other monitor may be called from within monitor 1.

Process descriptor table:

Processes	A	B	C	D	E
EXCLUSSET	φ	φ	φ	φ	φ
HELDSET	φ	φ	φ	φ	φ
NOOFELEMENTS	Ø	Ø	Ø	Ø	Ø

Note: The characters for the processes are used purely for ease of identification.

MONITORQUE



queues (indexed by the unique monitor numbers) are initially empty.

AVAILABLEMONITORS = [1, 2, 3, 4]

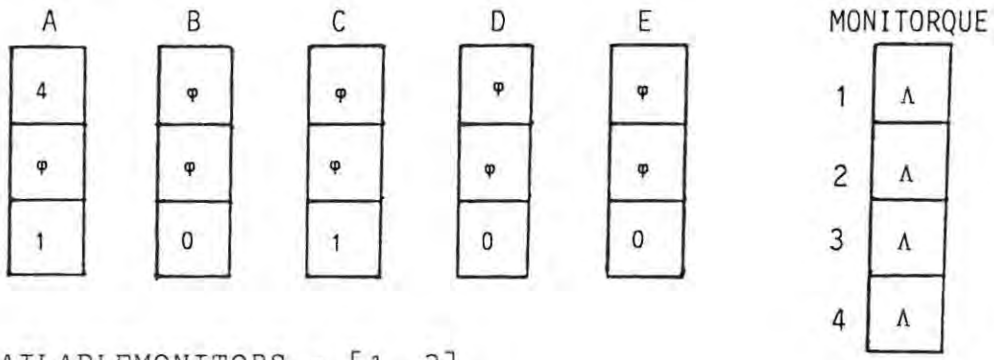
Processes ready for scheduling = (A, B, C, D, E)

There follows a trace of possible events and their effect once concurrency has commenced.

Process A asks for and receives exclusivity to monitor 4.

Process C asks for and receives exclusivity to monitor 2.

Picture so far:



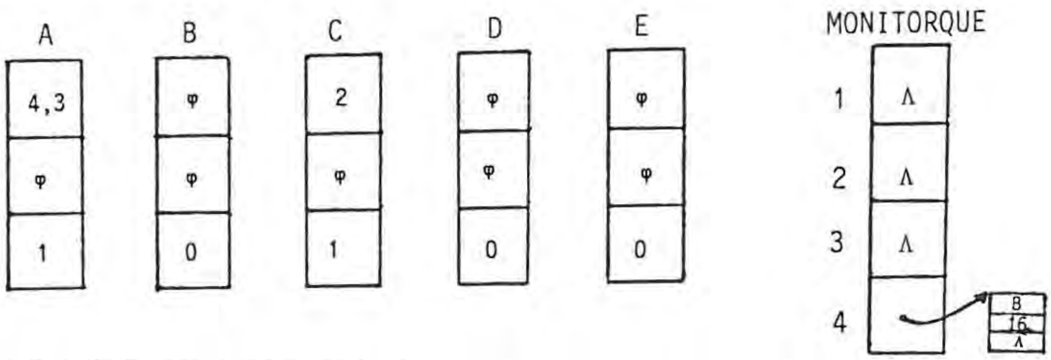
AVAILABLEMONITORS = [1, 3]

Processes ready for scheduling = (A, B, C, D, E)

Process A from within monitor 4 asks for and receives exclusivity to monitor 3.

Process B asks for exclusivity to monitor 4 and is therefore queued with priority = (15 + 1 - Ø) =16.

Picture so far:



AVAILABLEMONITORS = [1]

Processes ready for scheduling = (A, C, D, E)

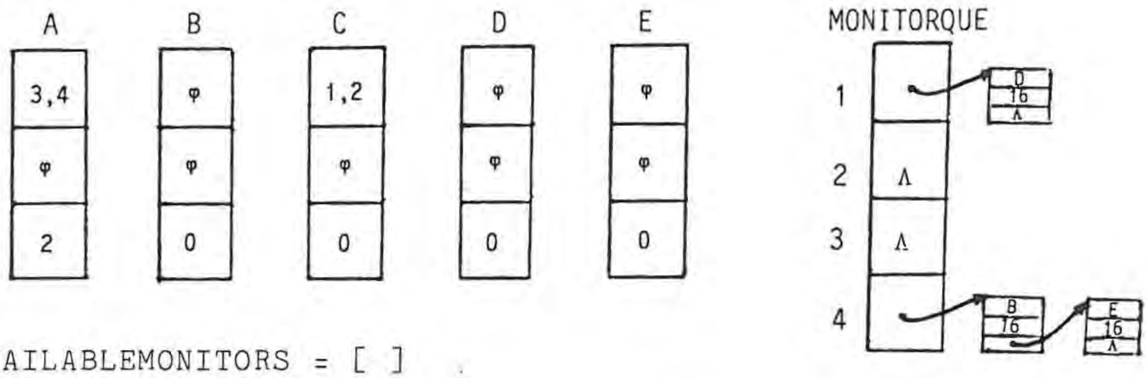
Note: Process B has yet to gain an exclusivity.

Process E requests for exclusivity to monitor 4 and is therefore suspended with priority = 16.

Process C requests and receives exclusivity to monitor 1.

Process D requests exclusivity to monitor 1 and is therefore queued with priority = 16.

Picture so far:



AVAILABLEMONITORS = [ ]

Processes ready for scheduling = (A, C)

Notes:

(a) Process E is queued behind process B.

(b) Process D is queued as process C requested the exclusivity to monitor 1 first.

Process A now requests exclusivity to monitor 1. The following events occur:

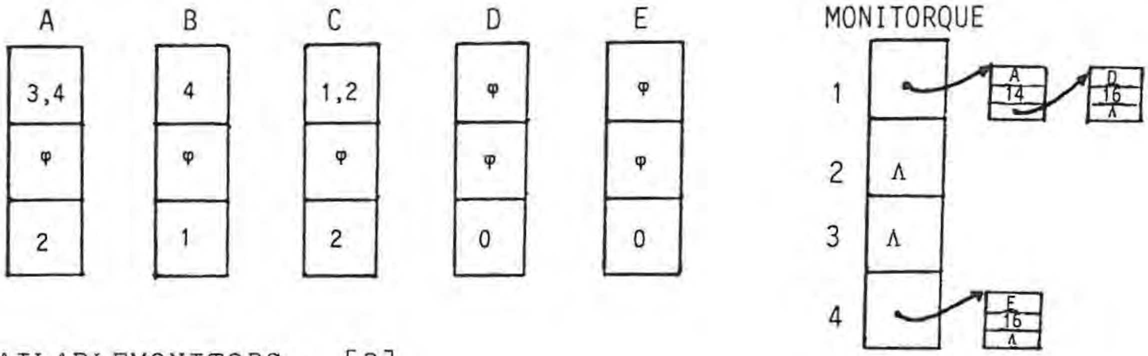
(1) Monitor 1 is not available so therefore process A is queued for monitor 1 with priority = 16-2= 14.

(2) Because process A is suspended it must release the exclusivity to the monitors that it is already holding, namely to monitors 3 and 4.

(3) As monitor 4 has now become available, process B is removed from the head of the queue, granted the exclusivity and restored to the ready state.

Note: Events (1) to (3) all take place during the execution of one P-code (the request by process A for exclusivity to monitor 1).

Picture so far:



AVAILABLEMONITORS = [3]

Processes ready for scheduling = (B, C)

Notes:

(a) Process A is queued in front of process D as process A has the higher priority.

(b) Process E is now the first element on the queue for monitor 4 as process B has been removed.

(c) The exclusivities released by process A are remembered by that process.

Process C now finishes with monitor 1 and releases the exclusivity with the following consequences:

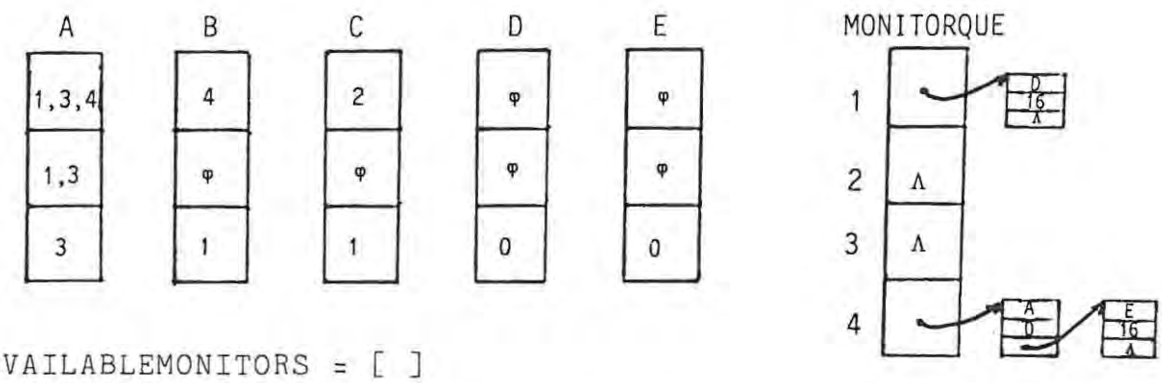
(1) Monitor 1 is removed from process C's EXCLUSSET.

(2) The queue for monitor 1 is examined - process A is on top of the queue and so process A is given the exclusivity to monitor 1.

(3) Process A needs exclusivity to monitors 3 and 4 before it can continue - are these available ? Monitor 3 is available so process A reacquires the exclusivity to monitor 3.
The exclusivity to monitor 4 is not available so process A is queued for this with priority Ø.

Note: Events (1) to (3) all occur as the result of one P-code (the release of the exclusivity to monitor 1 by process C).

Picture so far:



AVAILABLEMONITORS = [ ]

Processes ready for scheduling = (B, C)

Notes:

(a) Process A has the highest priority on the queue for the exclusivity to monitor 4 and is therefore queued in front of process E.

(b) Process A still has to get exclusivity to monitor 4 before it can continue.

(c) Processes E and D are still unable to continue.

Process B now leaves monitor 4 thus releasing the exclusivity. The following occurs:

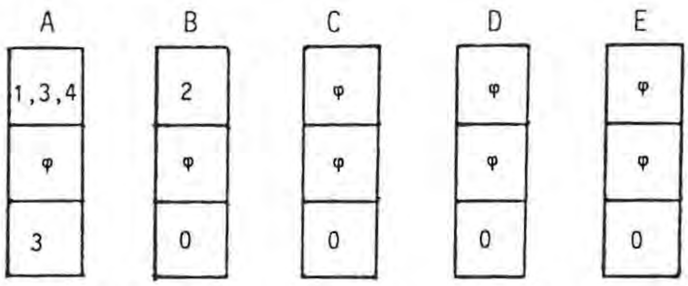
(1) The queue for monitor 4 is examined - process A is at the front of the queue with a priority of Ø.

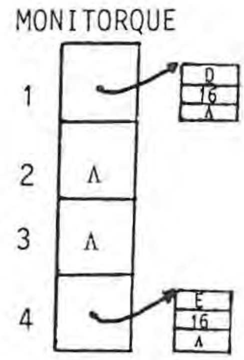
(2) The exclusivity to monitor 4 is added to the set of exclusivities already being held by process A (HELDSET) and process A is removed from the queue.

(3) A check is now carried out to see if process A now has all its required exclusivities. Indeed it does, so process A can be readied for scheduling.

Note: Again events (1) to (3) all occur as the result of the execution of one P-code (process B releasing exclusivity to monitor 4).

Picture so far:





AVAILABLEMONITORS = [ ]

Processes ready for scheduling = (A, B, C)

Notes:

(a) Process A now has all its desired exclusivities so its HELDSET is set back to NULL.

(b) Process E will be the next process to be granted exclusivity to monitor 4.

The weakness of the priority queueing strategy can be seen from the fact that processes D and E have yet to gain any exclusivities, but as will be discussed in the section on implementing monitor variable invariance, this consequence is far outweighed by the amount of storage that would be wasted if the priority strategy was not used.

(2) Condition variables

A unique number assigned during compile time is used to identify the individual condition variables at run time.

These unique numbers are assigned to the condition variables as they are declared and as each operation on a condition variable has to be prefixed:

ie. conditionvariablename.operation

the unique number can be incorporated into the P-code for that operation.

Associated with each condition variable is a queue on which processes can be suspended. An array (CONDVARQUE) of dynamic structures is used to implement these queues, indexed by the condition variable's unique number.

Of the five operations allowable on condition variables, qlength and queue are functions:

Qlength returns the number of processes suspended on a condition variable queue. This is easily achieved by a simple count of the number of processes on the queue.

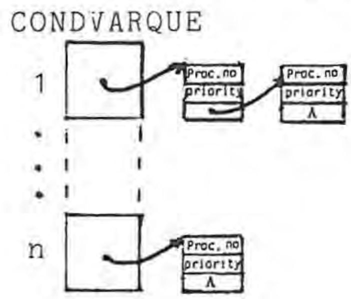
Queue returns the value 0 or 1 depending on whether the queue associated with the condition variable involved is empty or not.

The priority value for the qwait or qpwait(PRIORITY) operations is to be found on the stack frame of the process performing the operation, while the unique number of the condition variable is part of the P-code for that operation.

The process performing the operation is suspended on the relevant queue behind any process with an equal or higher priority. The suspended process then releases all its held exclusivities (as shown in figure II).

The information that needs to be stored on a condition variable queue includes: the process' index into the process descriptor table; the priority of the process on the queue; and a pointer to the next process on the queue (if any).

This can be viewed graphically as:



The activities involved when a process performs a qsignal operation on a condition variable are a little more complicated.

In order to implement this operation another queue, the so-called "polite" queue (GETFIRST) was introduced.

(<u>Aside:</u> It would have been possible to use the existing queue MONITORQUE and simply implement "politeness" by means of a high priority (higher than Ø, say -1). However this was rejected in favour of a separate queue, GETFIRST, so as clearly to distinguish the "polite" queue from the exclusivity queue.)

As can be seen in figure II, a process on the GETFIRST queue has the highest priority to gain the relevant monitor exclusivity as that queue is checked before MONITORQUE.

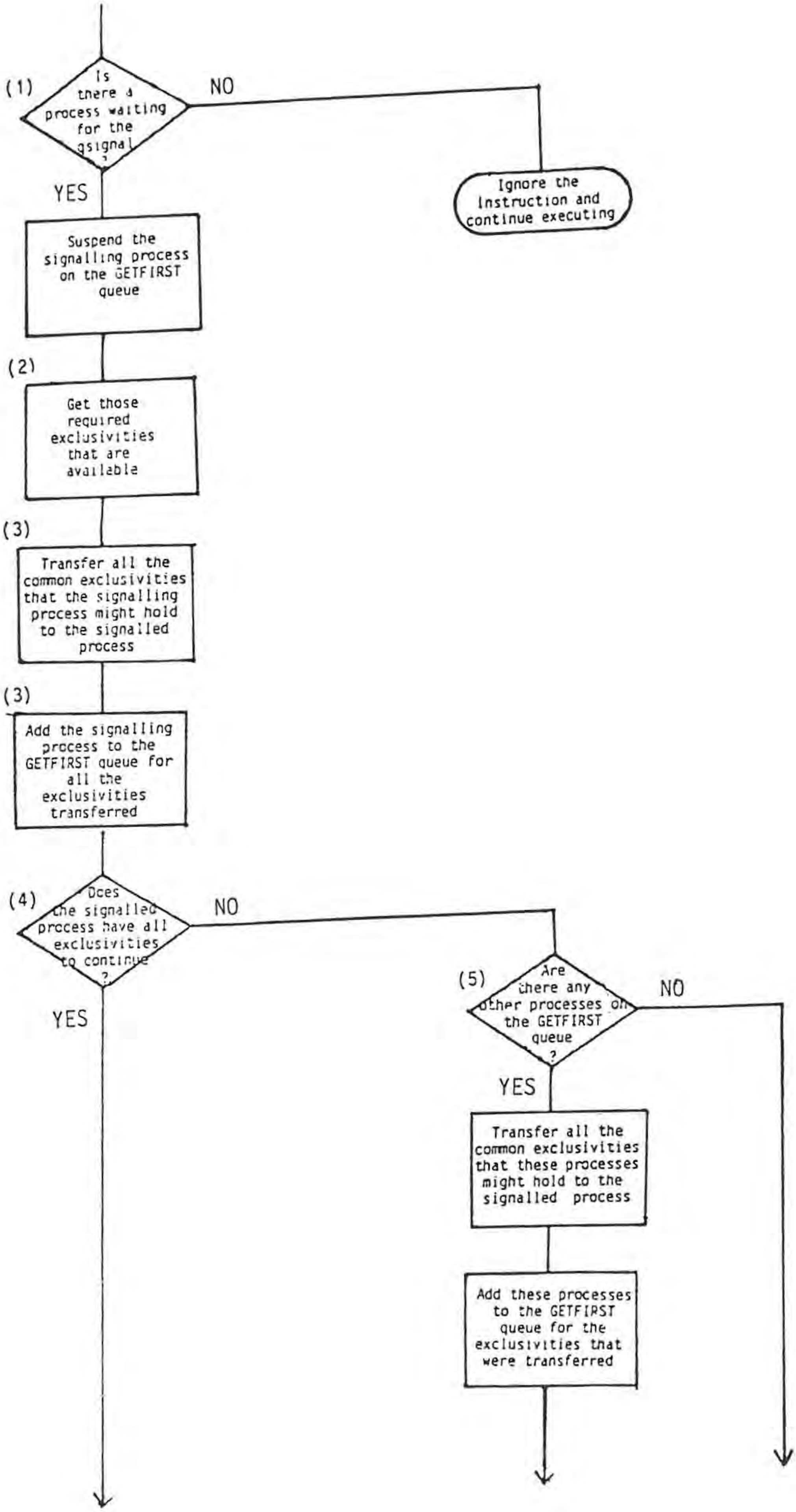
GETFIRST is also indexed by the unique number of the monitor whose exclusivity is being dealt with.

The actions involved when a process performs a qsignal operation on a condition variable can be seen diagrammatically in figure III.

(The process performing the qsignal operation is termed the <u>signalling</u> process, while the process at the head of the condition variable queue that is reactivated by the qsignal is termed the <u>signalled</u> process.)

<u>Notes relating to figure III</u>

(1) If the queue, indexed by the unique number of the condition variable concerned, is empty, then the qsignal operation has no effect.


(1)
Is there a process waiting for the signal ?
NO
Ignore the instruction and continue executing
YES
Suspend the signalling process on the GETFIRST queue
(2)
Get those required exclusivities that are available
(3)
Transfer all the common exclusivities that the signalling process might hold to the signalled process
(3)
Add the signalling process to the GETFIRST queue for all the exclusivities transferred
(4)
Does the signalled process have all exclusivities to continue ?
NO
YES
(5)
Are there any other processes on the GETFIRST queue ?
NO
YES
Transfer all the common exclusivities that these processes might hold to the signalled process
Add these processes to the GETFIRST queue for the exclusivities that were transferred

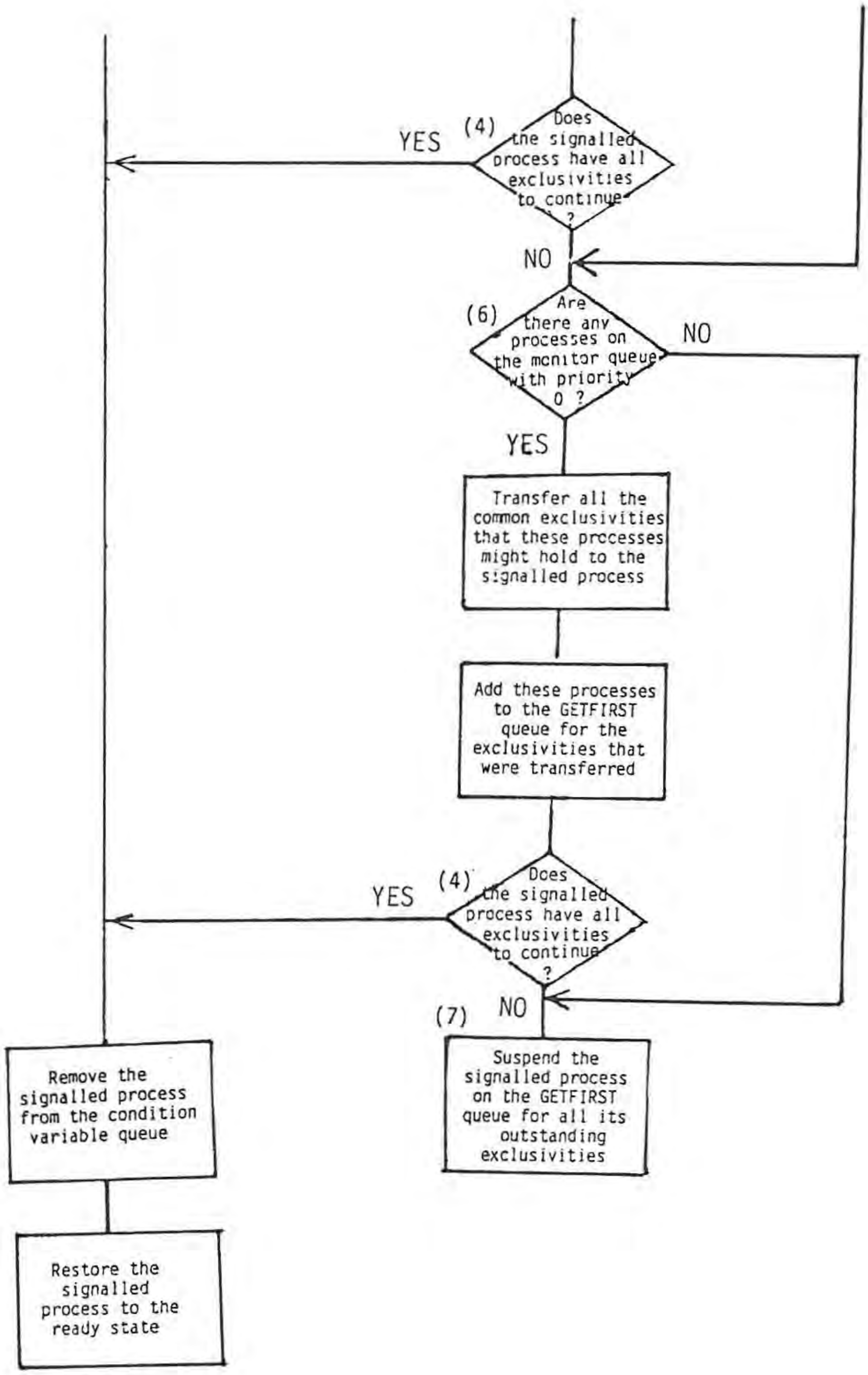


figure III: <u>The qsignal operation</u>

A signalled process may not proceed until it has reacquired all the exclusivities it released on suspension. In order to prevent deadlock the signalled process must be given those exclusivities that it needs, from the processes that are suspended but holding them. These processes must either be temporarily suspended (eg. the signalling process), or be processes that have been reactivated but are as yet unable to continue as all their necessary exclusivities (including the monitor in which the qsignal operation is taking place) are unavailable (cf. note 3 relating to figure II). Processes that give up held exclusivities must have first option for their return and so are added to the GETFIRST queue for the relevant exclusivity. (Obviously no two processes can be holding the same exclusivity.)

(2) The contents of AVAILABLEMONITORS is examined to see if any exclusivities required by the signalled process are available.

(3) The process that just performed the signal is checked first for any exclusivities it might have in common with those required by the signalled process. (One of these will be the exclusivity to the monitor in which the qsignal and the qwait, or qpwait(PRIORITY), operations took place.)

The signalling process is then added to the GETFIRST queues indexed by those exclusivities it transferred.

(4) A process can check whether it has now holds all its required exclusivities by comparing the contents of HELDSET with the contents of EXCLUSSET.

(5) If this "pilfering" of the exclusivities from the signalling process does not yield all the necessary exclusivities, the signalled process examines the HELDSETs of other processes that might be suspended on the same GETFIRST queue. These could include signalled processes which, without first releasing exclusivity to that monitor, have themselves performed a qsignal operation, and/or processes which have been suspended on this queue as the result of transferring the exclusivity concerned to a signalled process.

(6) If both (3) and (4) are still not enough, the signalled process examines the processes that have been reactivated elsewhere, but need to reacquire, at least, the exclusivity of the monitor concerned in order to continue. (They will be suspended on MONITORQUE with a priority of Ø.)

(7) Should the signalled process still not have reacquired all its necessary exclusivities then it is added to the GETFIRST queue for those exclusivities still outstanding and must remain delayed until such time as they become available.

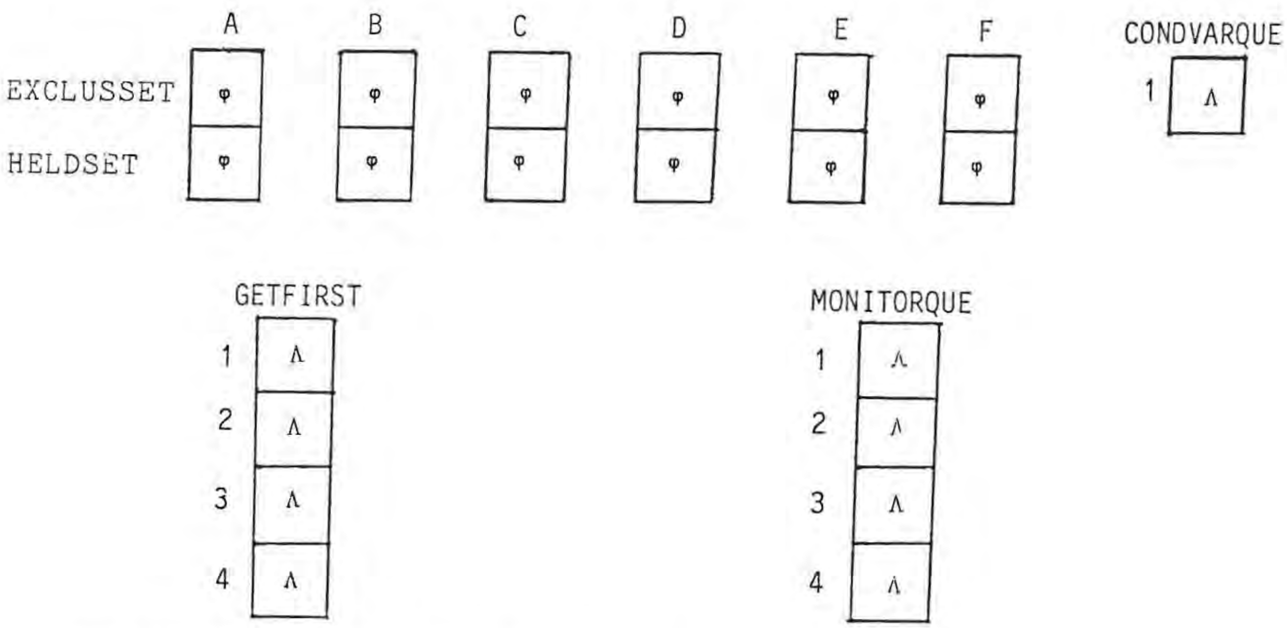
Detailed example

It is hoped that the careful study of the following example in conjunction with flow diagram III will provide the reader with some insight as to how the various queues relating to operations on condition variables are manipulated.

In this example there are four monitors and six processes. A condition variable, C1, has been declared in the second monitor.

There follows a trace of possible events and their consequences, from the launching of concurrency.

Picture so far



AVAILABLEMONITORS = [1, 2, 3, 4]

Processes ready for scheduling = (A, B, C, D, E, F)

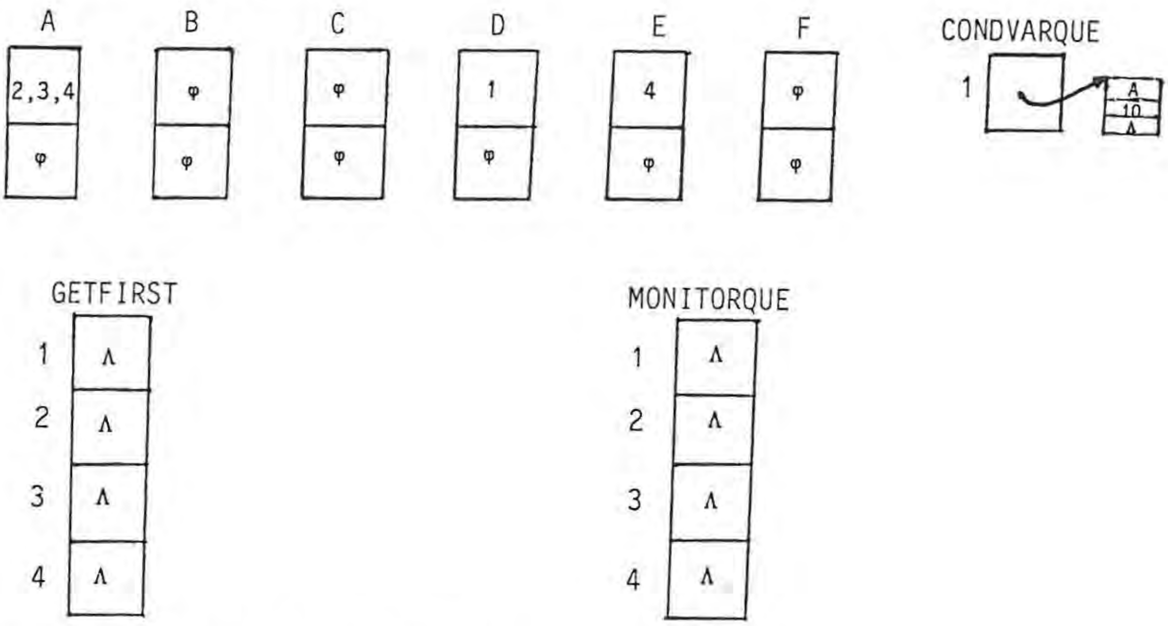
Process A performs succesful nested monitor calls to monitors 4, 3 and 2.

Process D performs a succesful call to monitor 1.

Process E requests exclusivity to monitor 4 and is suspended.

Process A now performs a qwait operation on the condition variable C1 in monitor 2 and is therefore suspended and as a result of this process E is granted the exclusivity to monitor 4 and reactivated; the exclusivities to monitors 2 and 3 are added to AVAILABLEMONITORS.

<u>Picture so far</u>



AVAILABLEMONITORS = [2, 3]

Processes ready for scheduling = (B, C, D, E, F)

Process E performs successful nested monitor calls to monitors 3 and then 2, but its nested monitor call to monitor 1 is blocked as process D currently has the exclusivity. Process E is suspended waiting for exclusivity to monitor 1 and releases exclusivity to monitors 2, 3 and 4.

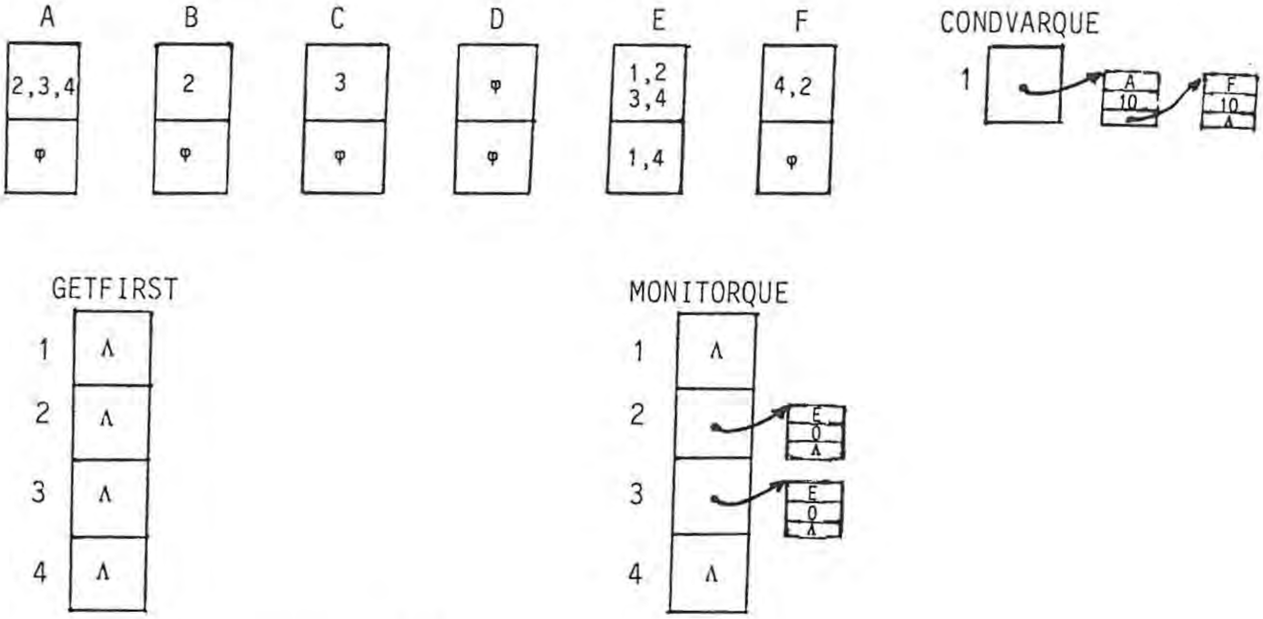
Process F executes nested monitor calls to monitors 4 and 2 and then performs a qwait operation on the condition variable C1 in monitor 2 and is suspended and releases exclusivity to monitors 4 and 2.

Process B requests and is granted exclusivity to (the now available) monitor 2.

Process C performs a successful monitor call to (the now available) monitor 3.

Process D leaves monitor 1 thus releasing exclusivity, which is then granted to process E. Process E is able to acquire exclusivity to monitor 4, which is available, but must be queued, with priority Ø, for monitors 2 and 3.

Picture so far



AVAILABLEMONITORS = [ ]

Processes ready for scheduling = (B, C, D)

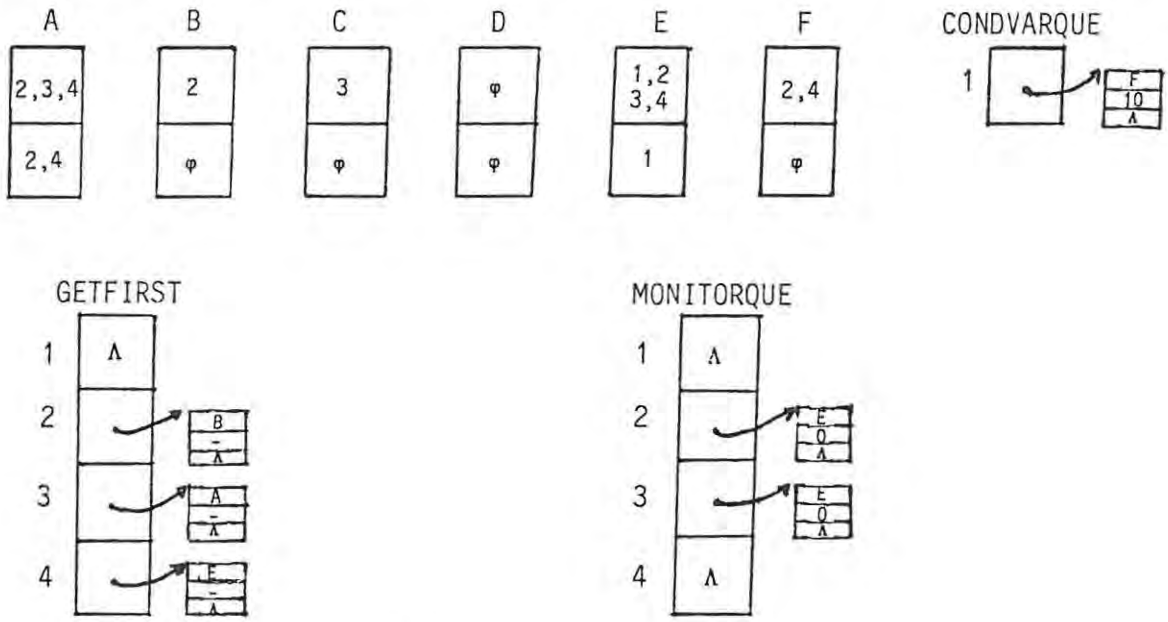
Process B executes a qsignal operation on the condition variable C1 in monitor 2. Process A is at the head of the queue for C1 so process A is reactivated and process B is suspended on the GETFIRST queue for monitor 2. Process A must now reacquire all the exclusivities it released, (remembered in EXCLUSSET), in order to continue.

The exclusivity to monitor 2 is transferred from process B (when the qsignal is performed). Process B has no further common exclusivities and there are no other processes suspended on the GETFIRST queue for monitor 2, so MONITORQUE queue for monitor 2 is checked for any processes with a priority of Ø.

Processes E is suspended on this queue with priority Ø and so the exclusivity already held by process E, (namely that to monitor 4), is transferred to process A and process E is suspended on the GETFIRST queue for exclusivity to monitor 4. The exclusivity to monitor 3 is unavailable, (process C is busy with it), so process A is suspended on the GETFIRST queue for exclusivity to monitor 3.

The execution of one P-code (the qsignal on C1) has changed the queues as such:

Picture so far



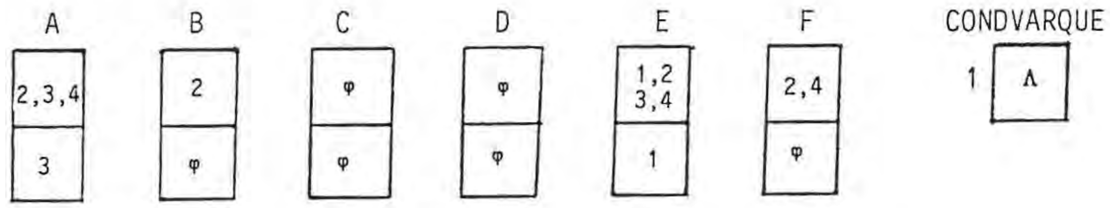
AVAILABLEMONITORS = [ ]

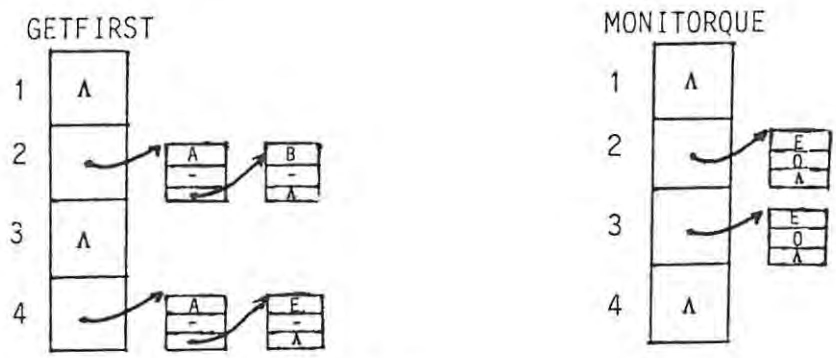
Processes ready for scheduling = (C, D)

Process C finishes with monitor 3, releasing the exclusivity which is given to process A. Process A is removed from the GETFIRST queue for monitor 3 and as it now has all its necessary exclusivities, is ready for scheduling.

Process A now executes a qsignal operation on the condition variable C1 in monitor 2. Process F is at the head of the queue and is thus reactivated. Process A is temporarily suspended on the GETFIRST queue for monitor 2, in front of process B, and the common exclusivities to monitors 2 and 4, held by process A and needed by process F, are transferred to process F. Process A is thus also suspended on the GETFIRST queues for exclusivity to monitor 4, in front of process E. Process F now has all its required exclusivities and is thus available for scheduling.

Picture so far





AVAILABLEMONITORS = [ ]

Processes ready for scheduling = (C, D, F)

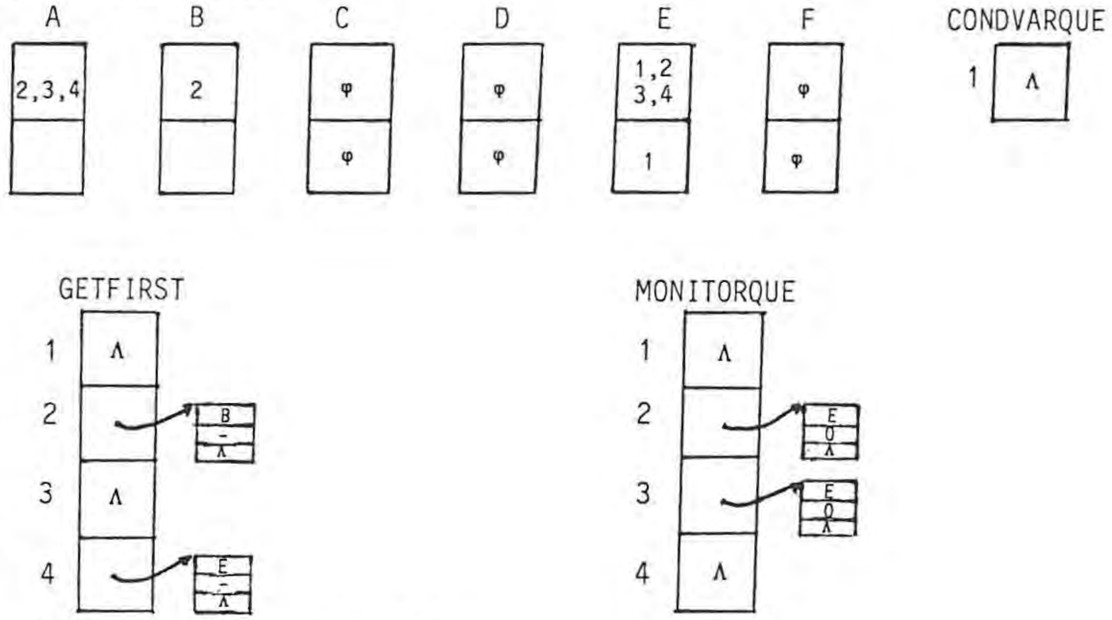
Note:

(a) When a process is ready for scheduling its HELDSET is set to NULL. Only while a process is reactivated, but still delayed, will its HELDSET contain the set of exclusivities which it is currently holding.

Process F leaves monitor 2, releasing the exclusivity which is given to process A (as process A is at the head of the GETFIRST queue for monitor 2). Process A is removed from the GETFIRST queue for monitor 2, but still needs the exclusivity to monitor 4 before it may continue.

Process F now exits monitor 4, releasing the exclusivity which is then given to process A. Process A is removed from the GETFIRST queue for monitor 4 and as it now has all its necessary exclusivities, is ready for scheduling.

Picture so far

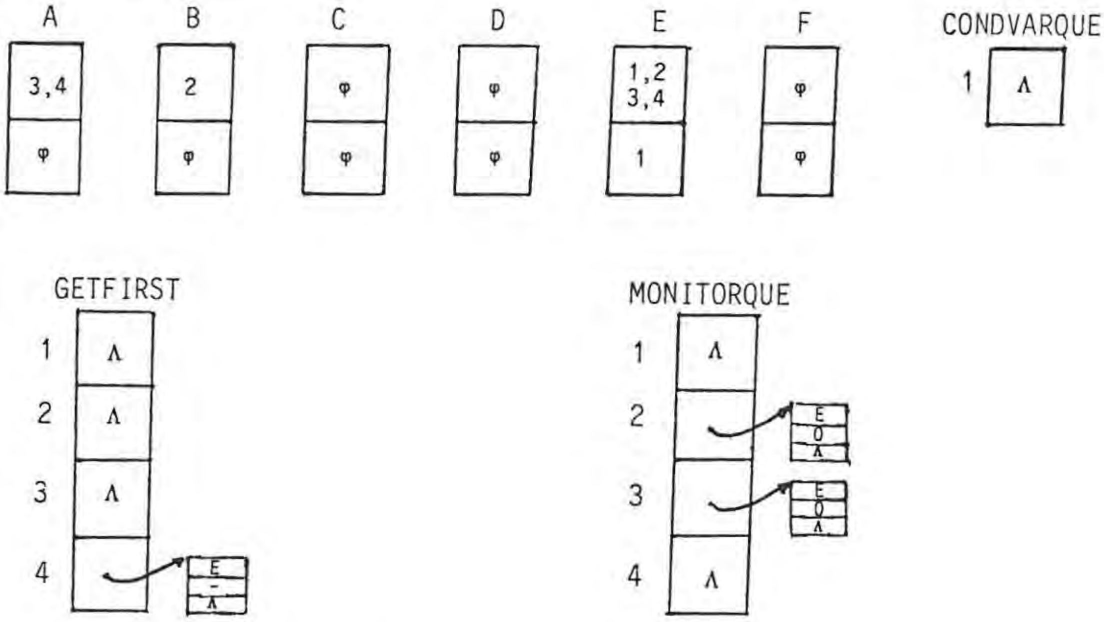


AVAILABLEMONITORS = [ ]

Processes ready for scheduling = (A, C, D, F)

Process A leaves monitor 2 and the exclusivity is given to process B which then has all its necessary exclusivities and may therefore also be readied for scheduling.

Picture so far

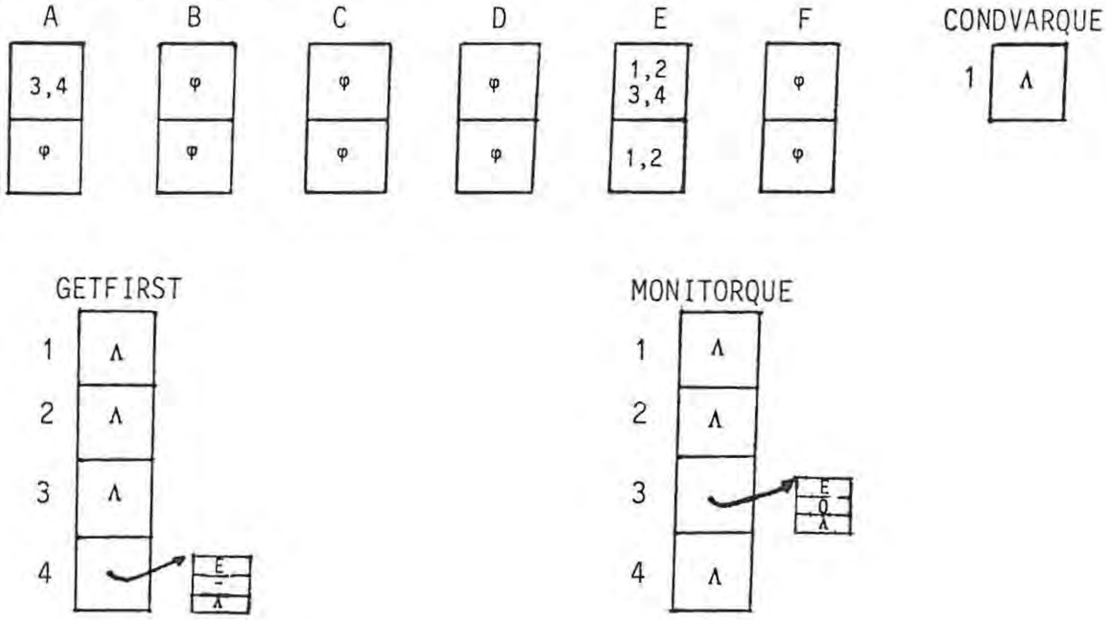


AVAILABLEMONITORS = [ ]

Processes ready for scheduling = (A, B, C, D, F)

Process B leaves monitor 2 and the exclusivity is given to process E as process E is sitting on MONITORQUE for monitor 2 with a priority of Ø and there are no other processes on the GETFIRST queue for monitor 2. Process E still does not have all its required exclusivities and thus remains delayed.

Picture so far



AVAILABLEMONITORS = [ ]

Processes ready for scheduling = (A, B, C, D, F)

Process A finishes with monitor 3 and the exclusivity is given to process E which still needs exclusivity to monitor 4 before it can be readied for scheduling.

Process A finishes with monitor 4 and the exclusivity is given to process E which now, finally, has all its desired exclusivities and can be readied for scheduling.

(3) Invariance of monitor variables

Invariance of monitor variables means that the monitor variables must have the same values when a process reacquires it exclusivities as when it was forced to release them on suspension.

Concern for this invariance is only necessary when a process is suspended either by executing a nested monitor call which is blocked, or at a conditioned PLOXY point.

To facilitate this "backing up" of the values of monitor variables a new field, VARSTACK, was introduced into the process descriptor table (PTAB). This field contains a pointer to a dynamically created list on which the values and addresses of the monitor variables can be saved when the need arises.

Should a process be suspended as the result of a blocked nested monitor call, all the variables of the monitors, whose exclusivities that process is holding, are saved.

At a conditioned PLOXY point, only if the explicit instruction SAVE(parameters) is used will any monitor variables be invariant. These variables are those from the monitor in which the conditioned PLOXY point occurs, which are specified in the parameter list of the SAVE, as well as all the monitor variables of the other monitors whose exclusivities that process might be holding.

Note: If the (*$B- *) compiler directive, with an answer of N)o to the prompt "Nested Backup" (cf. appendix A), is used, then no variables will be saved at a blocked nested monitor call, and only those explicitly specified in the parameter list of the SAVE instruction will be saved at a conditioned PLOXY point.

In the case of a blocked nested monitor call, the "saved" monitor variables will only be restored once the process has been granted the exclusivity for which it was suspended, and has required all its exclusivities necessary to continue.

For a process suspended at a conditioned PLOXY point, the monitor variables are only restored when the process executes the explicit RESTORE instruction. (This is only possible once the process is "running" again and will thus have reacquired all its necessary exclusivities.) There are safeguards in the form of warning messages during compile time for missing SAVE and RESTOREs and a check during run time to ensure that, in the event of a missing RESTORE, the variables still saved will not also be restored at a subsequent RESTORE. (Note: This can only be detected if the subsequent RESTORE is in a different monitor.)

Reasons for using the same nodes

UCSD Pascal, under which CLANG operates, does not support the DISPOSE(P) procedure where P is some pointer type. By using the same dynamic structures, CLANG is able to reuse nodes already used and finished with by other processes, perhaps on other queues. This is achieved by keeping a queue of possible reusable nodes and only creating a new node when there are no more available.

One of the motivations for the priority strategy used for queueing processes waiting for exclusivity to a monitor (cf. note 3 relating to figure I) was to minimise the number of new nodes created, and hence reduce the amount of dynamic storage required by the interpreter. A process holding several exclusivities will have a number of monitor variables saved, with one node per variable, so the delay before these nodes can be released (and thus reused), must be kept to a minimum.

Having the same node for several applications does mean that sometimes, in the code, the field of the node does not seem to correspond to what is being assigned to, but where they occur these discrepancies have been commented.

eg. VARSTACK^.PRIORITY := AD; (*address*)

(Aside: Use could have been made of variant records in the implementation of these nodes, but as the type of the fields were the same for all the applications it was felt that this added complexity was unnecessary.)

4.3.1 Conclusions on the monitor concept in CLANG

The conclusions reached in this section apply to the monitor concept in CLANG as opposed to the other languages assessed. The analysis of the monitor concept in the realm of concurrent programming is dealt with in chapter 6.

One of the explicit requirements of this thesis was the design and implementation of the monitor concept in CLANG. Thus every effort was made to accommodate all the possible permutations that can arise when concurrent processes synchronise and communicate by means of monitors. Here perhaps CLANG differs from the other languages in that in these languages the features for concurrent process synchronisation and communication were but one small aspect in the broader design requirements.

One criticism that could possibly be levelled at CLANG's in depth considerations, is that certain permutations should never arise in a "real" enviroment and indeed the languages Concurrent Pascal, Edison, Modula-2 and Pascal Plus are being used in "real" enviroments (eg. the writing of operating systems). If these permutations are in fact purely of academic interest, then they fall well within the scope of CLANG's development as a teaching language. A student should not be interested in being told:

"Oh, that case is not catered for as it should never occur in a "real" enviroment !"

What after all is a "real" enviroment ?

One fact that has intrigued the author in the assessment of the other languages is the apparent lack of consideration for the invariance of monitor variables. Much time has been spent by the author in considering whether this invariance is really necessary. It is possible to declare a number of monitor variables local to the monitor procedures of functions and thus ensure their invariance, but there are some variables for which this is not possible. To simply assume that it is not necessary to ensure that these variables are invariant, but only that they are accessed in mutual exclusion, is inviting disaster (cf. example 1 of section 4.1), and thus limiting the potential of the monitor concept.

In some instances, eg. Modula-2 and Pascal Plus, the global exclusion mechanism partially avoids the problem, and the current monitor exclusion technique of Concurrent Pascal limits the problem to a single monitor's variables, but the problem of invariance has not been completely eradicated and further complications of significant loss of parallelism and potential deadlock have been introduced.

The conclusion reached is thus: the implementation of the monitor concept in CLANG comes the closest, out of the five languages assessed, to solving all the problems associated with the monitor concept as a means of realising concurrent process synchronisation and communication.

Chapter 5: The synchroniser concept

> " ... a more natural approach to interprocess communication results if data transmission and synchronisation are considered to be two inseparable activities."
>
> S.J. Young [You82]

A synchroniser is the name applied to the construct available in the language CLANG for interprocess synchronisation and communication by means of message passing, or more specifically the method of rendezvous.

Message passing is based on the belief that data transmission and synchronisation are two inseparable activities. In its basic form it can be viewed as extending semaphores to convey data as well as to implement synchronisation.

One particular method of message passing is known as the rendezvous. In this method communication and synchronisation consist of processes sending and receiving messages. Communication is accomplished because a process, upon receiving a message, obtains values from the sender process. Synchronisation is accomplished because a message can only be received after it has been sent, thus constraining the order in which the two events can occur. During the "rendezvous" both processes remain synchronised and the message is transferred, whereafter both resume their respective activities independently.

The analogy can be drawn from human behaviour where two people meet (with one possibly waiting for the other), perform a transaction, and then go their separate ways again.

The original rendezvous model proposed by Hoare in 1978, [Hoa78], implemented process interaction symmetrically by treating both communicating partners equally. In Hoare's proposed language CSP (Communicating Sequential Processes) [Hoa78], the concurrent processes must synchronise in a rendezvous in order to transfer data. Whichever process issues the transfer command first is delayed until the other process issues its transfer command. Any actual data transfer is then assumed to take place instantaneously and both processes then proceed. This mechanism is symmetric in that both processes must explicitly name the other in order to enter a rendezvous. Symmetric communication poses the problem that it becomes impossible to write a general purpose process to deal with requests from any other process not necessarily known to it. For this reason the alternative approach of asymmetric communication was adopted by Brinch Hansen in his proposal [Bri78] and subsequently implemented as the method of rendezvous in the languages Ada and CLANG.

Asymmetric communication involves only one process (the so-called "client" process) naming the other process (the so-called "server" process) in order to perform a rendezvous.

An asymmetric rendezvous can be represented at the language level by including an "accept" statement in the server process. This accept statement generally takes the form:

```
accept REQUEST(parameters) then
 begin

   (*accept statement body*)

 end
```

The actual data transfer is performed in the same way as in an ordinary procedure call, that is, the actual parameters supplied in the request for rendezvous are bound to the formal parameters supplied in the specification of the accept statement.

The request for rendezvous consists of the client process explicitly naming the server process as well as the service required.

<u>ie.</u> server.REQUEST(parameters)

The two processes remain locked in rendezvous while the body of the accept statement is executed. The body of the accept statement is thus effectively executed as a critical section, which is necessary because the parameters to the accept statement are to be strictly local to it.

An advantage of this kind of mechanism for concurrent process synchronisation and communication is that the programmer can never be uncertain as to the state of a process when a message is sent to it; the process must be executing a rendezvous statement (request or accept) and so must the process that sent the message.

The simple use of accept statements to effect a rendezvous results in a very "tight" form of synchronisation of processes, prohibiting asynchronous behaviour and thus reducing the potential parallelism of the processes in the system. This problem is solved not by compromising the rendezvous principle, but by introducing the possibility of non-deterministic selection of accept statements. In addition to this certain situations, depending on the state of the data structures, may warrant conditions being imposed on the selection of an accept statement.

An additional construct may be introduced whereby a server process can avoid executing an accept statement and thereby committing itself to wait for a client process to rendezvous, until a client process is known to be actually waiting. The additional conditions can be imposed by the use of guard conditions, [Dij75], embedded in this construct and associated with the appropriate accept statements.

This construct typically takes the form of a "select" statement consisting of a set of requests for rendezvous that the server can handle (specified by accept statements) from which an arbitrary choice can be made of one accept statement that will not cause a delay.

A guard condition typically consists of a Boolean expression preceding an accept statement. When a select statement is entered all the guard conditions are evaluated and then only those accept statements whose preceding guard conditions evaluate to true will be considered as candidates for selection. A missing guard condition is considered as evaluating to true.

To illustrate the constructs used to implement the rendezvous technique, consider again the classic example of the so-called Warehouse problem (as mentioned in chapter 3). The interactions of the producer and consumer via a warehouse, which can only store a maximum of one item at a time, can be coded using the simple accept statement approach to rendezvous as follows:

```
program CLASSICEXAMPLE;

 synchroniser WAREHOUSE;
  var SHOP;
  entry DEPOSIT(ITEM), REMOVE(var ITEM);

   begin
    repeat
     accept DEPOSIT(ITEM) then
      begin
       SHOP := ITEM
      end;
```

```
      accept REMOVE(var ITEM) then
       begin
        ITEM := SHOP
       end
     forever
    end;     (*WAREHOUSE*)

   procedure PRODUCER;
    const SWEET = 1;
    var ITEM;
     begin
      repeat
       ITEM := SWEET  (*produce item*)
       WAREHOUSE.DEPOSIT(ITEM)
      forever
     end;   (*PRODUCER*)


   procedure CONSUMER;
    var ITEM, MOUTH;
     begin
      repeat
       WAREHOUSE.REMOVE(ITEM);
       MOUTH := ITEM   (*consume item*)
      forever
     end;   (*CONSUMER*)

  begin   (*CLASSICEXAMPLE*)
   cobegin
    WAREHOUSE;
    PRODUCER;
    CONSUMER
   coend
  end.    (*CLASSICEXAMPLE*)
```

In the above example it is not necessary to use the non-deterministic select statement as the order in which the producer and consumer interact is constrained (by the size of the warehouse) to a deposit by the producer followed by a removal by the consumer.

The more complex example of a warehouse of size greater than one must make use of the select statement and guard conditions to increase the parallelism between the producer and the consumer processes (and the warehouse synchroniser) and to prevent the unacceptable occurrences of a producer attempting to deposit an item in the warehouse that might already be full or the consumer attempting to remove an non-existent item. The modified warehouse might now be coded as:

```
synchroniser WAREHOUSE;
 const NONE = 0 ;
       LOWER = 1; UPPER = 6;
 var SHOP[LOWER:UPPER], STOCK;
 entry DEPOSIT(ITEM), REMOVE(var ITEM);

  begin   (*WAREHOUSE*)
   STOCK := NONE;   (*warehouse initially empty*)
   repeat
    select
     STOCK < UPPER : accept DEPOSIT(ITEM) then
                      begin
                       STOCK := STOCK + 1;
                       SHOP[STOCK] := ITEM
                      end;
     STOCK > NONE  : accept REMOVE(var ITEM) then
                      begin
                       ITEM := SHOP[STOCK];
                       STOCK := STOCK - 1
                      end
    end    (*select*)
   forever
  end;    (*WAREHOUSE*)
```

Note: The two processes, the producer and the consumer, are asynchronous and thus the additional active process, the synchroniser, is needed to act as a buffer so as to effect the transfer of the item from the producer to the consumer.

The rendezvous technique brings about an unification of the concepts of synchronisation and communication. Conditioned synchronisation is possible by means of simple accept statements (as in the first example) or by means of the guard conditions (as in the second).

The interactions involved take place between two active processes (as opposed to the passive construct of the monitor) and thus one consequence of the rendezvous method is likely to be an increase in the number of concurrent processes in a system.

The following sections will examine to what degree the concept of the rendezvous has been developed in the languages CHILL and Ada. This information will then be contrasted with the synchroniser construct available in CLANG. Also included is a description of how the synchroniser and its related features were implemented in CLANG.

5.1 The rendezvous concept in other languages

The rendezvous concept arose from ideas proposed in 1978 by Hoare, [Hoa78], and Brinch Hansen, [Bri78], in which inter-process synchronisation and communication were regarded as inseparable activities.

Being a reletively new concept (the monitor concept was introduced in 1974 [Hoa74]), the rendezvous has only been introduced into only a handful of languages. Two of these languages, which are assessed in this section, CHILL and Ada, enjoy enormous support from the CCITT (Telecommunications affiliate of the United Nations) and the United States Department of Defense, respectively, but at the time of writing, although their language designs are now fixed, full compilers are scarce.

Thus, as with the assessment of the monitor concept in other languages (except Modula-2), this assessment of the rendezvous concept as implemented in CHILL and Ada is based solely on information gleaned from the literature, [Fid83], [Bra82], [Bar80], [Ich79], [Uni81], [You83], and not from any practical experience.

5.1.1 CHILL

CHILL was based on the sequential languages Pascal, PL/1 and Algol 68, and developed in 1981 under the auspices of a CCITT study group specifically for real-time enviroments as well as for general systems and sequential programming [Fid83].

Several mechanisms are provided in CHILL for concurrent process synchronisation and communication.

Event mode locations and the operations continue, delay and delay case that can be performed on them enable explicit synchronisation of processes. When declaring an event mode location it is possible to specify the maximum number of processes which can be delayed on that event at any time.

A process executing a delay statement is suspended on a queue associated with the named event until another process executes a continue operation on the same event. A process may specify a priority when it is queued.

When a process executes a delay case statement it is suspended until a continue operation is performed on any of the named events contained within the delay case statement. For each named event it is possible to specify a different sequence of statements to be performed by the suspended process upon reactivation. It is also permissible for a process to specify a priority status on being suspended when executing a delay case statement. On reactivation a process may identify the process which caused this.

Execution of a continue statement causes the reactivation of the process at the head of the named event queue. If there are no processes delayed on this queue then the continue statement has no effect.

The second mechanism provided for interprocess synchronisation and communication is that of the signal. These signals are used in conjunction with send and receive case statements.

A signal is defined in a signal definition statement, may optionally have a message part, and may specify which process type can receive the signal.

Should a message part be included in the signal definition statement then the signal send statement will transfer a list of values to the named signal. Also optionally sent in the signal send statement can be a priority and an identification of the intended receiver. This identification must not conflict with any specification given in the signal definition statement.

A process can receive a signal by executing a receive case statement, which specifies a list of signals which may be received, each of which may have its own associated sequence of statements.

If none of the named signals is pending and no else clause has been included in the receive case statement then the process is delayed until one of the signals is forthcoming.

Note: Unlike the continue operation on an event mode location, the signals are persistent, which means that if no process is currently waiting to receive the signal then it is saved (becomes "pending") until a process needs it.

If more than one appropriate signal is pending, the signal with the highest priority is chosen; if several signals share the highest priority then the choice of these is implementation dependent (eg. random, FIFO etc.).

Yet another method available in CHILL for interprocess synchronisation and communication is provided by buffer mode objects and the operations send, receive and receive case on them. An object declared to be of type BUFFER must include the type of its elements and optionally the number of elements that the buffer can hold.

The send operation causes a specified value to be placed into a buffer location. If the buffer is full, the process executing the send statement is delayed, with an optional priority, until a space becomes available or the value being sent is consumed.

A process executing a receive expression will obtain one value from a set of values available in the buffer and delayed sending processes associated with that buffer (if any). If there are no values available then the process executing the receive expression is delayed until a value is sent to the buffer.

The buffer receive case statement allows a process to obtain a value from one of a number of named buffers and their associated suspended sending processes (if any), with a separate sequence of statements for each and an optional else clause. If no values are available and no else clause is specified, then the process executing the buffer receive case statement is delayed until a value arrives. The identity of the sending process may also be obtained.

Because the choice of value available to a process executing a receive expression or buffer receive case statement includes a value from the buffer as well as from any process that might be delayed after performing a send to the full buffer, the execution of such a statement will result in the reactivation of a delayed sending process (if there are any).

Finally, the CHILL concept of a region makes available a means of providing processes with mutually exclusive access to locations. These regions may only be declared at the outer level of a CHILL program (known as the "outer process"). A region's visibility is controlled by the statements grant and seize. Processes wishing to access locations declared in a region may only do so by calling procedures, which may not be recursive, defined within and GRANTed by the region. Objects declared within a region, which are to be shared by processes, may not be visible outside the region.

Any process attempting to access a region to which another process already has access is delayed until the exclusivity is released, either by that process leaving the region or being delayed within the region. If more than one process is suspended awaiting access to a region and the region is released, a process will be selected according to some algorithm which is implementation defined (eg. FIFO etc.).

In all CHILL provides four different methods for concurrent process synchronisation and communication. The reason for this is that CHILL was developed by a committee with the result that several alternatives were provided when unanimous agreement could not be reached [Fid83].

This has resulted in some of the constructs being syntactically almost identical, eg. the signal receive case statement and the buffer receive case statement, and yet they function differently, eg. the CHILL buffers differ from the signals in that they enable the user to control the allocation of the buffers explicitly, whereas the allocation for the signals is performed "automatically".

The facilities of the events, signals, buffers and regions seem to clutter the language when it appears that either the signals and modules (the data abstraction facility in CHILL) or buffers and processes are sufficient to provide all the concurrency requirements of a programmer [Fid83]. Even the CHILL introduction warns that:

> "...care should be taken not to mix the various methods within one subsystem." [Bra82]

One wonders how easy it would be for a programmer to learn the concurrency features when faced with so many subtly different constructs.

5.1.2 Ada

In 1976 the United States Department of Defense drew up a set of requirements they felt were desirable for a standard real-time programming language. They appreciated that the lack of a single standardised language was resulting in high costs being incurred not only in the development of new systems but also in the maintenance of existing ones. An evaluation of existing languages was undertaken to see if any of these could meet their set of requirements.

The evaluation concluded that no existing language fully met the requirements, although three languages (Pascal, PL/1 and Algol 68) had sufficiently sound and well proven structures to serve as the base for a new language design.

The design of this new language was then contracted out to competing organisations. Seventeen tenders were received, of which the language designed by Cii Honeywell Bull, primarily based on Pascal, was eventually selected in May 1979 to become the language Ada.

In 1981 the reference manual for Ada was published [Uni81], but as yet few compilers for a full version of Ada have been validated.

Ada uses the word "task" for a program activity which proceeds in parallel with others. A task is thus exactly synonymous with a process. "Task" will be used in this report in keeping with Ada notation.

A task consists of two parts: a specification and an optional body. The specification may contain entry declarations (see below) and a representation specification which may specify how the entries or the task itself map onto the underlying hardware. The task body may contain local declarations and statements. Ada allows a task to be declared as a type, thus permitting multiple instances of the same task.

The primary means of synchronisation and communication between tasks are <u>entry calls</u> and <u>accept statements</u>.

The entry declarations specify the entries that other tasks may call, and the formal parameters by means of which the communication may take place.

The actions that are to be performed when a declared entry is called are contained within the corresponding accept statements.

A task can call an entry in another task by specifying the entry name and the actual parameter list. If the task which owns the called entry has yet to reach the corresponding accept statement then the calling process is suspended. Similarly a task executing an accept statement, prior to the occurrence of any call to the named entry, is suspended until such a call happens. Thus the use of entry calls and corresponding accept statements always result in rendezvous.

The calling task remains suspended until the called task completes the statements contained within the accept statement (if any). After the rendezvous both tasks continue their (independent) parallel execution.

It is possible to declare a "family" of entries with the same name and parameters, with individual entries being accessed via indices. Several entry calls to the same accept statement are dealt with on a first-in-first-out basis; each rendezvous at an accept statement removing just one calling process from the queue. An exception is raised [Uni81] if an attempt is made to call an entry in a terminated task, or if the entry's family index is out of range.

A task body may contain one or more accept statement per entry declaration.

The accept statement enables a task to wait for some event to happen - signified by the calling of the corresponding entry. (Aside: An accept statement without parameters is purely a point of synchronisation.) To wait for several events all to have happened merely requires a sequence of accept statements.

To wait for one of several alternatives is not that easy and for this purpose Ada has introduced the select statement.

Three different types of select statement are provided in Ada.

The selective wait statement allows two or more alternatives to be named, each with an optional condition which must be satisfied before the associated alternative may be selected; an else part may also be included.

The form of the selective wait statement is:

```
select
    [when CONDITION =>]
      ALTERNATIVE
 or [when CONDITION =>]
      ALTERNATIVE
         ...
 [else
    STATEMENTS]
 end select;
```

An ALTERNATIVE may consist of:

(1) An accept statement plus other statements;

(2) A delay statement, which suspends the task for at least the time interval specified, plus other statements; and

(3) the reserved word TERMINATE which terminates the execution of a task.

A selective wait statement may only contain at most one TERMINATE and may not have delay statements as well as a TERMINATE. The use of TERMINATE or a delay statement precludes the use of the ELSE part.

As the selective statement is entered each ALTERNATIVE is examined to see if its associated when clauses evaluates to TRUE. If this is so then the ALTERNATIVE is considered to be <u>open</u>.

Based on the results of this examination the following actions may occur:

If there is one open ALTERNATIVE containing an accept statement to which a corresponding entry call has been made, <u>ie.</u> the called task will not be suspended, then it is chosen and a rendezvous initiated. Should there be more than one open ALTERNATIVE in this category then the choice is implementation dependent (eg. random, cyclic etc.).

If there have been no corresponding entry calls to any of the accept statements in the possible open ALTERNATIVEs then the task is suspended until one of these entry calls occurs. Should the selective wait statement contain an open ALTERNATIVE with a delay statement then if an entry call is not forthcoming before the time specified in the delay statement then that ALTERNATIVE will be executed instead.

An open ALTERNATIVE with a TERMINATE will only be selected if the parent block in which the task has been declared is ready to terminate or be left, and is only waiting for the termination of its dependent tasks.

The ELSE part is only executed if none of the ALTERNATIVEs are open.

The second type of select statement involves a conditional entry call. In this select statement a call to an entry will be made only if the rendezvous is immediately possible; otherwise the ELSE part is executed.

The form of this select statement is:

```
select
  ENTRY_CALL [STATEMENTS]
 else
  STATEMENTS
end select;
```

Finally the third type of select statement consists of a timed entry call. An entry call is only made if the rendezvous can be performed within a certain specified time; otherwise the delay statement is executed.

```
select
  ENTRY_CALL [STATEMENTS]
 or
  delay statement [STATEMENTS]
end select;
```

Ada programs might have to meet real-time response constraints; hence this type of select statement is available to prevent or control the length of time a task is delayed.

One loop-hole existing in connection with the integrity of variables during concurrency is that tasks may interact via shared variables declared in the enclosing block - there is no special mechanism provided for synchronising access to these shared variables; the responsibility for their integrity is left with the programmer. A more serious problem associated with this "loop-hole" is the subtle security risk it poses in the use of entries to ensure mutual exclusion. The parameters in an entry call are evaluated before entry to a rendezvous. This means that two tasks can call an entry simultaneously naming a single common shared variable as a variable parameter. If that parameter is used as a key to gain access to a resource then both tasks may be given access to the resource simultaneously because the initial value of the key is copied into the entry before rendezvous.

Ada is a large and complex programming language intended mainly for embedded computer applications, but it is also suitable for a large variety of uses. [You82]

Its success is assured, not only because it has an intrinsically good design which incorporates all the best ideas of the last decade into a clean and uniform language framework, backed by the considerable influence of the United States Department of Defense, but also because it will be part of a complete software development system.

As well as a compiler, an Ada support system will provide standard editors, debugging tools, text formatters, library management systems etc. Furthermore the entire system will be standardised program and programmer portability [You82].

The size and complexity does have its drawbacks. It seems likely that tolerable compilation speeds will only be achievable on large minis and main frame computers. Some of the methodologies used in Ada, eg. tasks mechanisms, may be completely alien to the average programmer schooled in the traditional high level language so that training a programmer to a working competence in the full Ada language will be a substantial problem compounded not only by the size of the language, but also by the need to design programs the "Ada way".

This consequence is not altogether surprising as the major motivation for developing Ada was to improve existing software design and implementation practices [Ich79], [Uni81], a step forward for which substantial training costs and effort are clearly unavoidable.

Here is perhaps where languages such as CLANG can fit in: as a bridge between existing methodologies and the introduction of new, hopefully better ideas.

It should not be surprising then that the constructs for teaching the rendezvous technique, in CLANG, were modelled on those available in Ada.

5.2 Using and implementing Synchronisers in CLANG - practical details

The synchroniser is the message passing equivalent of the monitor concept of synchronisation and communication via mutual exclusion.

This section will examine:

(1) The semantics of the constructs available in CLANG for allowing concurrent processes to synchronise and communicate by means of the rendezvous technique, and

(2) Illustrated details of how these constructs were actually implemented.

The actual syntax details of the synchroniser and its associated constructs can be found in appendix A, while the Pascal code comprising the CLANG compiler and interpreter can be found in appendix B.

Example:

The warehouse (as mentioned in chapter 3) may be coded as a synchroniser as follows:

```
synchroniser WAREHOUSE;
 var SHOP;
 entry DEPOSIT(ITEM), REMOVE(var ITEM);(*entry points*)

  begin   (*WAREHOUSE*)
   repeat
    accept DEPOSIT(ITEM) then
     begin
      SHOP := ITEM
     end;
    accept REMOVE(var ITEM) then
     begin
      ITEM := SHOP
     end
   forever
  end;    (*WAREHOUSE*)
```

The message passing methodology used in CLANG is a Many-to-one rendezvous situation, where many "client" processes may request rendezvous with one "server" process.

A "client" process is any concurrent process that wishes to synchronise and communicate with the "server" process.

The "server" process is the synchroniser.

A synchroniser is an active process and as such must be launched, as a normal process is, from inside a Cobegin..Coend construct. Being an active process it executes concurrently with the "client" processes until a rendezvous is established.

Once a rendezvous is established the "server" and "client" processes are ready to communicate.

The list of requests that a synchroniser can serve are termed <u>entry points</u> and are declared within synchronisers under the ENTRY declarations (along with the parameters via which the communication is actually effected). These entry points are the only parts of a synchroniser that are visible outside the synchroniser (and bear a vague resemblance to forward declarations of procedures).

A process wishing for a rendezvous with the synchroniser performs a request to the required entry point declared inside the synchroniser by appending the named entry point together with the necessary actual parameters, to the name of the synchroniser separated by a period ('.').

<u>ie.</u> synchronisername.entrypoint(parameters)

(Aside: The entry point request is similar to a call to a starred procedure of a monitor.)

The process is then suspended until the rendezvous is complete after which both the "client" process and the synchroniser continue their concurrent execution. The section of code in the synchroniser in which the actual communication takes place is contained within an <u>accept statement</u>.

The entry point request and the accept statement form the point of synchronisation between the "client" and the "server" processes.

If a synchroniser, during the execution of its code, should reach an accept statement for which, as yet, there has been no corresponding request, then the synchroniser is delayed until such time as one occurs. Similarly if a process performs an entry point request and the synchroniser, in which the entry point is declared, has yet to reach the corresponding accept statement, then the process is delayed until the accept statement is reached and the rendezvous performed by the synchroniser.

A request for rendezvous must match to an entry point declared in the named synchroniser, which in turn must match to that used in the corresponding accept statement. The formal and actual parameters in all instances must correspond.

The parameters of an entry point are strictly local to the accept statement for that entry point, and may be passed by value or by reference.

Note: An entry point without parameters is purely a synchronisation point.

Many processes may request one entry point and there may be many accept statements, each with its own sequence of actions, for that entry point declaration.

For example:

For the entry point DEPOSIT, there might be two accept statements:

```
synchroniser WAREHOUSE;
 var SHOP, TRUCK;
 entry DEPOSIT(ITEM), ...

  begin   (*WAREHOUSE*)
   ...
    accept DEPOSIT(ITEM) then
     begin
      SHOP := ITEM
     end;
    ...
    accept DEPOSIT(ITEM) then
     begin
      (*pay the client*)
      TRUCK := ITEM (*load item directly*)
     end;
    ...
   end;   (*WAREHOUSE*)
```

The requests for rendezvous for a particular entry point are performed on a First-in-First-out basis. Each execution of an accept statement deals with just one request.

If a synchroniser can never execute the corresponding accept statement for a request then deadlock may result. Similarly if a synchroniser executes an accept statement for which no request is ever forthcoming then deadlock may again result.

The select statement in CLANG enables asynchronous behaviour in a program and, increases potential parallelism by relaxing the "tight" synchronisation of the accept statement and entry point request.

The select statement grants a synchroniser a great deal of flexibility in that it allows it to "choose", from a list of possible requests to be serviced, a rendezvous for which there is a "client" process already waiting, and thus avoid being delayed.

The form of the select statement is:

```
select
 GUARD CONDITION1 : accept REQUEST1(parameters) then
                     begin
                      STATEMENTS
                     end;
                ....
  GUARD CONDITIONn : accept REQUESTn(parameters) then
                      begin
                       STATEMENTS
                      end;
  [else
    begin
     STATEMENTS
    end  ]
 end;    (*select*)
```

Further control over which accept statements the synchroniser may choose is exerted by the use of guard conditions preceding each accept statement. A guard condition may consist of a Boolean expression or the reserved word NOGUARD, which is equivalent to a Boolean expression which always evaluates to true.

Only those accept statements whose associated guard conditions evaluated to true on entering the select statement, will be considered for selection, and of these only an accept statement that does not cause the synchroniser to delay will actually be selected and the corresponding rendezvous performed. If there are several accept statements in this category then the choice will be random.

Should all the accept statements, with true guard conditions, if they were to be executed, cause the synchroniser to delay, then it is delayed, but only until the first request for rendezvous for any of these accept statements occurs. Thus the delay time is kept to a minimum; the synchroniser is reactivated and this rendezvous request serviced. After the rendezvous the synchroniser continues executing the statements after the select statement.

All the guard conditions evaluating to false implies that there are no valid accept statements from which the synchroniser can choose. Should this be the case then the else clause is executed if there is one; if not then a run time error will occur.

CLANG restricts the use of accept statements to within synchronisers and thus a rendezvous may only occur between a synchroniser and another process. This other process may not be a synchroniser, as rendezvous requests are not permitted from within a synchroniser.

Being an active process, the synchroniser's variables are not subject to alteration by other processes, and thus the synchroniser has mutually exclusive access to them all the time and they can be used as a buffer in the transmission of messages (in the form of data) from one process to another.

As with Ada there is no mechanism to prevent processes (including synchronisers) from "simultaneously" altering a program's global variables and it is thus up to the programmer to ensure that this never happens.

Implementing synchronisers in CLANG - illustrated details

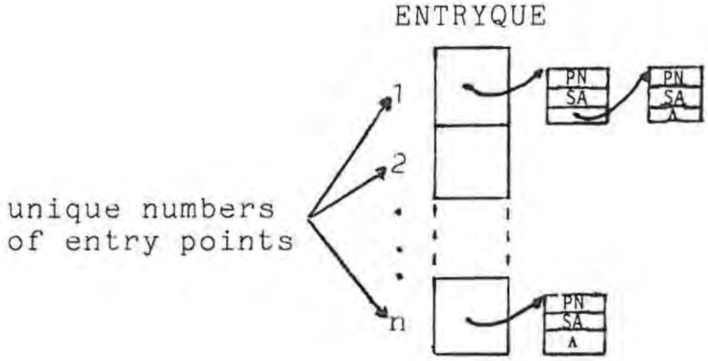
The parser and interpreter making up the compiler for the language CLANG are integrated into one program written in Pascal. (The current implementation is in UCSD Pascal, with use made of as few "extensions" as possible.)

This section includes a description of how the synchroniser and the associated constructs necessary to introduce the rendezvous concept into CLANG were implemented. This description takes the form of flow diagrams with accompanying notes and a detailed example at the end of the section to show how the queues associated with the rendezvous technique are manipulated. It is hoped that the study of this section in conjunction with the listing supplied in appendix B will give the reader insight into how a rendezvous might be implemented.

When parsing a CLANG synchroniser, each entry point is assigned a unique number. This number is used at run time to ascertain at which entry point a rendezvous or an accept statement is being performed.

Use is made of Pascal's pointer facilities to implement the queue associated with each entry point. An array ENTRYQUE of these queues was introduced, the individual queues for each entry point being indexed by its unique number.

ENTRYQUE can be viewed diagramatically as:



PN = process number. This is the index into the process descriptor table for the process (or synchroniser) which is suspended on the entry point queue concerned. This number is assigned just before the concurrent execution of the processes is launched by means of the Cobegin..Coend construct.

SA = start address. This field of a node on an entry point queue contains the start address of an accept statement for this entry point and is thus used only when queueing synchronisers.

One of the fields in the process descriptor table, HELDSET, used for implementing monitor exclusion (cf. chapter 4 section 4.2.1) is also used for implementing the rendezvous concept. HELDSET is used to hold the set of entry point queues on which a synchroniser is suspended as the result of all the accept statements, with guard conditions evaluating to true, causing a delay.

Note: The field HELDSET may safely be reused, as a synchroniser may not be called from within a monitor and, although a monitor procedure or function may be called from within a synchroniser, it is not possible for a synchroniser to be suspended on an entry point due to an accept statement and be delayed waiting to reacquire exclusivities to monitors simultaneously.

For the same reasons as given in chapter 4 section 4.3, the same type of nodes are used for processes, including synchronisers, which are suspended on an entry point queue. This, however, does result in what appear to be obscure statements in the interpreter:

eg. ENTRYQUE[U]^.PRIORITY := PTAB[CURPR].P

where PTAB[CURPR].P is the start address of an accept statement and clearly has nothing to do with a priority. These apparently confusing statements have been well commented.

An accept statement is the synchronisation point in a synchroniser where the rendezvous will be performed. Figure I shows diagramatically the actions undertaken by a synchroniser on executing an accept statement.

Notes relating to figure I

(1) If there has yet to be a request for rendezvous on the entry point corresponding to the accept statement concerned then the queue, indexed in ENTRYQUE by the unique number of the entry point, will be nil.

(2) When a synchroniser is suspended on an entry point queue it is distinguished from other processes by setting the number field of the relevant node to the process number of the synchroniser plus the constant value PRMAX, which is the maximum number of processes allowable per concurrent system.

The priority field of the node is used to hold the start address of the accept statement causing the synchroniser to delay. This is not actually needed in the case of a single accept statement - the synchroniser program counter will contain the correct value anyway - but is included for uniformity, as it is necessary in the case of an accept statement contained within a select statement, and thus when a request for rendezvous is forthcoming no distinction need be drawn as to which class of accept statement is being dealt with.

(3) The synchroniser will be reactivated by a process executing the P-code signifying a request for rendezvous. (cf. figure II)

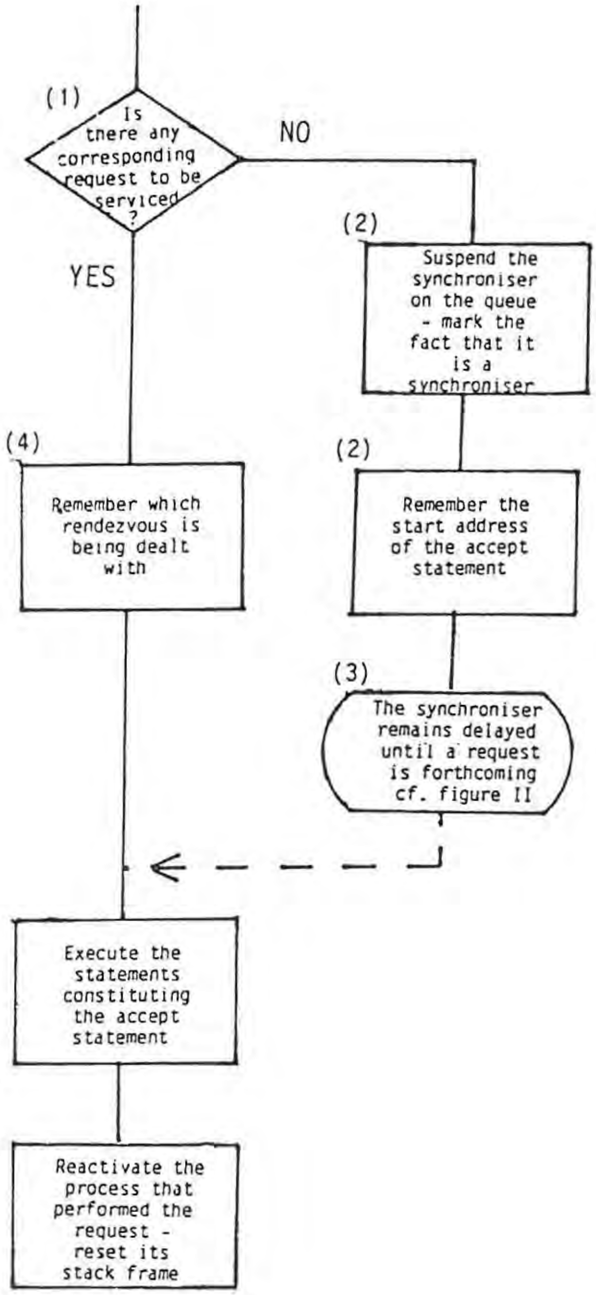


figure I: Executing an accept statement

(4) The entry point parameters to a rendezvous have to be obtained, not from the stack portion of the synchroniser which is servicing the request, but from the stack portion of the process which requested the rendezvous. Remembering the rendezvous is thus necessary, a situation which has necessitated the introduction of the LDE P-code when dealing with entry point parameters.

A request for rendezvous must correspond to an entry point declared within the synchroniser whose name is appended to the entry point concerned.

The actions taken when a process performs a request for rendezvous can be seen diagramatically in figure II.

Notes relating to figure II

(1) A rendezvous request is very similar to a procedure call and so an effective stack frame is created to facilitate the passing and receiving of parameters.

(2) A synchroniser can be detected as the number field of the node examined will be greater than PRMAX (cf. note (2) relating to figure I).

(3) An examination of the synchroniser's HELDSET field will reveal if the synchroniser was delayed in a select statement.

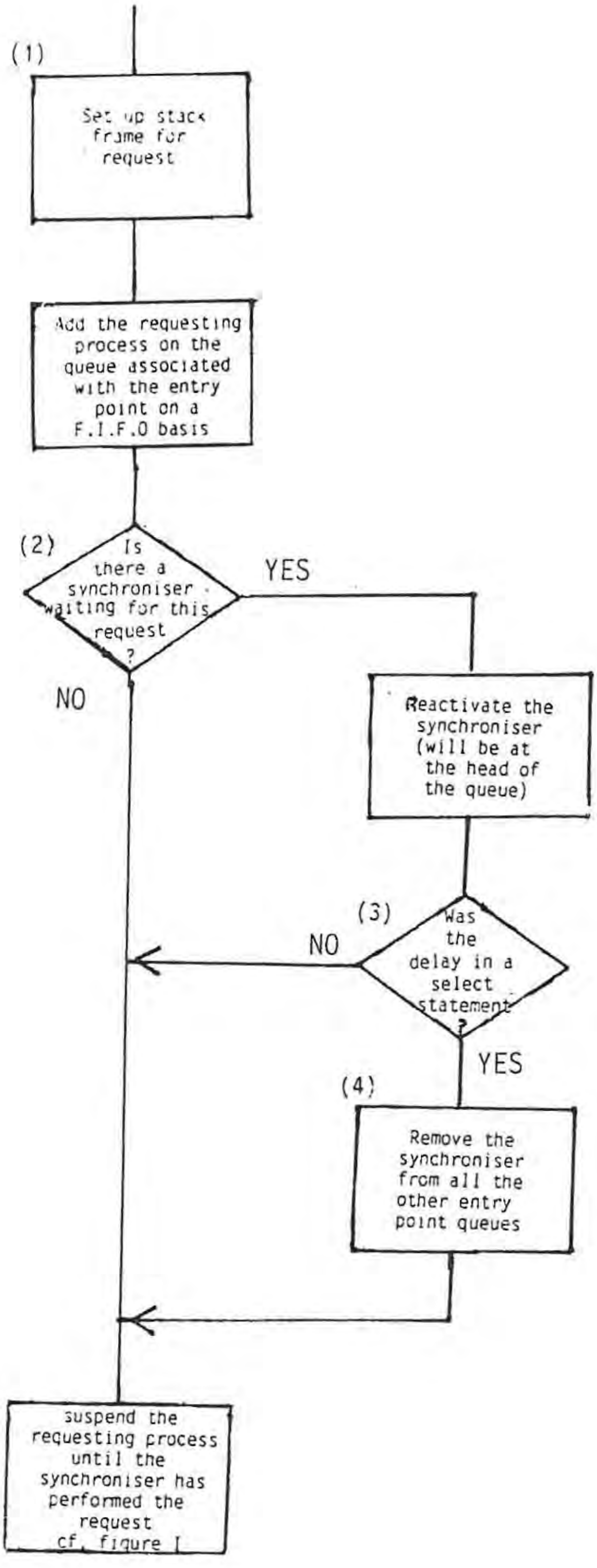


figure II: Request for rendezvous

(4) As in this case the synchroniser is only delayed until the first one of the necessary rendezvous requests is forthcoming, it must be removed from all the other entry point queues on which it was also delayed (cf. note (5) relating to figure III).

The select statement allows the synchroniser to "choose", out of a list of possibilities, a rendezvous to service, thus permitting asynchronous behaviour and increasing the potential parallelism of the system.

The SEL P-code which actually performs the selection, occurs right at the end of the P-codes constituting the select statement. These P-codes are for the guard conditions, the accept statements and the else clause (if any).

On encountering a select statement, the guard conditions must all be evaluated before any accept statement can be chosen for execution.

This is achieved at the P-code level by branching from guard condition to guard condition, bypassing the P-codes constituting the accept statements. After the last guard condition has been evaluated (or if there is an else clause, after this fact has been flagged), a branch occurs to the SEL instruction which will perform the selection, possibly resulting in the synchroniser being delayed.

The selection will result in the program counter of the synchroniser being set to the start address of an accept statement (possibly after a delay) or, if all the guard conditions evaluate to false, to the start address of the else clause if there is one, otherwise the program status, PS, is set to SELCHK, flagging the run time error:

'NO VALID SELECT GUARD'

The flow of execution can be viewed diagramatically as:

(a) <u>The evaluation of the guard conditions</u>

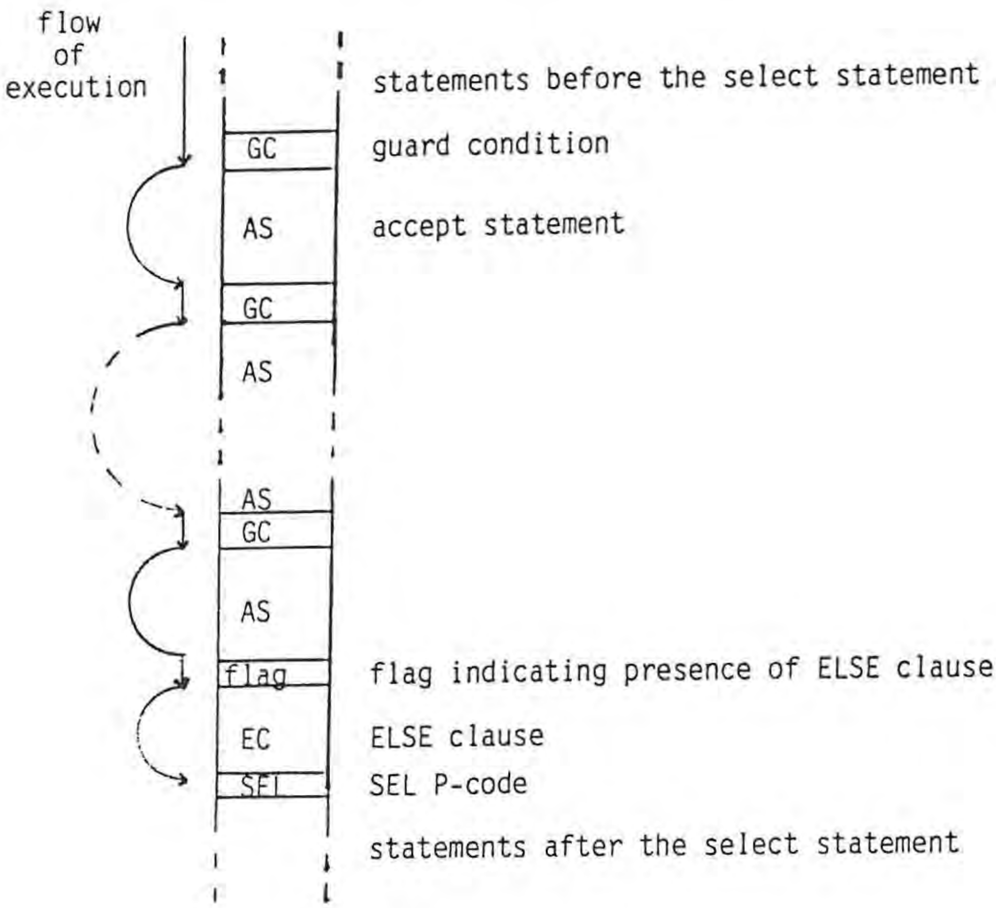


Table of P-codes

(b) The execution of an accept statement (or else clause)

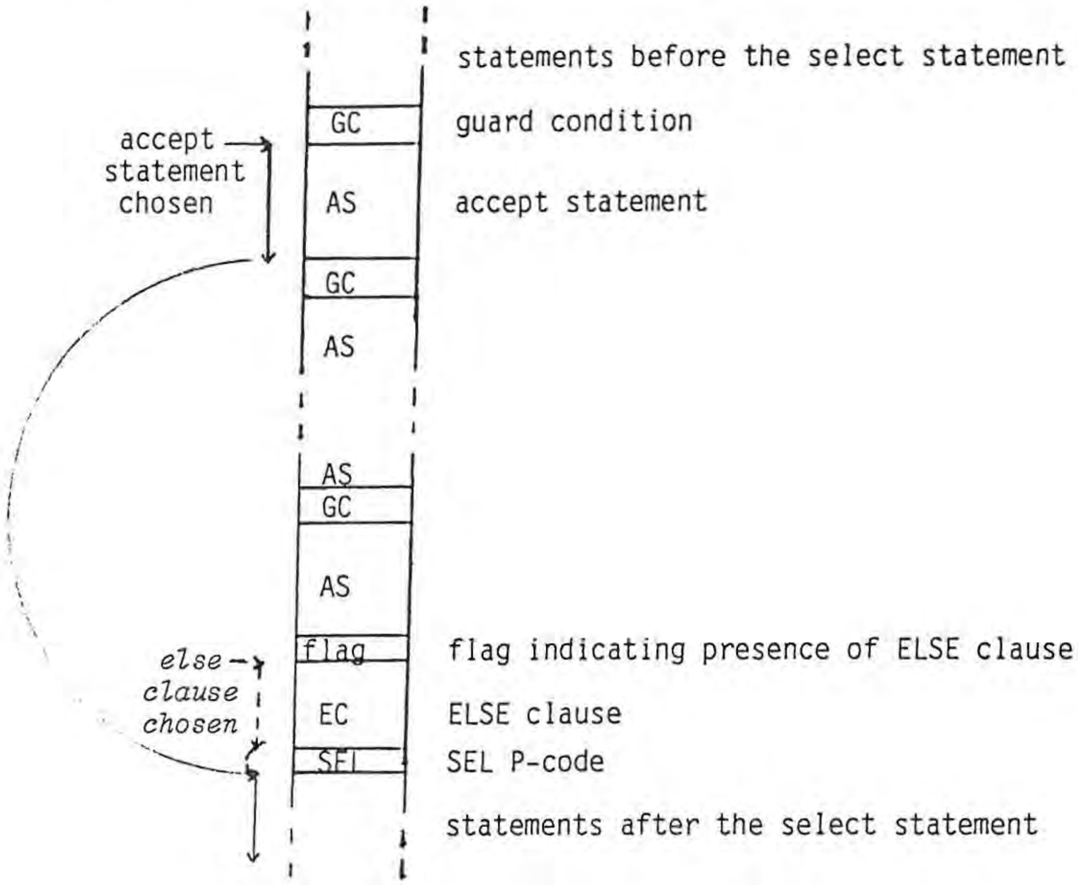


Table of P-codes

Figure III shows the actions undertaken when the SEL P-code is evaluated.

Notes relating to figure III

(1) The evaluation of the guard conditions prior to the execution of the SEL instruction has resulted in a "table" being built up as part of the synchroniser's variables. For each guard condition there are two entries in this "table"; one to hold whether the guard condition is true or false (1 or Ø), (or if it is the else clause, to hold the value 2); and the other to

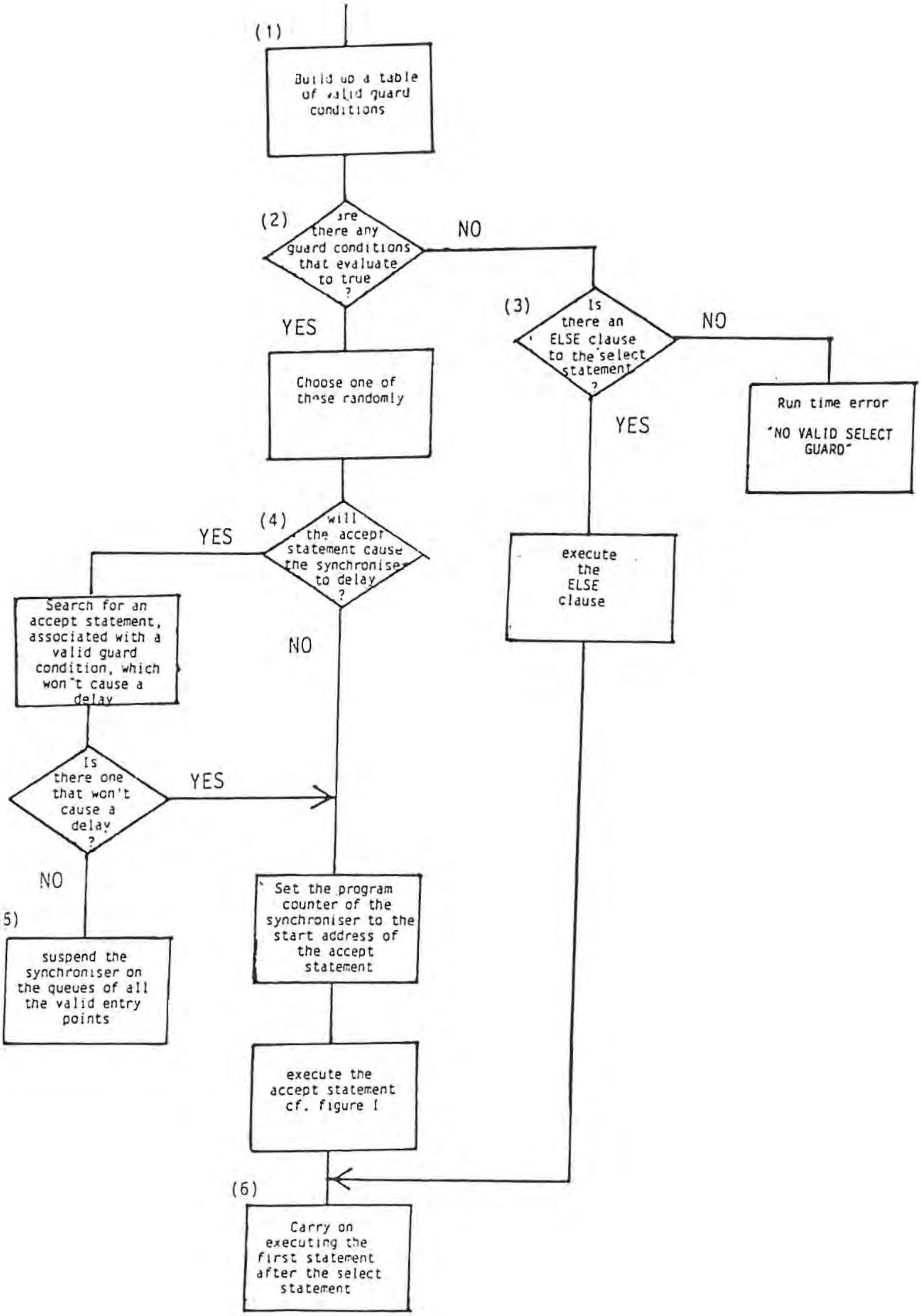


figure III: The select statement

hold the start address of the associated accept statement (or in the case of the else clause, the start address of the statements constituting the else clause).

Using these stored values a list, SELTABLE, is drawn up of the start addresses of those accept statements that are possible for selection.

(2) If this list is empty then all the guard conditions must have evaluated to false.

(3) The value 2 at the end of the "table" of the results of the guard conditions indicates that there is an else clause present in the select statement. If all the guard conditions evaluate to Ø (ie. false) then the program counter of the synchroniser is set to the entry in the "table" associated with the result of 2 ie. the start address of the else clause.

(4) Once an accept statement has been selected the synchroniser can ascertain whether its execution would cause a delay by examining the relevant entry point queue. If this queue is empty then there has yet to be a corresponding request for rendezvous implying that the accept statement concerned would cause the synchroniser to be suspended. If the queue is non-empty then there is at least one process already waiting for that rendezvous to occur and so the execution of the accept statement concerned will not result in the synchroniser being suspended.

(5) The synchroniser must be delayed until the first request for rendezvous for any of the possible accept statements is forthcoming. This is achieved by suspending the synchroniser on all the relevant queues - keeping track of what the queues are by means of the HELDSET field in the process descriptor table. The information needed on the queue is the fact that a sychroniser is suspended on the entry point queue (cf. note (4) relating to figure I) and the start address of the accept statement for that entry point.

If there is more than one accept statement for the same entry point amongst those available for selection, then only one of these will be selected if the corresponding request is the first to arrive. The choice for this selection is random and is done at this stage by ensuring that only one start address is stored along with the synchroniser on the queue for the relevant entry point.

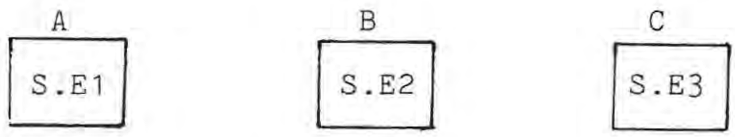
(6) Only one accept statement (or the else clause) is chosen per execution of the select statement. After the accept statement (or else clause) has been executed the statements after the select statement are executed. (Obviously the process suspended while the rendezvous is performed will then proceed concurrently with the synchroniser once again.) If there are still more requests for rendezvous to be serviced then the select statement must be contained within some sort of loop. (It is the responsibility of the programmer to ensure this.)

Detailed example

This example is designed to give the reader further insight into how the queues relating to entry points are manipulated when processes request and synchronisers service rendezvous. It should be studied in conjunction with figures I, II and III.

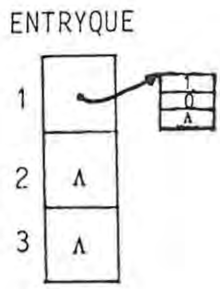
Consider the system consisting of a synchroniser, S, in which three entry points E1, E2 and E3 have been declared, and three processes A, B and C.

```
synchroniser S;
 entry E1, E2, E3;
  ...
   accept E1 then


    ...
   select
     NOGUARD : accept E2 then

     NOGUARD : accept E3 then

    end;    (*select*)
```



There follows a trace of possible events and their consequences.

Process A requests a rendezvous at entry point E1. The synchroniser S has yet to reach the corresponding accept statement so process A is suspended on ENTRYQUE indexed by the unique number of the entry point E1 (ie. 1).



Processes available for scheduling = (S, B, C)

Note: The process numbers are assigned to the processes just before concurrency is launched. Assume for this example that:

process A = 1

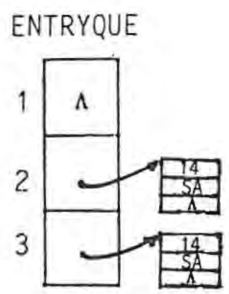
process B = 2

process C = 3

synchroniser S = 4

Synchroniser S reaches the accept statement for entry point E1 and performs the rendezvous requested by process A. Once the rendezvous has been performed (after the accept statement), process A is reactivated and is ready for scheduling once more.

Synchroniser S executes the select statement. Although both the guard conditions evaluate to true (NOGUARDs) there has yet to be a request for either accept statement, so synchroniser S is suspended on both the queue for entry point E2 and E3.

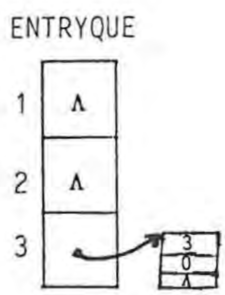


HELDSET $[2,3]^{S}$

Processes available for scheduling = (A, B, C)

S.A = start address for the associated accept statement

Process C now performs a request for rendezvous at entry point E3. The queue for E3 is examined - there is a sychroniser there (number > PRMAX). Process C is suspended and synchroniser S is reactivated and removed from all relevant entry point queues.



HELDSET $[\ ]^{S}$

Processes ready for scheduling = (S, A, B)

Process C is only delayed as long as it takes synchroniser S to perform the accept statement for entry point E3.

5.2.1 Conclusions on the synchroniser concept in CLANG

> "Ada is a jungle of intertwined features; one suspects it was designed as a challenge to compiler writers, not as a tool for software engineers"
>
> Joel McCormack and Richard Gleaves [McC83]

The synchroniser concept in CLANG is a simplified version of the rendezvous facilities available in Ada. CLANG does not support the conditional entry call or the timed entry call, but other features are available for possible usage in conjunction with the synchroniser. These are the ACTIVEINSYSTEM, RUNNINGINSYSTEM, STOPCONCURRENCY and SWITCH commands (see appendix A: The User Manual, chapter 4).

As the opening quote suggests Ada, and to a certain extent CHILL, confront the user with a plethora of new concepts and constructs. These are immersed in a syntax which, although originally based on that of Pascal, is so complex and vast as to appear only remotely similar to the high level languages, such as Pascal, with which the user might be familiar.

It is true to say that Ada and CHILL have incorporated most of the good ideas of the last decade, (and CHILL some of the not so good constructs as well), but it is just this overwhelming flood of new constructs that will make the teaching and understanding of just one aspect difficult and time consuming.

Also the sheer size of Ada and CHILL makes their universal availability only a remote possibility in the immediate future.

This is why experimental languages, such as CLANG, will be able to hold their own. Although the rendezvous facilities of CLANG are not as complex and complete as those of Ada, they are clearly distinguishable to a programmer and are used alongside notations fairly synonymous with those of Pascal (cf. chapter 2). This should allow for the easy teaching and studying of the synchroniser concept on available microcomputers so that when a programmer is eventually confronted with Ada or CHILL, the concept of the rendezvous will not be unknown. This should enable the fairly rapid mastering of at least one (perhaps the most important) aspect of these complex languages.

Chapter 6: The Monitor and Synchroniser concepts - Comparisons and conclusions

> "A programming language needs BOTH types of constructs to support the spectrum of concurrent applications"
>
> W. Eventoff, D. Harvey and R. Price [Eve80]

The monitor and the synchroniser concepts arose from differing ideas on how interprocess synchronisation and communication might be performed.

The monitor concept is based on communication via passive abstract data structures which are accessed in mutual exclusion, whereas the synchroniser (or rendezvous) concept follows the line of direct, synchronised transfer of messages (in the form of parameters) between two active processes.

This section will attempt to highlight the areas of difficulty associated with each concept (with special reference to the implementation in CLANG) and endeavour to show that although one concept may be a better choice for usage in certain situations than the other, neither concept makes the other redundant.

Conditioned Synchronisation

The passive monitor construct in its basic form does not provide any means of conditioned synchronisation, which has necessitated the introduction of condition variables. Both a monitor and its condition variables need very involved queue handling facilities to deal with suspended processes (cf. section 4.3).

Conditioned synchronisation in the active synchroniser concept can be achieved by the placing of the accept statements, either sequentially or within conditional constructs. If asynchronous communication is required then conditioned synchronisation can be achieved by means of guard conditions in the select statements. As can be seen in the implementation aspects for the synchroniser concept (cf. section 5.2), the queue handling facilities for dealing with rendezvous are fairly straight forward.

Avoiding deadlock

The potential for deadlock exists (through incorrect usage) with both the monitor and the synchroniser concepts, although with the latter this can take the form of a request for rendezvous not forthcoming to a corresponding accept statement (or vice-versa), which is slightly more obvious than those situations where deadlock can occur with the monitor.

Apart from the obvious cases associated with monitors, (such as a missing qsignal operation for a corresponding qwait etc.), condition variables have a further subtle problem associated with their usage in that, unless there is advance knowledge that a qwait operation will be performed before the corresponding qsignal operation, associated Boolean expressions will be necessary to prevent a qsignal operation (which is not "remembered") from "missing" the subsequent qwait operation and so causing deadlock.

Implications due to the nature of the constructs

The use of synchronisers, being active processes, can lead to limitations on other processes in the concurrent system, which are not prevalent with the use of monitors. This is because the two constructs have different scheduling implications. The synchroniser is executed as a separate entity, whereas the monitor is executed on behalf of the calling process.

Each synchroniser launched means one more active process in the system. Depending on the stack allocation algorithm for the processes, this typically means that less stack space will available for use by each process than in a similar system making use of the monitor concept. If the number of processes allowable in a system is limited (as in CLANG), then each synchroniser will count against this limit while monitors do not.

On a single processor using a cyclic scheduling scheme, (as in CLANG), the use of synchronisers will result, on average, in more context switching than there would be in a similar system making use of monitors. This will mean that the average time lapse between a process gaining use of the processor will be higher for a system using synchronisers. Also in large systems where backing storage is required, this process switching may be "expensive" in terms of the time wasted in the rolling in and out of processes from backing storage.

The active nature of the synchroniser has further consequences as the transfer of parameters between two active processes (the synchroniser and the "client" process) during a rendezvous involves the different stack sections of each process (this has necessitated the introduction of the LDE P-code cf. chapter 5 note (4) relating to figure I) while the monitor procedures or functions can be considered part of the calling process and therefore their local variables are accomodated only in the stack area of the calling process. The monitor variables are effectively global and thus contained in the stack portion for the main program (which is inactive during the concurrent execution of the processes).

The passive nature of the monitor concept makes it possible to call a monitor procedure / function from within a synchroniser, but an entry point request may not be made from within a monitor.

Multiple instances

Another facet associated with the synchroniser which may sometimes be construed as an advantage is that it is possible to launch multiple instances of the same synchroniser from within a single Cobegin..Coend construct.

eg.
```
synchroniser WAREHOUSE(SIZE);
 begin
  ...
 end;

 ...

begin   (*CLASSICEXAMPLE*)
 cobegin
  WAREHOUSE(1); (* The parameters may allow the *)
  WAREHOUSE(2); (* user to specify the  size of *)
                (* the warehouse.               *)
  PRODUCER;
  CONSUMER
 coend
end.    (*CLASSICEXAMPLE*)
```

Although the synchronisers will be distinct, and seemingly distinguishable to the programmer by means of parameters (something which is not possible with monitors), exactly which synchroniser will deal with a request for rendezvous may not be obvious to the programmer,

eg. WAREHOUSE.DEPOSIT(ITEM)

as the parameters to a synchroniser are not specified when a request for rendezvous is made.

In fact it is the synchroniser which reaches the corresponding accept statement first which will perform the rendezvous. This reduction in delay time may result in an increase in parallelism, but additional care will have to be taken to ensure the absence of deadlock.

Multiple instances of the same monitor are not permitted.

Favourable situations

In situations involving no contention, access to a monitor is similar to a simple procedure / function call, while a request for rendezvous still results in the requesting process being suspended (involving a process switch), until the rendezvous has been performed by the synchroniser.

On the other hand, in situations involving contention, a process calling a monitor procedure or function might be queued awaiting exclusivity to the monitor and once this has been obtained may also be queued "inside" the monitor on a condition variable. Even once the reasons for suspension have been satisfied the process may still be delayed further, as it endeavours to recover all those exclusivities released on suspension, before it may finally proceed. The rendezvous request mechanism requires that the requesting process be suspended once, and remain suspended until its request has been serviced, whereafter it will be free to proceed.

Flexibility of the constructs

The use of the select statement within the synchroniser permits non-deterministic selection of which rendezvous request the synchroniser wishes to service. This is not possible with regard to the monitor, the choice being determined by the order in which the calling processes are queued awaiting exclusivity.

The use of entry points, and the corresponding request and accept operations on them, permits flexibility within the synchroniser, as it is possible to have several accept statements per entry point, allowing different actions to be taken each time the corresponding request is made. This can be simulated within a monitor by means of additional parameters to the relevant procedures and then using these parameters in conjunction with if...then...else constructs to achieve the desired results - not altogether satisfactory.

Availability of local variables

Monitor variables (and constants) do have one advantage over those of synchronisers in that, should they be declared as starred identifiers, they are accessible outside the monitor block though only in a "read only" capacity. This allows processes to inspect the values of monitor variables without actually having to enter the monitor, a facility not permissable with a synchroniser's variables. However, monitor variables can run foul of the invariance problem, a factor to which sychroniser variables are not subject.

Finite system problem

Another problem relating to the use of the synchronisers which does not apply to the monitor, is what may be termed the "finite system problem". This problem comes about in a system where the concurrent execution of the processes only last a finite length of time before they terminate, whereafter the main program is reactivated and continues execution. This is particularly true in a teaching environment where it is desirable to demonstrate the effect of only a limited number of requests to a particular entry point.

For example, to study the effect of just three deposits to the warehouse synchroniser, the producer process may be coded as:

```
procedure PRODUCER;
 const SWEET = 1;
 var ITEM, NUMBER;
  begin
   for NUMBER := 1 to 3 do
    begin
     ITEM := SWEET; (*produce item*)
     WAREHOUSE.DEPOSIT(ITEM)
    end    (*for*)
  end;    (*PRODUCER*)
```

In order to avoid deadlock the number of requests for rendezvous by a "client" process must match the number of corresponding accept statements in the synchroniser concerned. The onus is on the programmer to ensure this. The problem may further be complicated by having multiple instances of the synchroniser or "client" processes, or by having the rendezvous request within some form of conditional construct.

For example:

Given the following section of code in the PRODUCER process:

```
if DAY=MONDAY then
 for NUMBER := 1 to 5 do
  begin
   ITEM := SWEET;    (*produce item*)
   WAREHOUSE.DEPOSIT(ITEM)
  end
else
 for NUMBER := 1 to 3 do
  begin
   ITEM := SWEET;    (*produce item*)
   WAREHOUSE.DEPOSIT(ITEM)
  end;
```

where DAY and MONDAY are declared local to PRODUCER, it would not be possible for the programmer to calculate the values of the matching loop for the corresponding accept statements in the WAREHOUSE synchroniser unless prior knowledge is available as to whether DAY = MONDAY or not.

Note: These problems will not occur in a infinite system or if the system makes use of the passive construct of the monitor.

In order to accomodate the finite system problem it has been necessary to implement additional constructs to be used within the synchroniser to control the execution of the accept statements. These operations include ACTIVEINSYSTEM, READYINSYSTEM, STOPCONCURRENCY and SWITCH (cf. chapter 4 of appendix A: The User Manual).

Note: Similar constructs are available in Ada for the same purpose, and include the DELAY statement and the operation TERMINATE.

Conclusion

Of all the languages assessed, apart from CLANG, only the language CHILL supports both types of concepts with its regions, buffers etc. (cf. chapter 5 section 5.1.1), but the blurred boundaries separating them has resulted in a cluttered language which can only confuse the programmer.

In a language supporting both concepts it is necessary to define clear boundaries between them, and for their definitions to be syntactically and semantically distinct. The concepts in CLANG adhere to this.

CLANG supports both constructs, because as a possible teaching language, with most of the concurrent languages "available" supporting one or the other, it is necessary for a student to have an understanding of both.

As can be seen in the above discussion the rendezvous concept undoubtably overcomes some of the problems associated with the monitor concept but still in certain situations, such as one involving no contention, the use of the monitor concept is more suitable.

It is the conclusion of this report that (in answer to the opening quote by Eventoff et al.) until a concept is forthcoming to replace those of both the monitor and synchroniser, the availability of both in a language will give a programmer a greater flexibility and allow the choice of concept to suite the situation - the increase in system performance will follow.

6.1 Other concepts proposed - a brief summary

Since the inception of the monitor concept in 1974 [Hoa74], numerous people [Cam74], [Ger77], [Kie83], [Ree79], have proposed modifications.

One criticism levelled at the monitor concept [Cam 74] is that synchronisation of monitor operations is realised by code scattered throughout the monitor, with some of this code, such as the operations on condition variables, being visible to the programmer, while other code, such as that ensuring the mutually exclusive access of the monitor, is not.

One of the most innovative solutions to this problem has been that of the Path expression [Cam74].

Path expressions are a synchronisation mechanism which enables a programmer to specify in one place, in each of those modules which will be subject to concurrent access, all constraints on the execution of operations defined by that module. The implementation of the operations is separated from the specification of the constraints, with the code for enforcing these constraints being generated by the compiler.

One programming language that incorporates path expressions is Path Pascal [Cam80]. In Path Pascal a module, using path expressions to "protect" a resource, has a structure like that of a monitor. Path expressions in the header of each module define constraints on the order in which the relevant operations on the resource will be performed. There is no code for expressing synchronisation within the procedures encapsulated within the module. Thus a path expression defines all legal sequences of operations performed on a resource [And83].

However, whether or not an operation may be performed on a resource may also depend on parameters to the operation and/or state information in a way not directly related to the history of operations already performed, and it is here that path expressions flounder. In order to express this conditioned synchronisation additional mechanisms must be introduced, but according to Andrews and Schneider [And83],

> "Regrettably, none of these extensions have solved the entire problem in a way consistent with the elegance and simplicity of the original proposals"

In an endeavour to overcome the shortcomings he perceived in the way the monitor and path expression concepts handled the problem of conditioned synchronisation, Gerber [Ger77] introduced the notation of (integer) counters which are incorporated into the definitions of data objects shared by several asynchronous processes.

This theory of counter variables is based on the belief that the specification of the synchronisation of the shared data object should not be included as part of the procedures which perform the required operations on the data objects, but rather, the synchronisation should take place before the desired procedure is entered.

This is achieved by the evaluation of a "when condition" (which is equivalent to a Boolean expression on the counter variable) prior to the execution of the procedure. If this "when condition" evaluates to true then the execution of the procedure may proceed and an implicit incrementing and/or decrementing occurs of a specified subset of the counters in the module in which the procedure was declared. If the evaluation returns false then the process attempting to call the procedure concerned is suspended until, at procedure exit by another process, an implicit "signal" operation reactivates it.

These "when conditions" of Gerber are a variation on the conditional critical region originally proposed by Hoare [Hoa72], and Brinch Hansen [Bri72], [Bri73]. Conditional critical regions provide a structured notation for specifying synchronisation where shared variables are explicitly placed into "resources" with each shared variable in at most one resource and only accessed in conditional critical region statements. Mutual exclusion is provided by guaranteeing that the execution of different conditional critical region statements which name the same resource, are not interleaved in time. Conditioned synchronisation is provided by explicit Boolean conditions in these statements. The major drawback of the conditional critical region is in their implementation, as the conditions within them can contain references to local variables. This means that each process must evaluate its own conditions, which is "expensive" as a process must be reactivated to check a condition which might still be false. Condition critical region statements provide the synchronisation mechanism in the programming language Edison (cf. section 4.2.2).

Path expressions and counter variables are just two of the alterations to monitors proposed. (The rendezvous, being a relatively new concept [Hoa78], has yet to spawn various extensions and subtle alterations).

A few other proposals include:

(1) Access-Right expressions, [Kie83], are a form of protocol specification, similar to that of a rendezvous, but between a passive data structure and the active process wishing to access it.

(2) Eventcounts and Sequences have been proposed, [Ree79], as abstract objects that allow processes, rather than using mutual exclusion to protect the manipulations of shared variables which control the ordering of events, to control the ordering of events directly. The event count is a communication path for signalling and observing the progress of concurrent computation while the sequencer assigns an order to the events occuring in the system.

6.2 Where do we go from here ?

> "All too often people think they have found the ultimate solution and give up searching, when in reality the ultimate solution may have eluded them."
>
> The author during a moment of quiet reflection

Approximately five years separate each of the major milestones in the development of methods of expressing interprocess synchronisation and communication; the semaphore [Dij68]; the monitor [Bri72], [Bri73] and [Hoa74]; and the rendezvous [Hoa 78] and [Bri78].

If this trend were to have continued a new method would have been due out in 1983 or 1984; as yet none has been forthcoming.

That each of the developments has been an improvement on what was before there can be no doubt, but as to whether the successive developments can be regarded as replacing the existing one is another question.

For example, the rendezvous technique goes a long way to solving many of the problems associated with the monitor concept, but has in turn introduced its own problem areas, which although maybe not as severe, still hamper the prospects of the rendezvous concept replacing that of the monitor.

The flow of development has been towards relieving the programmer of the burden of explicitly controlling the "simultaneous" alteration of the data structures shared by several process and also towards a more "natural" way of expressing synchronisation and communication.

Perhaps the best way of extending this idea and possibly achieving the "ultimate" solution, would be to examine further the "natural" way in which animals and human beings synchronise and communicate and then extend these observations into a model for concurrent process synchronisation and communication. After all we human beings are very adept at concurrent activities.

One possibility that springs to mind is that of a Professor-Student model.

A student wishing to discover a solution to a problem will go in search of a professor, possibly interrupting the professor's own train of thought, and together they will solve the problem. This could involve a scan of the professor's brain (<u>ie.</u> variables or, if the professor is fixed in his ways - constants), or both processes going off to a library to find out what is required.

The difference between this model and the rendezvous is the actual looking for, and possible interruption of the looked for process. The interruption could take the form of a flag in the professor process' descriptor table which, when the professor is about to be scheduled, would indicate the presence of the interruption and allow the professor to take the appropriate action. This is different from the rendezvous model where the "interruptions" take place at predetermined locations specified by the accept statements.

The nature of the interruption would be specified by the student process which could result in a "jump" to the correct position, possibly an explicitly declared procedure, in the professor's code to deal with this request.

Non-deterministic selection would be implicit by the arbitrary nature of the interrupts and here another improvement over the rendezvous model would be the specifying of a priority associated with the interruption - a professor only being interrupted by a student of a high enough priority (eg. a pretty girl).

(In the rationale for the design of Ada [Ich79], the specifying of a priority for rendezvous requests was suggested, but this was dropped in the final language specifications.)

To prevent deadlock, the professor processes would not be allowed to terminate before all the student processes (the professors normally being the last to leave), although this could result in the professor's "busy waiting". This technique will overcome the finite system problem that dogs the rendezvous model.

What interruptions could be dealt with by each professor would be explicitly set up by the programmer and if several professors could deal with one type of request, this request could be put in a common location ("library") to be accessed in mutual exclusion.

The professor-student model would consist of synchronisation and communication between two active processes, with possibly the passive construct of the "library". This is line with the more "natural" approach sought by Hoare and Brinch Hansen.

BIBLIOGRAPHY

Bibliography

[And83] Andrews, G.R. and Schneider, F.B. "Concepts and Notations for Concurrent Programming", ACM Computing Surveys, Vol. 15, No. 1, pgs. 3-43, March 1983.

[Bar80] Barnes, J.G.P. "An overview of Ada", Software - Practice and Experience, Vol. 10, No. 11, pgs. 851-887 November 1980.

[Ben82] Ben-Ari, M. "Principles of Concurrent Programming", Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1982.

[Bea78] Beaumont, W.P. "An Implementation of Structured Multiprogramming", Software - Practice and Experience, Vol. 8, No. 3, pgs. 313-322, May-June 1978.

[Bod83] Boddy, D.E. "Implementing Data Abstraction and Monitors in UCSD Pascal", ACM SIGPLAN Notices, Vol. 18, No. 5, pgs. 15-23, May 1983.

[Bod84] Boddy, D.E. "On the Design of Monitors with Priority Conditions", ACM SIGPLAN Notices, Vol. 19, No. 2, pgs. 38-46, February 1984.

[Bra82] Branquart, P., Louis, G. and Woden, P. "An Analytical Description of CHILL, the CCITT High Level Language", Lecture Notes in Computer Science 128, edited by G.Goos and J.Hartmanis, Springer-Verlag, Berlin-Heidelberg-New York, 1982.

[Bri72] Brinch Hansen, P. "Structured Multiprogramming", Communications of the ACM, Vol. 15, No. 7, pgs. 574-578, July 1972.

[Bri72] Brinch Hansen, P. "A Comparison of Two Synchronising Concepts", Acta Informatica Vol. 1, Springer-Verlag, Berlin-Heidelberg-New York, pgs. 190-199, 1972.

[Bri73] Brinch Hansen, P. "Operating System Principles", Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1973.

[Bri75] Brinch Hansen, P. "The Programming Language Concurrent Pascal", IEEE Transactions on Software Engineering, Vol. SE-1, No. 2, pgs. 199-207, June 1975.

[Bri76] Brinch Hansen, P. "The Solo Operating System: A Concurrent Pascal Program", "The Solo Operating System: Job Interface" and "The Solo Operating System: Processes, Monitors and Classes", Software - Practice and Experience, Vol. 6, No. 2, pgs. 141-200, April-June 1976.

[Bri77] Brinch Hansen, P. "The Architecture Of Concurrent Programs", Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1977.

[Bri78] Brinch Hansen, P. "Distributed Processes: A Concurrent Programming Concept", Communications of the ACM, Vol. 21, No. 11, pgs. 934-941, November 1978.

[Bri81] Brinch Hansen, P. "Guest Editorial: Introducing the Edison Papers", "Edison - A Multiprocessor Language", "The Design of Edison" and "Edison Programs", Software - Practice and Experience, Vol. 11, No. 4, pgs. 323-414, April 1981. (See also "Programming a Personel Computer", Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1982.)

[Bus80] Bustard, D.W. "An Introduction to Pascal Plus", Chapter 1, pgs. 1-57, in "On the Construction of Programs" edited by R.M. McKeag and A.M. Macnaghten, Cambridge University Press, Cambridge, 1980.

[Bus80] Bustard, D.W. "A User manual for Pascal Plus - Version 1d", Dept. of Computer Science, The Queen's University of Belfast, July 1980.

[Bus82] Bustard, D.W. "Pascal Plus Tutorial Guide: Draft 2.1", Dept. of Computer Science, The Queen's University of Belfast, 1982.

[Cam74] Campbell, R.H. and Habermann, A.N. "The Specification of Process Synchronisation by Path Expressions", Lecture Notes in Computer Science 16, Springer-Verlag, New York, 1974.

[Cam80] Campbell, R.H. and Kolstad, R.B. "An Overview of PATH PASCAL'S Design" and "PATH PASCAL User Manual", ACM SIGPLAN Notices, Vol. 15, No. 9, pgs. 13-24, September 1980.

[Cha82] Chalmers, A.G. "Pascal-S Mark1.HAC Compilers", B.Sc.(Hons) project, Dept. Computer Science, Rhodes University, South Africa, 1982.

[Cha83] Chalmers, A.G. "Concurrent Features in CLANG", Symposium and Workshop on Computer Science Theory and Practice, Rhodes University, South Africa, November 1983.

[Col79] Coleman, D., Gallimore, R.M., Hughes, J.W. and Powell, M.S. "An Assessment of Concurrent Pascal", Software - Practice and Experience, Vol. 9, No. 10, pgs. 827-837, October 1979.

[Col80] Coleman, D. "Concurrent Pascal - an appraisal", Chapter 6, pgs. 213-227 in "On the Construction of Programs" edited by R. McKeag and A.Macnaghten, Cambridge University Press, Cambridge, 1980.

[Dij68] Dijkstra, E.W. "Cooperating Sequential Processes", in "Programming Languages", edited by F. Genuys, Academic Press, New York, 1968.

[Dij75] Dijkstra, E.W. "Guarded commands, nondeterminacy, and formal derivation of programs", Communications of the ACM, Vol. 18, No. 8, pgs. 453-457, August 1975.

[Eve80] Eventoff, W., Harvey, D. and Price, R.J. "The Rendezvous and Monitor Concepts: Is there an Efficiency Difference ?", ACM SIGPLAN Notices, Vol. 15, No. 11, pgs. 156-165, November 1980.

[Fid83] Fidge, C.J. and Pascoe, R.S.V. "A Comparison of the Concurrency Constructs and Module Facilities of CHILL and Ada", The Australian Computer Journal, Vol. 15, No. 1, pgs. 17-27, February 1983.

[Ger77] Gerber, A.J. "Process Synchronisation by Counter Variables", ACM Operating Systems Review, Vol. 11, No. 4, pgs. 6-17, October 1977.

[Gre82] Greiter, G. "Remarks on Language Concepts for specifying Process Synchronisation", ACM SIGPLAN Notices, Vol. 17, No. 9, pgs. 58-61, September 1982.

[Har77] Hartmann, A.G. "A Concurrent Pascal Compiler for Mini-Computers", Lecture Notes in Computer Science 50, edited by G. Goos and J. Hartmanis, Springer-Verlag, Berlin-Heidelberg-New York, 1977.

[Had77] Haddon, B.K. "Nested Monitor Calls", ACM Operating Systems Review, Vol. 11, No. 4, pgs. 18-23, October 1977.

[Hoa72] Hoare, C.A.R. "Towards a theory of parallel programming", in "Operating Systems Techniques", edited by C.A.R. Hoare and R.H. Perrott, Academic Press, New York, 1972.

[Hoa74] Hoare, C.A.R. "Monitors: An Operating System Structuring Concept", Communications of the ACM, Vol. 17, No. 10, pgs. 549-557, October 1974.

[Hoa78] Hoare, C.A.R. "Communicating Sequential Processes", Communications of the ACM, Vol. 21, No. 8, pgs. 666-677, August 1978.

[How76] Howard, J.H. "Proving Monitors", Communications of the ACM, Vol. 19, No. 5, pgs. 273-279, May 1976.

[Ich79] Ichbiah, J.D., Heliard, J.C, Roubine, O., Barnes, J.G.P., Krieg-Brueckner, B. and Wickmann, B.A. "Rationale for the Design of the Ada Programming Language", ACM SIGPLAN Notices, Vol. 14, No. 6, part B, June 1979.

[Kau76] Kaubisch, W.H., Perrott, R.H. and Hoare, C.A.R. "Quasiparallel Programming", Software - Practice and Experience, Vol. 6, No. 3, pgs. 341-356, July-September 1976.

[Kee78] Keedy, J.L. "On Structuring Operating Systems with Monitors", The Australian Computer Journal, Vol. 10, No. 1, pgs. 5-9, February 1978.

[Kie83] Kieburtz, R.B. and Silberschatz, A. "Access-Right Expressions" ACM Transactions on Programming Languages and Systems, Vol. 5, No. 1, pgs. 78-96, January 1983.

[Lis76] Lister, A. and Maynard, K.J. "An Implementation of Monitors", Software - Practice and Experience, Vol. 6, No. 3, pgs. 377-385, July-September 1976.

[Lis77] Lister, A. "The Problem of Nested Monitor Calls", ACM Operating Systems Review, Vol. 11, No. 3, pgs. 5-7, July 1977.

[McC83] McCormack, J. and Gleaves, R. "Modula-2: A Worthy Successor to Pascal", BYTE, Vol. 8, No. 4, pgs. 38-48, April 1983.

[Neh79] Nehmer, J. "The Implementation of Concurrency for a PL/I-like Language", Software-Practice and Experience, Vol. 9, No. 12, pgs. 1043-1057, December 1979.

[Par78] Parnas, D.L. "The non-problem of Nested Monitor Calls", ACM Operating Systems Review, Vol. 12, No. 1, pgs. 12-14, January 1978.

[Ree79] Reed, D.P. "Synchronisation with Eventcounts and Sequencers", Communications of the ACM, Vol. 22, No. 2, pgs. 115-123, February 1979.

[Sew84] Sewry, D.A. "Concurrency in Modula-2", M.Sc. Thesis, Dept. of Computer Science, Rhodes University, South Africa, 1984.

[Sto82] Stotts Jr, P.D. "A Comparative Survey of Concurrent Programming Languages", ACM SIGPLAN Notices, Vol. 17, No. 10, pgs. 50-61, October 1982.

[Sum80] Sumpter, A.G. and Quick, G.E. "Concurrency Specification in High Level Languages", ACM SIGPLAN Notices, Vol. 15, No. 12, pgs. 75-81, December 1980.

[Ter83] Terry, P.D. and Chalmers, A.G. "CLANG - A Concurrent Language", Symposium and Workshop on Computer Theory and Practice, Rhodes University, South Africa, November 1983.

[Tho78] Thorelli, E. "A Monitor for Small Computers", Software - Practice and Experience, Vol. 8, No. 4, pgs. 439-450, July-August 1978.

[Uni81] United States Department of Defense "The programming language Ada", Lecture Notes in Computer Science 106, Springer-Verlag, Berlin-Heidelberg-New York, 1981.

[Vol83] Manual for Modula-2 on the Sage IV, Volition Systems, 1983.

[Wel79] Welsh, J. and Bustard, D.W. "Pascal-Plus - another Language for Modular Multiprogramming", Software - Practice and Experience, Vol. 9, No. 11, pgs. 947-957, November 1979.

[Wel80] Welsh, J. and McKeag, M. "Structured System Programming", Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1980.

[Wel81] Welsh, J. and Lister, A. "A comparative study of task communication in Ada", Software - Practice and Experience, Vol. 11, No. 3, pgs. 257-290, March 1981.

[Wet78] Wettstein, H. "The problem of Nested Monitor Calls revisited", ACM Operating Systems Review, Vol. 12, No. 1, pgs. 19-23, January 1978.

[Wir75] Wirth, N. "Pascal-S: A subset and its implementation", Berichte Nr. 12, Institut fur Informatik, Eidgenossische Technische Hochschule, Zurich, Switzerland, 1975. Also in "Pascal - The Language and its Implementation", edited by D.W. Barron, John Wiley, Chichester, England, 1981.

[Wir76] Wirth, N. "Algorithms + Data Structures = Programs", Prentice-Hall Inc. Englewood Cliffs, New Jersey, 1976.

[Wir83] Wirth, N. "Programming in Modula-2 and Report on the Programming Language Modula-2", Springer-Verlag, Berlin-Heidelberg-New York, 1983.

[You82] Young, S.J. "Real Time Languages - design and development", Ellis Horwood, Chichester, England, 1982.

[You83] Young, S.J. "An Introduction to Ada", Ellis Horwood, Chichester, England, 1983.

APPENDIX A

USER MANUAL

Table of contents

Introduction

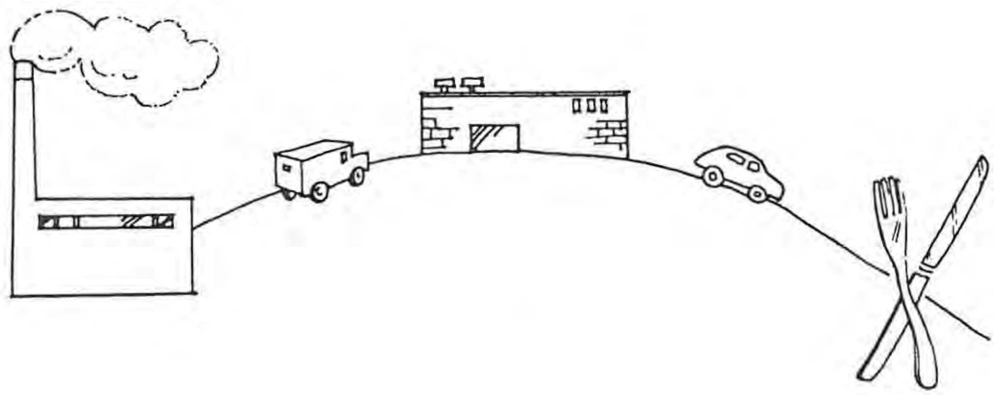
> "A sequential program specifies sequential execution of a list of statements; its execution is called a process. A concurrent program specifies two or more sequential programs that may be executed concurrently as parallel processes." [And83]

In CLANG a concurrent program is executed by allowing processes to share one processor.

In order to cooperate, concurrently executing processes must communicate and synchronise.

Communication allows the execution of one process to influence the execution of another. Because these processes are executed at unpredictable speeds, synchronisation is often necessary when processes communicate. One can view synchronisation as a set of constraints on the ordering of events. Thus a process sometimes is delayed so that a sequence of events may occur in a desired order.

To illustrate the need for communication and synchronisation between concurrent processes, consider this example:



A common event in our daily lives is that of a producer who produces an item and delivers it to a warehouse, from where a consumer acquires the item and does with it what consumers do best - consumes it.

The producer and the consumer can be represented by two concurrent processes.

```
procedure PRODUCER;
 begin
  while in business do
   produce ITEM
   deposit ITEM in warehouse
 end
```

```
procedure CONSUMER;
 begin
  while desire lasts do
   remove ITEM from warehouse
   consume ITEM
 end
```

Note: The actions of actually producing the ITEM and actually consuming the ITEM are totally independent of each other. However both the producer and the consumer have a need to access the warehouse: the producer to deposit the ITEM and the consumer to remove the ITEM.

If we assume that only one ITEM at a time may be in the warehouse and bearing in mind the independent speeds at which each of the two processes operate, it can be seen that there will come a time at which one of the processes will have to be delayed, waiting for the other. The consumer may have to wait for the producer to deposit the item before he can remove it, or the producer may have to wait for the consumer to remove the item before he can deposit the next one.

Thus these two processes communicate in that the item passes from the producer via the warehouse to the consumer, and as just shown they synchronise.

As well as supporting the low level synchronisation primitive, the semaphore, CLANG supports two distinct high level constructs for concurrent process communication and synchronisation - the MONITOR and the SYNCHRONISER.

The difference between the two is the manner in which interprocess communication is performed.

The **monitor** concept is based on communication via passive abstract data structures which are accessed in mutual exclusion.

The **synchroniser** concept is based on direct synchronised transfer of messages (ie. parameters) between two active processes, one of which is the synchroniser itself.

By having both types of high level constructs CLANG is able to support a wide spectrum of concurrent applications.

The following two chapters provide a description and the general form of each of the two concepts and their associated structures and components.

The third chapter contains descriptions of four useful features available for use in conjunction with the monitor and synchroniser concepts.

Examples of usage of each of the components will be found at the end of their respective subsections.

Example programs illustrating the appropriate concept will be given in their entirety, together with results, at the end of the chapters.

The last chapter contains the list of error messages that can occur when there is incorrect usage of any of the features described in the first three chapters and an explanation of what the error message implies and an example of how it might appear.

Chapter 1: Monitors

A monitor is a construct used local to a program. It is formed by encapsulating data structures, which may be shared by concurrent processes, with a set of procedures / functions which access that data.

A special property of monitors is that only one concurrent process may be active "in" a monitor executing its procedures / functions at any given time.

Monitors thus provide a passive high level construct for implementing communication between concurrent processes via mutually exclusive access to the shared data structures.

The general form of a monitor is:

```
monitor MNAME;
 const declarations   including starred identifiers (1.1)
 var declarations
 condition declarations    (1.3)

  procedure / function declarations                  (1.1)

  begin
    body of monitor        (1.2)
  end;
```

Use of Monitors

(a) A monitor must be declared at the outer level of a program after the global variable declarations. The monitor declarations may be interspersed with the program's procedure / function and synchroniser declarations.

Note: Monitors may not be declared local to procedures, functions or synchronisers, nor may they be declared local to another monitor. ie. Monitor declarations may not be nested.

(b) The monitor identifier (MNAME) is significant to eight characters and must be unique.

(c) There are no parameters to a monitor.

(d) The current implementation restricts the number of monitors that may be declared per program to 15.

Examples of monitors

The program segments shown in the following examples are reproduced as part of entire working programs at the end of this chapter.

The warehouse mentioned in the introduction to the user manual may be coded as a monitor as follows:

```
monitor WAREHOUSE;
 const FULL=1;
       EMPTY=0;
 var SHOP, SPACE;
 condition AVAILABLE, FREE;        (1.3)

  procedure *DEPOSIT(ITEM);        (1.1)
   begin
    if SPACE = FULL then
      FREE.qwait;                  (1.3)
    SHOP := ITEM;
    SPACE := FULL;
    AVAILABLE.qsignal              (1.3)
   end;

  procedure *REMOVE(var ITEM);     (1.1)
   begin
    if SPACE = EMPTY then
      AVAILABLE.qwait;             (1.3)
    SPACE := EMPTY;
    ITEM := SHOP;
    FREE.qsignal                   (1.3)
   end;

  begin
   SPACE := EMPTY                  (1.2)
  end;
```

A monitor to provide simulation facilities in the form of pseudo-TIME might be coded as follows:

```
monitor SIMULATION;
 const *TIMELIMIT = 20; (*max length of simulation*)
 var *TIME;
 conditon ALARMCLOCK;

  procedure *HOLD(DELAY);
   (*delay caller for DELAY of simulated time*)
   var ALARM;
    begin
     if DELAY > 0 then
      begin
       ALARM := TIME + DELAY;
       ALARMCLOCK.qpwait(ALARM);
       TIME := ALARM
       (*when woken, advance pseudotime*)
      end
    end;   (*HOLD*)

   procedure *ADVANCE; (*keep waking up next job*)
    begin
     ALARMCLOCK.qsignal
    end;

   function *ENQUEUED;
   (*allow outside world to examine queue*)
    begin
     ENQUEUED := ALARMCLOCK.qlength
    end;

   begin (*SIMULATION*)
    TIME := 0   (*initial value of TIME*)
   end;  (*SIMULATION*)
```

1.1 Identifiers declared local to monitors

In this section identifiers declared local to a monitor include constants, variables, procedures and functions (but not condition variables (cf. section 1.3)).

1.1.1 Starred Indentifiers

Any identifier declared local to a monitor which has its declaration prefixed by an asterix ('*') is termed a starred identifier.

The general form is:

```
*identifier
```

An identifier may only be declared as starred at the outer level of a monitor's declarations. Starred identifiers may not be declared local to procedures or functions. Monitors themselves may not be starred.

A starred identifier is deemed to be globally accessible, subject to the normal scope rule that it must be declared before it may be referenced.

A starred identifier is referenced (using a notation similar to that used when accessing records in Pascal) by means of prefixing the name of the monitor, in which the identifier was declared, to the name of the identifier, separated by a period ('.').

The general form of accessing a starred identifier is:

```
Monitorname.identifier
```

Only starred identifiers may be accessed in this way.

Inside the monitor in which they were declared, starred identifiers may be referred to either by prefixing them with the monitor name or not. As in this case the prefixing is not really neccessary it is perhaps better practice to leave it out.

For example, given the following declaration

```
monitor MON;
 var *IDENT;
```

the starred variable, IDENT, may be referred to in the monitor, MON, by either

```
MON.IDENT    or simply    IDENT
```

When accessing a starred identifier from outside the monitor in which it was declared the prefixing must be used.

1.1.1.1 Starred procedures / functions

Starred Monitor procedures or functions may have parameters, both value and variable, associated with them, subject to the current implementation limit of 25. (This also applies to non-starred procedures / functions declared local to monitors.)

Indeed it is by means of these parameters that communication between the concurrent processes is established.

1.1.1.2 Starred variables and accessing variables from outside a monitor

The values of a monitor's variables, both starred and unstarred, are retained between activations of monitor procedures / functions. This means that a monitor's variables are effectively at the global level although, the scope of access is determined by their point of declaration and whether they are starred identifiers or not.

Starred monitor variables may be accessed from outside the monitor in which they were declared by the normal method of prefixing; however these monitor variables may only be accessed in a "**read only**" capacity which implies the value of the starred monitor variables may not be altered, by any means, outside the monitor in which they were declared.

The values of monitor variables, both starred and unstarred, may be altered within the monitor in which they were declared.

The program's global variables are within the scope of the monitors and so may be accessed from within the monitors, but only in a "**read only**" capacity. ie. The values of global variables may be examined, but not altered, within a monitor. Thus the body of a monitor may not be used to assign initial values to any variables declared globally in the program. (cf. section 1.2)

To help clarify, consider the following example program. In this program both valid and invalid usages of variables are demonstrated and marked accordingly.

```
program DEMONSTRATION;
 var G1, G2;    (*program's global variables*)

  monitor MON1;
   var MV1A, *MV1B;    (*monitor variables*)

    procedure *M1PROC;
     begin
      MV1A := G1         VALID
     end;

   begin    (*MON1*)
    MV1A := Ø;           VALID
    G1 := Ø;             INVALID - global variables
                                   read only
    MV1B := Ø;           VALID
    MON1.MV1B := Ø;      VALID
    read(G1)             INVALID - may not alter value
                                   of global variables
   end;    (*MON1*)

  monitor MON2;
   var *MV2A, MV2B;

   begin    (*MON2*)
    MV2B := MON1.MV1B;  VALID
    MV2B := MON1.MV1A   INVALID - MV1A is not a starred
                                  variable
   end;    (*MON2*)

 begin   (*body of program DEMONSTRATION*)
  G1 := MON2.MV2A;       VALID
  read(MON2.MV2A);       INVALID - may not alter the
                                   value of MV2A
  G2 := MON2.MV2A * 2 * MON1.MV1B     VALID
 end.    (*DEMONSTRATION*)
```

<u>Further examples of accessing starred identifiers</u>

To show how starred procedures are called consider how the two processes of the producer and the consumer may be coded to access the warehouse developed as a monitor. (cf.section 1)

```
procedure PRODUCER;
 const SWEET = 1;
 var ITEM;
  begin
   while BUSINESS = GOOD do
    begin
     ITEM := SWEET;    (*produces item*)
     WAREHOUSE.DEPOSIT(ITEM)
    end    (*while*)
  end; (*PRODUCER*)


procedure CONSUMER;
 var ITEM, MOUTH;
  begin
   while DESIRE = GOOD do
    begin
     WAREHOUSE.REMOVE(ITEM);
     MOUTH := ITEM     (*consume item*)
    end    (*while*)
  end;   (*CONSUMER*)
```

In conjunction with the monitor SIMULATION which provides the simulation facilities for pseudo-time we have two processes, TICK and TOCK, which actually operate the "clock".

```
procedure TICK; (* keeps the clock going to
                   wake up jobs when complete*)
 begin
  while (SIMULATION.TIME < SIMULATION.TIMELIMIT)
                                       or
        (SIMULATION.ENQUEUED > 0 ) do
   begin
    if readyinsystem = 1 then (*cf. chapter 3*)
      SIMULATION.ADVANCE;
   end;   (*while*)
 end;   (*TICK*)


procedure TOCK; (*record the passage of time*)
 begin
  while SIMULATION.TIME < SIMULATION.TIMELIMIT do
   begin
    SIMULATION.HOLD(1);
    writeln(SIMULATION.TIME, ' seconds');
   end;
 end;   (*TOCK*)
```

1.2 The body of a monitor

The body of a monitor is executed before the execution of the body of the main program.

The general form is:

```
begin
  statements
end;
```

If there is more than one monitor declared in a program then the body of the first monitor declared is executed first, followed by the body of the second monitor declared and so on until the body of the final monitor declared is executed, then the body of the main program starts to execute.

Diagramatically the flow of execution is:

```
                         program DIAGRAM;
                            .
                            .

                           monitor M1;
                              .
                              .
Start of execution -->      begin
                              body of M1
                            end;
                            .
                            .

                           monitor M2;
                              .
                              .
                            begin
                              body of M2
                            end;
                            .
                            .
                            .

                           monitor Mn;
                              .
                              .
                            begin
                              body of Mn
                            end;
                            .
                            .

                           begin
                             body of main program
End of execution -->       end.
```

Notes:

(a) The monitor declarations may be interspersed among procedure / function declarations; (hence the ... between (for example) monitor M1 and monitor M2).

(b) Obviously the flow of execution may be temporarily sidetracked due to procedure or function calls and the launching of concurrent processes.

Thus the body of a monitor may be used to give initial values to the monitor variables and to set up the data structure encompassed by the monitor before the execution of the body of the main program starts; hence the body of a monitor is sometimes referred to as the "initialisation code" of a monitor.

Example

In the monitor WAREHOUSE (cf. section 1) the body of the monitor was:

```
begin
 SPACE := EMPTY
end;
```

which ensures that the warehouse is initially empty.

1.3 Condition variables

Monitors offer a means of communication between concurrent processes; however the only synchronisation they offer is in the fact that only one process may be active in a monitor at any given time and that other processes wishing for access to the monitor are queued on a first-come-first-served basis. Thus the monitor concept has been supplemented with **condition variables** which can be used to provide a means of conditioned synchronisation within a monitor.

1.3.1 Declaration of condition variables

Condition variables may only be declared local to monitors.

Condition variables are declared after the monitor's variable declarations and before any procedure / function local to the monitor.

The general form of declaration is:

```
condition CONDVAR1, CONDVAR2[M:N], ... CONDVARn;
```

Notes:

(a) The same rules for naming of identifiers apply to condition variables.

(b) Condition variables may not be declared as starred identifiers (cf. section 1.1.1) and therefore condition variables are not accessible outside the monitor in which they were declared.

(c) In the current implementation there may only be a maximum of 25 condition variables per **program**.

(d) Arrays of condition variables may be declared, but every array element counts towards the restriction of (c) above.

Examples of declarations of condition variables

```
condition BUSY, FREE[1:4];
```

1.3.2 Operations on condition variables

Condition variables are not variables in the "true" sense, but rather implicit queues on which concurrent processes can suspend themselves, waiting for an event to occur.

There are five operations available for the manipulation of these implicit queues.

These are:

qwait, qpwait(PRIORITY), qsignal, queue, qlength

These operations are used by prefixing them with the name of the condition variable to which they apply, separated by a period ('.').

The general form is:

conditionvariablename.operation

Condition variables may only be used in conjunction with these operations.

1.3.2.1 QWAIT

The operation qwait delays a process on the implicit condition variable queue with a default priority.

Example of usage

```
FREE[4].qwait
```

1.3.2.2 QPWAIT(PRIORITY)

The operation qpwait(PRIORITY) delays the process on the implicit condition variable queue with a priority specified by the expression "(PRIORITY)". This priority must be in the range 1..MAXINT. A low priority value indicates a high priority status.

The default priority used for qwait is 10.

Thus qpwait(PRIORITY), and qwait, can be used to influence the order in which processes are queued on the condition variable queues, waiting for an event to occur.

Examples of usage

```
BUSY.qpwait(12*AVAR) - where AVAR is a variable

FREE[4].qpwait(10) -is equivalent to- FREE[4].qwait
```

1.3.2.3 QSIGNAL

The operation qsignal will reactivate the process at the head of the implicit condition variable queue, at the same time temporarily suspending the signalling process. This implies there may be more than one process "inside" a monitor, but only one of these processes will be active.

If the queue for the associated condition variable is empty the operation will have no effect.

Qsignal is used to signify that an event has occurred and thus reactivate the processes suspended by qwait or qpwait(PRIORITY). The process that executed the qsignal will be suspended until the reactivated process has left the monitor in question, and then it will proceed.

Examples of usage

```
BUSY.qsignal
FREE[4].qsignal
```

1.3.2.4 QUEUE

The operation queue is used as a function as it returns the ord(TRUE) ie. the value 1 (there is no Boolean type in CLANG) if there is at least one process on the implicit condition variable queue, ord(FALSE), ie. Ø, otherwise.

Example of usage

If there are three processes suspended on the condition variable FREE[4] then

```
I := FREE[4].queue
```

will assign the value 1 to the variable I.

1.3.2.5 QLENGTH

The operation qlength is used as a function to return the number of processes suspended on the implicit condition variable queue (ie., the "length" of the queue).

If the queue is empty the value Ø is returned.

Example of usage

If there are four processes suspended on the condition variable BUSY then

```
I := BUSY.qlength
```

will assign the value 4 to the variable I.

1.3.3 Providing conditioned synchronisation with condition variables

By themselves condition variables provide synchronisation of the concurrent processes analogous to that provided by binary semaphores. (Semaphores are not allowed in monitors.)

Condition variables reach their full potential when used in conjunction with Boolean expressions (although there is no Boolean type implemented in CLANG.) Used thus, condition variables can provide conditioned synchronisation of the concurrent processes accessing the monitor.

The general form is:

```
                   Boolean expression *
                    condvar.operation
```

* qwait, qpwait(PRIORITY), qsignal

Care must be taken, as it is the responsibility of the programmer to ensure that the use of condition variables does not lead to deadlock.

<u>Example of usage</u>

In the implementation of a warehouse by means of a monitor (cf. section 1), reproduced here with line numbers for ease of reference, conditioned synchronisation is used at two locations.

```
 1:  monitor WAREHOUSE;
 2:   const FULL = 1;
 3:         EMPTY = Ø;
 4:   var SHOP, SPACE;
 5:   condition AVAILABLE, FREE;   (*condition vars*)
 6:
 7:    procedure *DEPOSIT(ITEM);
 8:     begin
 9:      if SPACE = FULL then
10:        FREE.qwait;
11:      SHOP := ITEM;    (*deposit the item*)
12:      SPACE := FULL;
13:      AVAILABLE.qsignal
14:     end;    (*DEPOSIT*)
15:
16:    procedure *REMOVE(var ITEM);
17:     begin
18:      if SPACE = EMPTY then
19:        AVAILABLE.qwait;
20:      SPACE := EMPTY;
21:      ITEM := SHOP;    (*remove the item*)
22:      FREE.qsignal
23:     end;
24:
```

```
25:     begin   (*WAREHOUSE*)
26:       SPACE := EMPTY
27:     end;    (*WAREHOUSE*)
```

The conditioned synchronisation expression at lines 9 and 10

```
if SPACE = FULL then
  FREE.qwait
```

will delay the producer process (cf. section 1.1.1.2) from depositing his item if the warehouse is full. (The consumer has yet to remove the item.)

If SPACE <> FULL then the producer process is not delayed but goes on to execute line 11.

```
SHOP := ITEM;    (*deposits the item*)
```

If the producer is delayed, it will remain so until the comsumer process executes the corresponding qsignal (on line 22).

FREE.qsignal   ie. after the item has been removed

A similar set up is used to ensure that the consumer process does not try to remove an item until there is one available - at lines 18 and 19

```
if SPACE = EMPTY then
  AVAILABLE.qwait
```

The corresponding "go ahead" signal from the producer when an item is available is at line 13.

```
AVAILABLE.qsignal
```

Note:

When a process is suspended on a condition variable it must release exclusivity to that monitor thus allowing another process access. The ramifications of this are dealt with in section 1.4.

1.4 The invariance of monitor variables

When a process is suspended it must release all of the exclusivities to monitors that it might hold.

A concurrent process, on reacquiring those exclusivities to monitors it was forced to released before it had finished inside them, might reasonable expect most of the values of the regained monitor's variables to have the same values as when exclusivity was released. This may, however, not always be the case as, in the interim, other concurrent processes may gain access to those monitors and possibly alter the values of the variables.

This section will detail the constructs CLANG has available for the solution of this problem. For further information concerning the problem of invariance of monitor variables and the terminology used to enlarge on it, the reader is refered to chapter 4 of the assessment.

1.4.1 At a nested PLOXY point

When a concurrent process executes a nested monitor call and is blocked it must release all its held exclusivities. In CLANG, when the process reacquires all these exclusivities and may proceed, the invariance of all the appropriate monitor's variables is assured. This guaranteeing of invariance is implicit and "automatic".

1.4.1.1 The (*$B- *) compiler directive

CLANG is seen as a teaching language and as it is sometimes desirable to demonstrate the effects of not ensuring the invariance of monitor variables to students, the (*$B- *) compiler directive has been provided.

If this option is used anywhere in a user's program, each time the program starts to execute the user will be prompted as to whether he or she wishes invariance of monitor variables, when executing a nested monitor call, or not.

The general form of the prompt is:

Nested Backup?

To this the user replies "Y" for yes, or "N" for no.

This allows the same program to be run several times and the effects of the invariance of monitor variables to be studied.

Example of usage

```
(*$B- *)     (*compiler directive*)
program DEMONSTRATION;
 (*to demonstrate the effects of invariance
   of monitor variables*)

  monitor MONIT2;

   procedure *TYUP;
    var I;
     begin
      for I := 1 to 100 do
       begin (*nothing*) end
      (*makes the conditions ripe for a blocked
        nested monitor call*)
     end;

   begin   (*MONIT2*)
    writeln('Body of MONIT2')
   end;    (*MONIT2*)

  monitor MONIT1;
   var LOOP;

    procedure *A;
     begin
      LOOP := 0;
      writeln('Initially LOOP is ',LOOP);
      while LOOP < 5 do
       begin
        LOOP := LOOP + 1;
        MONIT2.TYUP;   (*nested PLOXY point*)
        writeln('The value of LOOP is ', LOOP)
       end   (*while*)
     end;    (*A*)

    procedure *B;
     begin
      LOOP := 6
     end;    (*B*)

   begin   (*MONIT1*)
    writeln('Body of MONIT1')
   end;    (*MONIT1*)


  procedure PROC1;
  (*accesses procedure A of MONIT1*)
   begin
    MONIT1.A;
    writeln('The end of PROC1')
   end;
```

```
procedure PROC2;
(*accesses procedure B of MONIT1*)
  begin
   for I := 1 to 30 do
    begin (*delay*) end;
    (*delay so PROC1 enters MONIT1 first*)
   MONIT1.B;
   writeln('End of PROC2')
  end;

procedure PROC3;
(*to block PROC1 from entering MONIT2
  immediately*)
 begin
  MONIT2.TYUP;
  writeln('End of PROC3')
 end;

begin   (*DEMONSTRATION*)
 writeln('Start of main program');
 cobegin   (*launch concurrent processes*)
   PROC1; PROC2; PROC3
 coend
end.
```

If, after the program has compiled and starts to execute, the answer to the prompt "NESTED BACKUP?" is given as "N" for no the following output results:

```
Body of MONIT2
Body of MONIT1
Start of main program
Initially LOOP is 0
End of PROC2
End of PROC3
The value of LOOP is 6
End of PROC1
```

If, however, the answer to the prompt is "Y" for yes then the desired output is produced.

```
Body of MONIT2
Body of MONIT1
Start of main program
Initially LOOP is 0
End of PROC2
End of PROC3
The value of LOOP is 1
The value of LOOP is 2
The value of LOOP is 3
The value of LOOP is 4
The value of LOOP is 5
End of PROC1
```

1.4.2 At a conditioned PLOXY point

The loss of exclusivity to monitors due to condition variables is planned by the user to provide synchronisation between concurrent processes.

In this case it is not always desirable for all the variables of a monitor to be invariant. This is catered for by providing an explicit scheme to allow the user to specify which variables need to be invariant.

1.4.2.1 SAVE(parameters) and RESTORE

SAVE(parameters) and RESTORE are explicit statements that a user can use to bracket a conditioned PLOXY point to ensure invariance of the desired monitor variables. The variables to be made invariant must be specified in the parameters of the SAVE instruction.

The general form is:

```
SAVE(variable 1, variable 2,...variable n);
 conditioned PLOXY point
RESTORE
```

Notes:

(a) SAVE(parameters) and RESTORE are an explicit bracketing pair and if either is omitted no invariance will be assured. Warning messages will appear if this is the case (cf. chapter 4)

(b) SAVE(parameters) and RESTORE may only be used inside monitors.

(c) Although there is nothing to prevent SAVE(parameters) and RESTORE from being used other than around a conditioned PLOXY point, they are redundant elsewhere and it is efficient programming to restrict their usage to such points. (SAVE (parameters) and RESTORE may of course be used around a nested PLOXY point without redundancy if the (*$B- *) option is being used (cf. section 1.4.1.1))

More about SAVE(parameters) and RESTORE

A whole array or individual array elements may be made invariant.

For example given:

```
monitor M1;
 var A[1:4];
  ...
  SAVE(A);
   ---conditioned PLOXY point---
  RESTORE
  ...
```

would ensure that every element of the array A would be invariant, whereas

```
SAVE(A[1],A[3]);
 ---conditioned PLOXY point---
RESTORE
```

would only ensure that elements 1 and 3 of array A would be invariant.

The parameters to the SAVE may only be variables declared at the outer level of that monitor in which the SAVE is used.

Aside:

(a) There is no need to "save" the global variables as they are "read only" inside the monitor and thus may not be altered.

(b) There is no need to "save" the variables declared local to the procedure in which the SAVE(parameters) is used as these local variables will be invariant due to the fact that each invocation of a procedure sets up its own "space" for the local variables and parameters.

There is no limit to the number of SAVE(parameters) that may be used before a conditioned PLOXY point.

eg. SAVE(I); SAVE(J); -is equivalent to- SAVE(I,J);
 -PLOXY point- -PLOXY point-

Only one RESTORE is needed to ensure the invariance of the variables SAVEd before the PLOXY point. Any further RESTOREs will have no effect.

There are no parameters to RESTORE; only those variables that were specified as the parameters of the SAVE will be restored.

1.4.3 Final notes and summary on invariance of monitor variables

Due to the arbitrary nature in which concurrent processes are executed in relation to each other, it is possible to execute a program without ensuring any invariance of monitor variables and still achieve the desired results.

However, without monitor variable invariance it is not possible to guarantee that the next time the program is run the desired results will again be achieved.

Summary

When a process has to release exclusivity to its held monitors as the result of a blocked nested monitor call the invariance of all monitor variables concerned is "automatically" guaranteed unless the (*$B- *) directive is used.

When a process has to release its held monitor exclusivities as the result of a qwait, qpwait(PRIORITY) or qsignal operation on a condition variable the monitor variables, of the monitor in which the operation on the condition variable takes place, specified explicitly in the parameter list of the SAVE(parameters) instruction, will be saved, as will all the monitor variables of the other monitors that the process may have acquired, and with which it is still busy, as the result of successful nested monitor calls. Following the execution of the RESTORE instruction, (once the process has reacquired all its exclusivities), these variables will be restored and will thus be guaranteed to have the same values as prior to the release of the exclusivities.

Example programs

Here then are the full working programs, including results, from which segments have been taken to illustrate various concepts.

Toy Compiler Mark 21.2C m cv s spr nb

```
  0   (*$W- *)
  0   (*$S+ *)
  0   program COMMONEVENT;
  0
  0   (****************************************************)
  0   (* This program deals with the common event in our *)
  0   (* daily lives, that of a producer who produces an *)
  0   (* item and delivers it to a warehouse, from where *)
  0   (* a consumer acquires the item and consumes it.   *)
  0   (*                                                 *)
  0   (* The  warehouse  is implemented  by  means of a  *)
  0   (* monitor and the producer and consumer by means  *)
  0   (* of two concurrent processes.                    *)
  0   (****************************************************)
  0
  0   const GOOD = 1;
  1         MAXTIME = 5;
  1
  1     monitor WAREHOUSE;
  2      const FULL = 1;
  2            EMPTY = 0;
  2      var SHOP, SPACE;
  2      condition AVAILABLE, FREE;    (*condition variables*)
  2
  2       procedure *DEPOSIT(ITEM);
  3        begin
  4         if SPACE = FULL then FREE.qwait;
 12         SHOP := ITEM;
 16         SPACE := FULL;
 19         writeln('Item has been deposited');
 47         AVAILABLE.qsignal     (*Item available for consumption*)
 49        end;    (*DEPOSIT*)
 51
 51       procedure *REMOVE(var ITEM);
 52        begin
 53         if SPACE = EMPTY then AVAILABLE.qwait;
 61         SPACE := EMPTY;
 64         ITEM := SHOP;
 69         writeln('Item has been removed');
 96         FREE.qsignal      (*space in the warehouse*)
 98        end;    (*REMOVE*)
100
100       begin    (*WAREHOUSE*)
101        SPACE := EMPTY     (*warehouse initially empty*)
102       end;    (*WAREHOUSE*)
```

```
105
105     procedure PRODUCER;
106      const SWEET = 1;    (*item that is being produced*)
106      var ITEM, BUSINESS;
106       begin
107        BUSINESS := MAXTIME;
110        while BUSINESS >= GOOD do
115         begin
115          ITEM := SWEET;    (*produce item*)
118          writeln('Item has been produced');
145          WAREHOUSE.DEPOSIT(ITEM);
149          BUSINESS := BUSINESS - 1
153         end    (*while*)
155       end;   (*PRODUCER*)
157
157      procedure CONSUMER;
158       var ITEM, MOUTH, DESIRE;
158        begin
159         DESIRE := MAXTIME;
162         while DESIRE >= GOOD do
167          begin
167           WAREHOUSE.REMOVE(ITEM);
170           MOUTH := ITEM;    (*consume item*)
174           writeln('Item has been consumed');
201           DESIRE := DESIRE - 1
205          end    (*while*)
207         end;   (*CONSUME*)
209
209  begin  (*COMMONEVENT*)
209   writeln('About to start business');
237   cobegin
237    PRODUCER;
239    CONSUMER
239   coend;
241   writeln('Business is closed for the day')
274  end.   (*COMMONEVENT*)
```

```
About to start business
 Item has been produced
 Item has been deposited
 Item has been removed
 Item has been consumed
 Item has been produced
 Item has been deposited
 Item has been removed
 Item has been produced
 Item has been consumed
 Item has been deposited
 Item has been removed
 Item has been consumed
 Item has been produced
 Item has been deposited
 Item has been produced
 Item has been removed
 Item has been consumed
 Item has been deposited
 Item has been removed
 Item has been consumed
Business is closed for the day
```

Toy Compiler Mark 21.2C m cv s spr nb

```
   0  (*$S+ *)
   0  (*$W- *)
   0  program SIMULATEUSERS;
   0
   0  (*****************************************)
   0  (* This program simulates the actions of  *)
   0  (* three users of a  multi-access system  *)
   0  (* Monitors  are  used  to  provide  the  *)
   0  (* manager  of  the  system  and also to  *)
   0  (* provide  simulation facilities in the  *)
   0  (* form of pseudo-TIME.                   *)
   0  (*****************************************)
   0
   0  monitor MANAGER;
   2   var NEXTJOB;
   2
   2    procedure *ASSIGNJOB(var ASSIGN);
   3    (*assign a job an accounting number*)
   3     begin
   4      NEXTJOB := NEXTJOB + 1;
  10      ASSIGN := NEXTJOB
  15     end;   (*ASSIGNJOB*)
  17
  17    begin   (*MANAGER*)
  18     NEXTJOB := 0
  21    end;    (*MANAGER*)
  22
  22  monitor SIMULATION;
  23   const *TIMELIMIT = 20;   (*max. length of simulation*)
  23   var *TIME;
  23   condition ALARMCLOCK;
  23
  23    procedure *HOLD(DELAY);
  24    (*delay caller for DELAY of simulated time*)
  24     var ALARM;
  24      begin
  25       if DELAY > 0 then
  29        begin
  30         ALARM := TIME + DELAY;
  37         ALARMCLOCK.qpwait(ALARM);
  41         TIME := ALARM  (*when woken, advance pseudo time*)
  45        end
  45      end;    (*HOLD*)
  47
  47    procedure *ADVANCE;  (*keep waking up next job*)
  48     begin
  49      ALARMCLOCK.qsignal
  51     end;
  53
```

```
 53     function *ENQUEUED;
 54     (*allow outside world to examine queue*)
 54      begin
 55       ENQUEUED := ALARMCLOCK.qlength
 59      end;
 61
 61    begin   (*SIMULATION*)
 62     TIME := 0
 65    end;    (*SIMULATION*)
 66
 66   procedure TICK;
 67    begin
 68     while (SIMULATION.TIME < SIMULATION.TIMELIMIT)
 72                             or
 72           (SIMULATION.ENQUEUED > 0) do
 81      begin
 81       if readyinsystem = 1 then SIMULATION.ADVANCE
 87      end    (*while*)
 88     end;   (*TICK*)
 89
 89    procedure TOCK; (*record passage of time*)
 90     begin
 91      while SIMULATION.TIME < SIMULATION.TIMELIMIT do
 96       begin
 96        SIMULATION.HOLD(1);
 99        writeln(SIMULATION.TIME, ' seconds')
115       end
116      end;   (*TOCK*)
117
117    procedure USER(I);
118    (*simulate user of the system*)
118     var JOB, JOBTIME, JOBNUMBER;
118      begin
119       for JOB := 1 to 5 do
122        begin   (*for JOB*)
123         MANAGER.ASSIGNJOB(JOBNUMBER);
126         JOBTIME := random mod 6 + 1;
133         writeln('Request job ',JOBNUMBER,' to finish at ',
167                  JOBTIME + SIMULATION.TIME);
175         SIMULATION.HOLD(JOBTIME);
179         writeln('End of job ',JOBNUMBER)
198        end;    (*for JOB*)
199       writeln('End of user ',I)
219      end;   (*USER*)
220
220    begin   (*SIMULATEUSERS*)
220     cobegin
220      USER(1);
223      USER(2);
225      USER(3);
227      TICK;
228      TOCK
228     coend
230    end.    (*SIMULATEUSERS*)
```

```
Request job 1 to finish at 6
Request job 2 to finish at 1
Request job 3 to finish at 1
1 seconds
End of job 2
Request job 4 to finish at 7
End of job 3
Request job 5 to finish at 5
2 seconds
3 seconds
4 seconds
End of job 5
Request job 6 to finish at 6
5 seconds
End of job 1
Request job 7 to finish at 11
End of job 6
Request job 8 to finish at 7
6 seconds
End of job 4
Request job 9 to finish at 12
End of job 8
Request job 10 to finish at 10
7 seconds
8 seconds
9 seconds
End of job 10
End of user 3
10 seconds
End of job 7
Request job 11 to finish at 15
11 seconds
End of job 9
Request job 12 to finish at 18
12 seconds
13 seconds
14 seconds
End of job 11
Request job 13 to finish at 19
15 seconds
16 seconds
17 seconds
End of job 12
Request job 14 to finish at 21
18 seconds
End of job 13
Request job 15 to finish at 24
19 seconds
20 seconds
End of job 14
End of user 1
End of job 15
End of user 2
```

Chapter 2: Synchronisers

The synchroniser is the message passing equivalent of the monitor concept (cf. chapter 1). When message passing is used for communication and synchronisation, concurrently executing processes send and receive messages.

Communication is accomplished because a process receives values as part of a message from the sender.

Synchronisation is accomplished by the constraint that messages can only be received once they have been sent.

Message passing in CLANG is a Many-to-One relationship where many "client" processes request rendezvous with one "server" process.

The "server" process is the synchroniser.

The "client" process is a concurrent process that wishes to communicate and synchronise with the synchroniser.

Once a rendezvous has been established the "server" and "client" processes are ready to communicate.

The general form of a synchroniser is:

```
synchroniser SNAME(parameters);
 const declarations
 var declarations
 entry point declarations     (2.1)

  procedure / function declarations

 begin
  body of the synchroniser    (2.2) & (2.3)
 end;
```

The synchroniser is an **active** process, (it must be launched from within a Cobegin..Coend construct), and as such executes concurrently with the "client" processes until a rendevzous is established. During this, the "client" process is suspended while the "server" process (the synchroniser) performs the rendezvous. On completion of the rendezvous both the "client" and the "server" processes once more proceed concurrently.

If the synchroniser reaches a point of rendezvous before there are any "client" processes available, the synchroniser is suspended until one arrives.

Use of synchronisers

(a) Synchronisers may only be declared at the outer level of a program after the global variable declarations. ie. they may not be declared local to any procedures, functions, monitors or other synchronisers.

(b) Synchronisers are active processes so they count against the number of concurrent processes allowed in a program at any one time (which is 10 in the current implementation).

(c) There may be more than one synchroniser per program, the limit being determined by (b) above.

(d) As in standard procedures the parameters to synchronisers are optional and may include both value and variable parameters. The limit to the number of parameters in the current implementation is 25. Complete arrays may not be used as parameters.

(e) Synchronisers may only be initiated from within a cobegin..coend construct and may not be called from any other position. They may be "called" by means of a rendezvous request. (cf. section 2.1.2)

Example of usage

The warehouse mentioned in the example to the user manual may be coded as a synchroniser as follows:

```
synchroniser WAREHOUSE;
 var SHOP;
 entry DEPOSIT(ITEM), REMOVE(var ITEM);    (2.1.1)

  begin    (*WAREHOUSE*)
   while activeinsystem > 1 do     (cf. chapter 3)
    begin
     accept DEPOSIT(ITEM) then                (2.2)
      begin
        SHOP := ITEM
      end;
     accept REMOVE(var ITEM) then            (2.2)
      begin
        ITEM := SHOP
      end
    end;    (*while*)
   stopconcurrency                  (cf. chapter 3)
  end;    (*WAREHOUSE*)
```

The problem of several processes to deposit and remove values from a buffer that is bounded in size may be dealt with by means of a synchroniser as follows:

```
synchroniser HANDLER;
 var BUFFER[Ø:7], SIZE, NUMBER;
 entry DEPOSIT(X), REMOVE(var X);

  begin   (*HANDLER*)
   SIZE := Ø;    (*buffer initially empty*)
   for NUMBER := 1 to 32 do
    begin
     select
      SIZE > Ø: accept REMOVE(var X) then
                 begin
                  GRAPHICS.DRAWB(SIZE,SPACE);
                  (*call to graphic routine*)
                  X := BUFFER[SIZE];
                  SIZE := SIZE - 1
                 end;
      SIZE < 6: accept DEPOSIT(X) then
                 begin
                  SIZE := SIZE + 1;
                  BUFFER[SIZE] := X;
                  GRAPHICS.DRAWB(SIZE,X)
                  (*call to graphic routine*)
                 end
     end   (*select*)
   end;   (*HANDLER*)
```

The call to the graphic routine allows the results to be graphically displayed on a screen addressable SOROC terminal. See the end of the chapter for the full working program.

2.1 Entry points

An entry point defines the point of rendezvous between the "server" process (the synchroniser) and the "client" process and specifies just how communication between the two processes will be performed at this point.

2.1.1 Entry point declaration

The entry point declarations provide a visible list, to the "client" processes, of requests that the "server" process can service. These together with a formal parameter list, through which the message passing will be performed, are declared in the synchroniser under the entry declarations.

The general form is:

```
entry REQUEST1(parameters), REQUEST2(parameters), ...,
      REQUESTn(parameters);
```

Notes:

(a) Entry points may only be declared at the outer level of synchronisers. The entry point declarations must be after the synchroniser's variable declarations and before any local procedures or functions.

(b) The same rule for naming identifiers apply to entry points, namely, eight significant characters.

(c) The parameters to the entry points are optional. The same rules for parameters to procedures / functions apply to the parameters of entry points. An entry point without parameters is purely a synchronisation point.

(d) The current implementation restricts the number of entry points that may be declared per program, to a maximum of 25. There is no limit to the number of entry points per synchroniser except in accordance with the above.

Examples of usage

```
entry DEPOSIT(ITEM), REMOVE(var ITEM);

entry REQUEST1, REQUEST2(A, var B, C);
```

2.1.2 Requests for rendezvous

A process wishing for a rendezvous with the synchroniser performs a "call" to the required entry point and is then suspended until the rendezvous is complete.

The call is made by prefixing the name of the synchroniser, to which the request is directed, to the name of the request, separated by a period ('.').

The general form is:

```
synchronisername.REQUEST(parameters)
```

Note: The parameter list of the synchroniser is **not** specified even if there are parameters to the synchroniser.

The entry point request may only be made from within a process that is executing concurrently with the synchroniser.

A request for rendezvous may not be made from within a synchroniser.

An entry point request must match exactly (in name, number, and type of parameters to an entry point declared in the synchroniser whose name is appended to the request.

Note: The entry points are the only parts of a synchroniser that are accessible outside the synchroniser.

Examples of usage

The two "client" processes, the producer and consumer, for the warehouse example, coded in section 2 as a synchroniser, may be coded as follows:

```
procedure PRODUCER;
 const SWEET = 1;
 var ITEM;
  begin
   while BUSINESS = GOOD do
    begin
     ITEM := SWEET;   (*produce item*)
     WAREHOUSE.DEPOSIT(ITEM)
     (*request for rendezvous at the entry
       point DEPOSIT*)
    end    (*while*)
  end;    (*PRODUCER*)
```

```
procedure CONSUMER;
 var ITEM, MOUTH;
  begin
   while DESIRE = GOOD do
    begin
     WAREHOUSE.REMOVE(ITEM)
     (*request for rendezvous at the entry
       point REMOVE*)
     MOUTH := ITEM     (*consume item*)
    end   (*while*)
  end;   (*CONSUMER*)
```

Note:

Notice how the number of parameters to the request for rendezvous correspond to the number of parameters to the entry points declared in the synchroniser WAREHOUSE in section 2. However the names of the parameters need not correspond.

Aside:

The synchroniser WAREHOUSE and the two processes, PRODUCER and CONSUMER, would be launched concurrently in the main program as follows:

```
cobegin
 WAREHOUSE;  }
 PRODUCER;   }  order unimportant
 CONSUMER    }
coend;
```

In the bounded buffer problem there are several processes wishing to install items and one wishing to fetch them.

The common process for the installers could be coded as:

```
procedure INSTALLER(I);
 var TIME, REQUIRED;
  begin   (*INSTALLER(I)*)
   for REQUIRED := 1 to 8 do
    begin
     for TIME := 1 to (100 + random mod 20) do
      begin
       (*manufacture product*)
      end;
     GRAPHICS.DRAWP(I, PRODUCT[I]);
     HANDLER.DEPOSIT(PRODUCT[I]);
     GRAPHICS.DRAWP(I, SPACE)
    end   (*for REQUIRED*)
  end;    (*INSTALLER(I)*)
```

The fetcher process could be coded as:

```
procedure FETCHER;
 var TIME, REQUIRED, ITEM;
  begin
   for REQUIRED := 1 to 16 do
    begin
     HANDLER.REMOVE(ITEM);
     GRAPHICS.DRAWC(ITEM);
     for TIME := 1 to (200 + random mod 100) do
      begin
       (*carry item away*)
      end;
     GRAPHICS.DRAWC(SPACE)
    end    (*for REQUIRED*)
  end;   (*FETCHER*)
```

2.2 The ACCEPT statement

The section of code in the synchroniser, in which the actual rendezvous or message passing takes place, is contained within an accept statement.

The general form is:

```
accept REQUEST(parameters) then
 begin
  statements
 end;   (*accept*)
```

The accept statement is a compound statement and encloses the statements which involve the parameters (if any) which create the communication.

An accept statement is the point in the synchroniser where the "server" process will be delayed until there is a corresponding request by a "client" process. Thus the rendezvous request in the "client" process and the accept statement in the "server" process provide the points of synchronisation between the two processes.

Notes:

(a) Accept statements may only be used within synchronisers. They may be used in procedures / functions declared local to the synchroniser.

(b) The number and type (ie. value or variable) of the parameters in the entry point declaration (cf. section 2.1.1) must match exactly the number and type of the parameters used in the corresponding accept statement. However, as with forward procedure declarations, the parameters need not have matching names.

eg. The following is legal

```
entry REQUEST(var A, B);
 ...
 accept REQUEST(var I, J) then
  begin
    ...
```

(c) The parameters of an accept statement are strictly local to it.

(d) There may be more than one accept statement per entry point declaration.

(e) Accept statements may not be nested. This implies that once synchronisation has been established between the "server" and "client" processes, the rendezvous must first be completed before the "server" process can deal with requests from other "client" processes.

(f) The requesting process will be delayed until the end of the accept statement and then both the synchroniser and the process, whose request has now been dealt with, will proceed concurrently once more.

If the synchroniser can never execute the corresponding accept statement for the request then deadlock will result.

Simarly if there is an accept statement for which there is no request then deadlock will again arise.

Example of usage

The communication and synchronisation between the producer process and the consumer process (cf. section 2.1.2) is achieved by means of the accept statements in the synchroniser WAREHOUSE reproduced here, as well as the producer and consumer processes, with line numbers for easy reference.

```
 1:  synchroniser WAREHOUSE;
 2:   var SHOP;
 3:   entry DEPOSIT(ITEM), REMOVE(var ITEM);
 4:    begin   (*WAREHOUSE*)
 5:     while activeinsystem > 1 do
 6:      begin
 7:       accept DEPOSIT(ITEM) then
 8:        begin
 9:         SHOP := ITEM
10:        end;
11:       accept REMOVE(var ITEM) then
12:        begin
13:         ITEM := SHOP
14:        end
15:      end     (*while*)
16:     stopconcurrency
17:    end;    (*WAREHOUSE*)
18:
19:   procedure PRODUCER;
20:    const SWEET = 1;
21:    var ITEM;
22:     begin   (*PRODUCER*)
23:      while BUSINESS = GOOD do
24:       begin   (*while*)
25:        ITEM := SWEET; (*produce ITEM*)
26:        WAREHOUSE.DEPOSIT(ITEM);
27:       end     (*while*)
28:     end;    (*PRODUCER*)
```

```
29:
30:     procedure CONSUMER;
31:      var ITEM, MOUTH;
32:       begin
33:        while DESIRE = GOOD do
34:         begin   (*while*)
35:          WAREHOUSE.REMOVE(ITEM);
36:          MOUTH := ITEM     (*consume ITEM*)
37:         end      (*while*)
38:       end;    (*CONSUMER*)
```

Firstly note that the entry points declared in line 3 correspond exactly in name, and number and type of parameters to the accept statements on lines 7 and 11, and exactly in name, and number of parameters to the requests for rendezvous on lines 26 and 35.

If no request from the producer process has arrived, implying the producer process has not yet reached line 26, by the time the WAREHOUSE synchroniser reaches the accept statement on line 7, accept DEPOSIT(ITEM) then the synchroniser will be suspended until such time as the request for rendezvous from the producer is forthcoming.

Notes:

(a) Should the request from the consumer process (at line 35) be made during the time the warehouse synchroniser is suspended, the only change in the state of the processes will be the suspension of the consumer process. The consumer process will remain suspended until after the warehouse synchroniser has executed the accept statement from line 11 to line 14. ie. dealt with the request to remove the item.

When the request for rendezvous (at line 26) arrives from the producer process, or if there was already a request by the producer process by the time the warehouse synchroniser reached line 7, it is now dealt with by the warehouse synchroniser. (The synchroniser is reactivated, if it had been suspended, immediately the request comes in.)

(2) The producer process will remain suspended (it was suspended immediately after executing its request for rendezvous at line 26) while the synchroniser executes lines 7, 8, 9 and 10. After the synchroniser has executed line 10 (ie. at the end of the accept statement) the producer process will be reactivated and proceed to execute concurrently (from line 27) with the warehouse synchroniser (and the consumer process, if active) once more.

Aside:

Because the producer process is reactivated immediately after the end of the accept statement (at line 10) additional lines could have been inserted in the synchroniser between lines 10 and 11 to allow the warehouse to be "tidied" before the comsumer's request is dealt with, while proceeding concurrently with the producer (and maybe the consumer) process.

eg.
```
for TIME := 1 to 20 do  (*TIME a variable*)
 begin
  (*sweep floor*)
 end;
```

Having dealt with the request from the producer to deposit the item, the warehouse synchroniser now deals with the consumer process' request to remove the item (at line 11).

Once again the warehouse synchroniser may either proceed to deal with the request or is suspended, depending on whether the consumer process' request (at line 34) has been forthcoming or not.

Having dealt with the consumer's request to remove the item, at line 14 (ie. at the end of the accept statement), the consumer process is reactivated (having been suspended after making the request) and once more the synchroniser and the two processes (warehouse, consumer and producer) proceed concurrently.

The while loop (lines 5 to 15) now readies the warehouse to deal with the producer again.

Synchronisation

The order in which the accept statements have been used in the warehouse synchroniser, ie. accept DEPOSIT(ITEM) at line 7 and accept REMOVE(var ITEM) later at line 11, has constrained the order in which the producer and consumer requests are dealt with. ie. alternatively, starting with the producer. This provides the neccessary synchronisation between the two processes to prevent the producer process trying to deposit an item in the warehouse that already has an item, or the consumer process trying to acquire an item that is not yet in the warehouse.

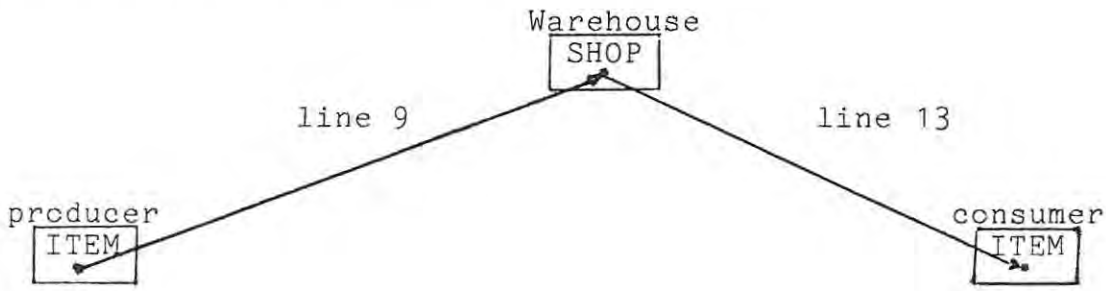
Communication

The "transfer" of the item from the producer, via the warehouse, to the consumer is achieved by the use of the parameters to the accept statements on lines 7 and 11.

The value parameter in the accept statement on line 7 accepts the value of ITEM passed from the producer process by the request for rendezvous at line 26. This value of ITEM is then stored in the variable SHOP, declared local to the synchroniser and therefore not susceptible to alteration from any other processes, at line 9 ie. SHOP := ITEM.

The value of SHOP is then passed across to the consumer process, by means of the variable parameter in the accept statement on line 11, by the warehouse synchroniser executing line 13 ie. SHOP := ITEM.

This can be shown diagramatically as:



Thus it can been seen that the accept statements in a synchroniser bring about the syncronisation and communication between concurrent processes.

2.3 The SELECT statement

The very "tight" synchronisation of processes by means of the accept statements prohibits any asynchronous operation and thus prevents most of the potential parallelism in a program being utilised.

The select statement solves this problem by giving the synchroniser a great deal of flexibility in allowing it to "choose", from a selection of possible requests, which request for rendezvous to deal with. This means that the synchroniser (ie. the "server" process) can avoid executing an accept statement and thereby committing itself to waiting for a "client" process to rendezvous, until a "client" is known to be waiting.

The general form is:

```
select
 guard condition 1: accept REQUEST1(parameters) then
                      begin
                        statements
                      end;
 guard condition 2: accept REQUEST2(parameters) then
                      begin
                        statements
                      end;
          .
          .
          .

 guard condition n: accept REQUESTn(parameters) then
                      begin
                        statements
                      end
end;    (*select*)
```

Notes:

(a) Select statements may only be used within a synchroniser - they may be used inside procedures / functions declared local to a synchroniser.

(b) Select statements may not be nested and in fact the only statement allowable in conjunction with a guard condition is an accept statement (but cf. section 2.3.3).

After executing an accept statement within the select statement, the next bit of code to be executed by the synchroniser is the first statement after the end of the select.

Diagramatically the flow of execution is:

```
  statements before the select statement

  select
    .
    .
    .
    some guard condition : accept REQUESTx(parameters) then
     (cf. section 2.3.1)       begin
                                ...
                               end;
    .
    .
    .
  end; (*select*)
 statements after the select statement
```

2.3.1 Guard conditions and the NOGUARD condition

The guard conditions allow control to be exercised by the synchroniser as to which request, or group of requests, for rendezvous, are more "favourable" to be dealt with than others.

The NOGUARD condition specifies that the request for rendezvous which it controls is always "favourable" for selection.

A guard condition consists of a boolean expression or the reserved word NOGUARD and is separated from the accept statement by a colon (':').

The general form is:

Boolean expression :

or

NOGUARD :

Notes:

(a) The NOGUARD condition is equivalent to a guard condition that is always true.

(b) In the current implementation there may only be a maximum of 20 guard conditions (including NOGUARDs) per select statement.

(c) A guard condition can be considered to be equivalent to accepting a request when a certain condition holds.

Examples of usage

```
select
  SIZE > 0 : accept REMOVE(var X) then
                  begin
                   X := BUFFER[SIZE];
                   SIZE := SIZE - 1
                  end;
  SIZE < 6 : accept DEPOSIT(X) then
                  begin
                   SIZE := SIZE + 1;
                   BUFFER[SIZE] := X
                  end;
  NOGUARD :  accept ANYTHING then
                  begin
                   writeln('In here')
                  end
end;   (*select*)
```

2.3.2 How the SELECT statement works

When a select statement is encountered all the guard conditions are first evaluated.

Two possible results can arise:

(1) If all the guard conditions evaluate to false (this would imply that no NOGUARD condition had been used) then the ELSE clause to the select statement (cf. section 2.3.3), if there is one, would be executed.

If there is no ELSE clause than the run time error

'NO VALID SELECT GUARD'

will be generated. This causes the execution of the program to abort.

Example

Given that the value of a synchroniser's variable SIZE is -1, then the following select statement in the synchroniser would generate the run time error

'NO VALID SELECT GUARD'

```
select
 SIZE > 6 : accept (*some request*) then
            begin
             ...
            end;
 SIZE = 0 : accept (*some request*) then
            begin
             ...
            end
 end;     (*select*)
```

(2) If NOGUARD conditions are used and / or there are some guard conditions that evaluate to true, then the synchroniser will "choose" to execute one of the accept statements controlled by a true guard condition.

The initial choice of an accept statement, from the set of possible ones, is random.

If this initial choice of accept statement would not cause the synchroniser to be delayed, then it is executed; However, should the initial choice of accept statement, if it were to be executed, cause the synchroniser to be suspended (ie. there has yet to be a request for rendezvous for that particular accept statement), then the set of possible accept statements is searched from the initial choice in a circular fashion until an accept statement is found which would not cause the synchroniser to delay.

This accept statement is then executed.

The synchroniser is therefore able to "peek" at the possible accept statements until a non-delaying one is found, thus increasing potential parallelism.

If there are no accept statements, among the set of possible ones, which would not cause the synchroniser to delay, then the synchroniser is suspended until such time as the first request for rendezvous, applicable to the set of possible accept statements, is received. The synchroniser is then reactivated and executes the relevant accept statement.

Note:

If there is more than one accept statement for a single request among the set of possible accept statements, then only one of these accept statements (chosen randomly) will be executed, should that request be the first request received.

Detailed example

The select statement is particularly useful for synchronising concurrent processes when alternate synchronisation is not really necessary ie. asynchronous communication.

This can be illustrated in the case of the warehouse, coded as a synchroniser (cf section 2.2). If the warehouse could accomodate more than one item at a time it would not be neccessary for the warehouse synchroniser to ensure that it first dealt with a deposit by the producer process followed by a remove by the consumer process. Should the size of the warehouse be bounded (say it can accomodate a maximum of 6 items) then it is still necessary to impose some limit on the number of producer's deposits that can be dealt with, before there is a remove from a consumer, so as to prevent the consumer from trying to remove an non-existent item or the producer trying to deposit an item in the warehouse which is already full. The role of ensuring this falls to the guard conditions of the select statement.

The warehouse synchroniser for this new case may be coded as follows (with line numbers for easy referrence):

```
 1:  synchroniser WAREHOUSE;
 2:   const MAXSIZE = 6; (*maximum space available*)
 3:   var SIZE, SHOP[1:MAXSIZE];
 4:   entry DEPOSIT(ITEM), REMOVE(var ITEM);
 5:
 6:    begin   (*WAREHOUSE*)
 7:     SIZE := 0;   (*warehouse is initially empty*)
 8:     while activeinsystem > 1 do
 9:      begin   (*while*)
10:       select
11:        SIZE < MAXSIZE: accept DEPOSIT(ITEM) then
12:                         begin
13:                          SIZE := SIZE + 1;
14:                          SHOP[SIZE] := ITEM
15:                         end;   (*deposit*)
16:        SIZE > 0: accept REMOVE(var ITEM) then
17:                   begin
18:                    ITEM := SHOP[SIZE];
19:                    SIZE := SIZE -1
20:                   end    (*remove*)
21:       end    (*select*)
22:      end;    (*while*)
23:     stopconcurrency
24:    end;    (*WAREHOUSE*)
```

(the producer and consumer processes remain unchanged cf. section 2.2)

Initially, as the value of the variable SIZE is set to 0 (at line 7), only one of the two guard conditions (the one at line 11) in the select statement evaluates to true (<u>ie.</u> SIZE < MAXSIZE) and so only a deposit request may be dealt with by the warehouse synchroniser. Thus if the consumer process makes a request to remove an item it will be suspended until such time as at least one item has been deposited. If there has yet to be a request by the producer to deposit an item then the warehouse synchroniser will be delayed until such time as one arrives.

The producer process' request to DEPOSIT an item is dealt with by the warehouse synchroniser from lines 11 to 15.

Having dealt with the request, the next line that the synchroniser executes is line 22; the first statement after the select statement.

The while loop (lines 8 to 22) now brings the synchroniser back to the start of the select statement (at line 10). Once more both the guard conditions (on lines 11 and 16) are evaluated, but this time, due to the fact that the value of SIZE is now 1 (the synchroniser having executed line 13), both the guard conditions evaluate to true.

Therefore now both DEPOSIT and REMOVE are possible requests that can be dealt with by the synchroniser.

One of these is chosen arbitrarily (say REMOVE).

If there has yet to be a request from the customer to remove an item, ie. by executing the accept statement the synchroniser would be obliged to delay itself, the state of the DEPOSIT request is then examined (ie. circular search).

If this too would cause the synchroniser to delay (there has yet to be a further request to deposit from the producer) then the synchroniser is suspended, but only as long as it takes for either request to come in. The first request that arrives is then dealt with.

If instead, the accept statement for the REMOVE request would not cause a delay (ie. the consumer is already waiting to remove an item) or if the REMOVE request would cause a delay, but the DEPOSIT would not, then that request is dealt with and the synchroniser is not suspended.

Note:

By the guard condition on line 11 (SIZE < MAXSIZE), there may not be more than 6 items in the warehouse at one time (ie. overflow the capacity of the warehouse). This is because, after six deposits without a remove the guard condition would evaluate to false (the value of SIZE would now be equal to MAXSIZE) and so the corresponding accept statement for a deposit would fall out of the set of possible requests that can be dealt with. Thus a further deposit request would cause the producer to be delayed until such time as at least one remove request from the consumer process had been dealt with and there is space in the warehouse again.

Thus by means of the select statement the order in which the producer deposits the items and the consumer removes them has been rendered unimportant except for when the extremes are encountered (ie. the warehouse is empty (SIZE = Ø) or the warehouse is full (SIZE = MAXSIZE)). This greatly improves the potential parallelism of the program.

<u>Aside</u>

The original case where the size of the warehouse was 1 could be coded by means of a select statement as follows:

```
synchroniser WAREHOUSE;
 var SIZE, SHOP;
 entry DEPOSIT(ITEM), REMOVE(var ITEM);
  begin   (*WAREHOUSE*)
   SIZE := Ø;  (*warehouse is initially empty*)
   while activeinsystem do
    begin   (*while*)
     select
      SIZE < 1: accept DEPOSIT(ITEM) then
                 begin
                  SIZE := SIZE + 1;
                  SHOP := ITEM
                 end;   (*deposit*)
      SIZE > Ø: accept REMOVE(var ITEM) then
                 begin
                  ITEM := SHOP;
                  SIZE := SIZE + 1
                 end   (*remove*)
     end    (*select*)
    end;    (*while*)
   stopconcurrency
  end;   (*WAREHOUSE*)
```

2.3.3 The ELSE clause to the select statement

If all the guard conditions to a select statement evaluate to false then, if there is one, the ELSE clause to the select statement will be executed and followed by the first statement after the select statement.

The general form is:

```
select
 guard condition 1: accept REQUEST1(paramaters) then
                      begin
                       statements
                      end;
   .
   .
   .
 guard condition n: accept REQUESTn(parameters) then
                      begin
                       statements
                      end   (*REQUESTn*)
 else
  begin
   statements
  end    (*else clause*)
 end;   (*select*)
```

Note

(a) The statements in an ELSE clause may not include another select statement, but they may include any other statements allowable in a synchroniser, including accept statements.

The ELSE clause is a much neater and concise way of imposing an if...then...else condition on a select statement.

The ELSE clause may thus be used to prevent the run time error 'NO VALID SELECT GUARD' from occurring and so allows recovery should all the guard conditions evaluate to false. (If the run time error occurs the program aborts execution.)

Note

The NOGUARD condition is equivalent to a guard condition which always evaluates to true, so the ELSE clause is rendered redundant if a NOGUARD condition is used in a select statement. A warning message to this effect will be generated if an ELSE clause is used in a select statement where NOGUARD conditions are present.

Example of usage

If the example of the producer, consumer and warehouse, where the size of the warehouse is greater than 1 (cf. section 2.3.2), is extended to allow the user to read in the initial number of items in the warehouse, some checks would have to be made to ensure a valid initial number was received. This check may easily be made by means of an ELSE clause to the select statement, which can issue a relevant message informing the user if the value input was incorrect.

The modified warehouse synchroniser might be coded as follows:

```
synchroniser WAREHOUSE;
 const MAXSIZE = 6; (*max. size of the warehouse*)
 var SIZE, SHOP[1:MAXSIZE];
 entry DEPOSIT(ITEM), REMOVE(var ITEM);

  begin   (*WAREHOUSE*)
   read(SIZE); (*user inputs initial no. of items*)
   while activeinsystem > 1 do
    begin
     select
      (SIZE>=0) and (SIZE<MAXSIZE):
         accept DEPOSIT(ITEM) then
          begin
           SIZE := SIZE + 1;
           SHOP[SIZE] := ITEM
          end;   (*deposit*)
      (SIZE>0) and (SIZE<=MAXSIZE):
         accept REMOVE(var ITEM) then
          begin
           ITEM := SHOP[SIZE];
           SIZE := SIZE - 1
          end   (*remove*)
      else
       begin
        writeln('The input of ',SIZE,' is invalid')
       end   (*else clause*)
     end    (*select*)
    end;    (*while*)
   stopconcurrency
  end;   (*WAREHOUSE*)
```

Example programs

Here then are the full working programs from which segments have been taken to illustrate various concepts. Those programs that do not produce grahic output have been included with their results, the other results can be seen when the programs are executed using a SOROC addressable screen terminal.

Toy Compiler Mark 21.2C m cv s spr nb

```
  0   (*$w- *)
  0   (*$s+ *)
  0   program COMMONEVENT;
  0
  0   (***************************************************)
  0   (* This program deals with the common event in our *)
  0   (* daily lives, that of a producer who produces an *)
  0   (* item and delivers it to a warehouse, from where *)
  0   (* a consumer acquires the item and consumes it.   *)
  0   (*                                                 *)
  0   (* The  warehouse  is implemented  by  means of a  *)
  0   (* synchroniser  and the producer and consumer by  *)
  0   (* means of two concurrent processes.              *)
  0   (***************************************************)
  0
  0   const GOOD = 1;
  1         MAXTIME = 5;
  1
  1     synchroniser WAREHOUSE;
  2      var SHOP, TIME;
  2      entry DEPOSIT(ITEM), REMOVE(var ITEM);
  2
  2       begin   (*WAREHOUSE*)
  3        while activeinsystem > 1 do
  7         begin   (*while*)
  7          accept DEPOSIT(ITEM) then
  8           begin
  8            SHOP := ITEM;
 12            writeln('Item has been deposited')
 38           end;   (*DEPOSIT*)
 42          accept REMOVE(var ITEM) then
 43           begin
 43            ITEM := SHOP;
 48            writeln('Item has been removed')
 73           end;   (*REMOVE*)
 77          for TIME := 1 to 100 do
 80           begin
 81            (*ready warehouse for next item*)
 81           end
 81         end;    (*while*)
 83        stopconcurrency
 84       end;    (*WAREHOUSE*)
 85
```

```
 85     procedure PRODUCER;
 86      const SWEET = 1;    (*item that is being produced*)
 86      var ITEM, BUSINESS;
 86       begin
 87        BUSINESS := MAXTIME;
 90        while BUSINESS >= GOOD do
 95         begin
 95          ITEM := SWEET;    (*produce item*)
 98          writeln('Item has been produced');
125          WAREHOUSE.DEPOSIT(ITEM);
128          BUSINESS := BUSINESS - 1
132         end    (*while*)
134       end;   (*PRODUCER*)
136
136      procedure CONSUMER;
137       var ITEM, MOUTH, DESIRE;
137        begin
138         DESIRE := MAXTIME;
141         while DESIRE >= GOOD do
146          begin
146           WAREHOUSE.REMOVE(ITEM);
148           MOUTH := ITEM;    (*consume item*)
152           writeln('Item has been consumed');
179           DESIRE := DESIRE - 1
183          end    (*while*)
185        end;   (*CONSUME*)
187
187  begin  (*COMMONEVENT*)
188   writeln('About to start business');
216   cobegin
216    WAREHOUSE;    (*WAREHOUSE is an active process*)
218    PRODUCER;
219    CONSUMER
219   coend;
221   writeln('Business is closed for the day')
254  end.   (*COMMONEVENT*)
```

```
About to start business
 Item has been produced
 Item has been deposited
 Item has been removed
 Item has been consumed
 Item has been produced
 Item has been deposited
 Item has been removed
 Item has been produced
 Item has been consumed
 Item has been deposited
 Item has been removed
 Item has been consumed
 Item has been produced
 Item has been deposited
 Item has been removed
 Item has been consumed
 Item has been produced
 Item has been deposited
 Item has been removed
 Item has been consumed
Business is closed for the day
```

```
Toy Compiler Mark 21.2C m cv s spr nb

    0   (*$S+ *)
    0   (*$W- *)
    0   program BOUNDEDBUFFER;
    0
    0   (*********************************************)
    0   (* This  is a solution to the bounded buffer *)
    0   (* problem using a synchroniser  and message *)
    0   (* passing  rendezvous. Also  included  is a *)
    0   (* monitor which handles the pseudo graphics *)
    0   (* to allow for graphic output onto a screen *)
    0   (* addressable SOROC terminal                *)
    0   (*********************************************)
    0
    0   const SPACE = 32;    (*ASCII equivalent*)
    1         ESC = 27;     EQL = 61;
    1         STAR = 42;    HASH = 35;
    1         DOWN = 124;   ACROSS = 45;
    1    var PRODUCT[1:2];
    1
    1    monitor GRAPHICS;
    2     var PX[1:2], PY[1:2], CX, CY, BX[1:6], BY[1:6], LC;
    2
    2      procedure GOTOXY(X, Y, CH);
    3      (*writes CH to relevant screen position*)
    3       begin
    4        write(ESC$,EQL$);    (*necessary start characters*)
   10        write(32+Y$, 32+X$, CH$)
   25       end;
   26
   26      procedure *DRAWP(I,CH);
   27      (*draw production under producer*)
   27       begin GOTOXY(PX[I], PY[I], CH) end;
   47
   47      procedure *DRAWC(CH);
   48      (*draw acquired item under consumer*)
   48       begin GOTOXY(CX, CY, CH) end;
   58
   58      procedure *DRAWB(I, CH);
   59      (*draw buffer being modified*)
   59       begin GOTOXY(BX[I], BY[I], CH) end;
   79
   79      procedure INITIALISE;
   80      (*set up position arrays*)
   80       var LC;
   80        begin
   81         for LC := 1 to 6 do BX[LC] := 42;
   94         for LC := 4 to 9 do BY[LC-3] := LC;
  110         PX[1] := 10; PY[1] := 4;
  124         PX[2] := 24; PY[2] := 4;
  138         CX := 57;    CY := 4
  144        end;   (*INITIALISE*)
  145
```

```
145    begin   (*GRAPHICS*)
146     write(ESC$,'*');    (*clear screen*)
153     GOTOXY(5, 3, SPACE); write('PRODUCER 1');
171     GOTOXY(20,3, SPACE); write('PRODUCER 2');
189     GOTOXY(38,3, SPACE); write('BUFFER     ');
207     GOTOXY(53,3, SPACE); write('CONSUMER   ');
225     for LC := 4 to 9 do
228      begin
229       GOTOXY(40,LC,DOWN); GOTOXY(44,LC,DOWN)
239      end;
240     for LC := 41 to 43 do
243      GOTOXY(LC,10,ACROSS);
250     INITIALISE
251    end;    (*GRAPHICS*)
252
252
252   procedure DEFAULT;
253    begin write(ESC$, EQL$, 32+22$, 32+1$) end;
269
269   synchroniser HANDLER;
270    var BUFFER[0:7], SIZE, NUMBER;
270    entry DEPOSIT(X), REMOVE(var X);
270     begin   (*HANDLER*)
271      SIZE := 0;    (*buffer initially empty*)
274      for NUMBER := 1 to 32 do
277       select
278        SIZE > 0: accept REMOVE(var X) then
289                   begin
289                    GRAPHICS.DRAWB(SIZE, SPACE);
294                    X := BUFFER[SIZE];
304                    SIZE := SIZE - 1
310                   end;    (*REMOVE*)
312        SIZE < 6: accept DEPOSIT(X) then
324                   begin
324                    SIZE := SIZE + 1;
330                    BUFFER[SIZE] := X;
339                    GRAPHICS.DRAWB(SIZE, X)
345                   end     (*DEPOSIT*)
347       end    (*select*)
351     end;   (*HANDLER*)
352
352    procedure INSTALLER(I);
353     var TIME, REQUIRED;
353      begin   (*INSTALLER(I)*)
354       for REQUIRED := 1 to 8 do
357        begin
358         for TIME := 1 to (100 + random mod 20) do
365          begin
366           (*manufacture product*)
366          end;
367         GRAPHICS.DRAWP(I, PRODUCT[I]);
378         HANDLER.DEPOSIT(PRODUCT[I]);
386         GRAPHICS.DRAWP(I, SPACE)
391        end    (*for REQUIRED*)
392      end;   (*INSTALLER(I)*)
```

```
393
393      procedure FETCHER;
394       var TIME, REQUIRED, ITEM;
394        begin
395         for REQUIRED := 1 to 16 do
398          begin
399           HANDLER.REMOVE(ITEM);
401           GRAPHICS.DRAWC(ITEM);
405           for TIME := 1 to (200 + random mod 100) do
412            begin
413             (*carry item away*)
413            end;
414           GRAPHICS.DRAWC(SPACE)
417          end   (*for REQUIRED*)
418        end;   (*FETCHER*)
419
419       begin   (*BOUNDEDBUFFER*)
419        PRODUCT[1] := STAR; PRODUCT[2] := HASH;
433        cobegin
433         HANDLER;
435         FETCHER;
436         INSTALLER(1);
438         INSTALLER(2)
439        coend;
441        DEFAULT   (*put cursor at bottom of screen*)
442       end.   (*BOUNDEDBUFFER*)
```

Chapter 3: Additional useful features

Chapters 1 and 2 describe the two high level constructs available in CLANG for concurrent process synchronisation and communication. This chapter details two useful "external" function calls and two useful "external" procedure calls available in CLANG for use with concurrent programming.

3.1 ACTIVEINSYSTEM

This is a function which will return the number of concurrent processes currently executing. If ACTIVEINSYSTEM is used while there are no concurrent processes active then the value 0 will be returned.

Example

If four processes are launched concurrently, but by the time one of them executes the ACTIVEINSYSTEM call one of the processes has finished its concurrent execution then the value 3 will be returned. It does not matter that the other two processes may be temporarily suspended - they have as yet not finished concurrent execution.

Example of usage

In the case of the warehouse example (cf. Introduction) coded as a synchroniser (cf. chapter 2) ACTIVEINSYSTEM was used as follows:

```
begin   (*WAREHOUSE*)
 while activeinsystem > 1 do
  begin

   (*deal with rendezvous requests*)

   for TIME := 1 to 100 do
    begin
     (*ready warehouse for next item*)
    end
  end;   (*while loop*)
```

This ensured that the warehouse synchroniser would continue dealing with the requests for rendezvous from the consumer and producer processes until such time as the synchroniser (which is an active process cf. section 2) is the only process active in the system. ie. The other two processes in the system, the producer process and the consumer process, have both finished execution.

Note:

The "for loop":

```
for TIME := 1 to 100 do
 begin
  (*ready warehouse for item*)
 end;
```

is a neccessary time delay so as to remove the threat of deadlock as it prevents the synchroniser from checking the condition of the "while loop":

```
while activeinsystem > 1 do
```

before the consumer process has finished, because if the condition is checked before such time, ACTIVEINSYSTEM will be 2 and so the synchroniser will proceed to execute the contents of the "while loop". This means that the synchroniser will attempt to deal with requests for rendezvous which will never be forthcoming as both the producer process and the consumer process have completed their execution - deadlock.

3.2 READYINSYSTEM

This function call will return the number of concurrent processes ready for scheduling which includes those processes which have not yet terminated or been suspended. If READYINSYSTEM is used while there is no concurrency in progress the value Ø is returned.

Example

If four processes are launched concurrently, but at the time one of the processes executes the READYINSYSTEM call, one of the processes has finished concurrent execution and another is suspended waiting for an event to occur, the value 2 is returned.

Example of Usage

The provision of pseudo-time by means of a monitor (cf. section 1) and the process, TICK, to keep the "clock" going to wake up jobs when complete (cf. section 1.1.1.2) made use of the function call READYINSYSTEM as follows:

```
begin   (*TICK*)
 if readyinsystem = 1 then
   SIMULATION.ADVANCE;
 ...
end;    (*TICK*)
```

This use of READYINSYSTEM ensures that once all the other processes in the system, other than process TICK, are either completed or suspended then pseudo-time can be advanced.

3.3 STOPCONCURRENCY

This "external" procedure call will do as its name implies: on execution it effectively ends the concurrent execution of all processes, regardless of their condition, and reactivates the main process ie. starts execution of the main program again, after the relevant cobegin..coend construct that launched the now aborted concurrent processes. If no concurrency is in operation then this procedure call will have no effect.

Care must be taken with the use of the STOPCONCURRENCY procedure call because of its carte blanche ability to stop concurrency.

Example of usage

The majority of usage envisaged for this "external" call will be when dealing with a finite state problem associated with the synchroniser concept (cf. chapter 6 of the assessment), as it can be used as a simple method, in conjunction with the "external" function calls of ACTIVEINSYSTEM (cf. section 3.1) or READYINSYSTEM (cf. section 3.2)), of controlling the number of executions of the "server" process without having to calculate exactly the desired number of executions required to deal with all the rendezvous requests.

Thus in the coding of the warehouse as a synchroniser (cf. chapter 2) STOPCONCURRENCY was used as follows:

```
begin   (*WAREHOUSE*)
 ...

 while activeinsystem > 1 do
  begin   (*while*)
   (*deal with the requests for rendezvous*)
  end;    (*while*)
 stopconcurrency
end;    (*WAREHOUSE*)
```

This ensures that once the execution of the producer and consumer processes have finished (cf. section 3.1), concurrent execution is stopped and the main program resumes ie. The warehouse synchroniser is no longer needed and so its concurrent execution is aborted.

Note:

Any statements between the STOPCONCURRENCY call and the end of the process (or synchroniser) will be ignored as all concurrency is terminated immediately the STOPCONCURRENCY call is encountered.

ie If the end of the warehouse synchroniser had been coded:

```
 stopconcurrency;
 writeln('Finished with the warehouse');
end;    (*WAREHOUSE*)
```

The message ('Finished with the warehouse') would never be written.

3.4 SWITCH

This "external" procedure call may be used to cause a process switch.

Example of usage

SWITCH may be used instead of a "delaying for loop" at the end of the while loop (cf. section 3.1), in the warehouse coded as a synchroniser, to prevent the deadlock that might otherwise occur.

```
begin   (*WAREHOUSE*)
 while ACTIVEINSYSTEM do
  begin

   (*deal with rendezvous requests*)

   switch
  end;   (*while*)
```

Chapter 4: Error and Warning messages

This chapter specifies the error messages, both compile time and run time, as well as the warning messages, that arise due to incorrect usage of the features described in chapters 1 and 2.

As well as an explanation of the error message there is also an example showing how the error / warning message might arise.

4.1 Error messages relating to chapter 1

4.1.1 Compilation errors

CONDITION VARIABLES ONLY IN MONITORS

Condition variables (cf. section 1.3) may only be declared, and operations (cf. section 1.3.2) performed on them within a monitor.

Example of occurrence

This error will occur if an attempt is made to declare a condition variable at a global level as follows:

```
          program ERRORS;
           const ONE = 1;
           var I, J;
           condition ALARMCLOCK;
      **** ^CONDITON VARIABLES ONLY IN MONITORS
```

INCORRECT CONDITION VARIABLE USAGE

Only the operations QUEUE and QLENGTH (cf. sections 1.3.2.4 & 1.3.2.5) act as "function" calls and may be used as such in conjunction with condition variables.

Simarly only the operations QWAIT, QPWAIT(priority) and QSIGNAL (cf. sections 1.3.2.1 - 1.3.2.3) act as "procedures" and may be used as such in conjunction with condition variables.

Any attempt to deviate from this, or if any incorrect operation is used in conjunction with a condition variable, this error will result.

Example of occurrence

This error will occur if an incorrect operation is performed on the condition variable BUSY as follows:

```
                monitor MON1;
                 var FULL;
                 condition BUSY;

                 procedure *CHECK;
                  begin
                   if FULL = 1 then
                    BUSY.DELAY;
         ****            ^INCORRECT CONDITION VARIABLE USAGE
```

MONITORS IN MAIN BLOCK ONLY

Monitors may only be declared at the outer level of a program (cf. section 1). Any attempt to declare them at any other level ie. local to procedures, functions, synchronisers or other monitors, will result in this error.

Example of occurrence

An attempt to declare the monitor MON1 local to the procedure FIRST results in this error as follows:

```
               program ERROR;
                var I, J;

                procedure FIRST;
                 monitor MON1;
          **** ^ MONITORS IN MAIN BLOCK ONLY
```

NO SEMAPHORES IN MONITORS

Synchronisation is achieved within monitors by means of condition variables (cf. section 1.3). Any attempt to use the low level synchronisation primitive, the semaphore, in a monitor will result in this error.

Example of occurrence

An attempt to perform the low level synchronisation operation WAIT on the semaphore SEMA within the procedure, LOC, local to the monitor MON, results in this error as follows:

```
               program WRONG;
                var SEMA;

                 monitor MON;
                  procedure LOC;
                   begin
                    wait(SEMA);
               **** ^ NO SEMAPHORES IN A MONITOR
```

ONLY CURRENT MONITOR VARIABLES MAY BE SAVED

When using the explicit method of ensuring the invariance of monitor variables, SAVE(parameters) (cf. section 1.4.2.2.1), only those variables declared local to the monitor in which SAVE(parameters) is used may be included as parameters to the SAVE.

Example of occurrence

This error occurs in the following segment of code because an attempt was made to include the global variable, GLOB, as a parameter to a SAVE used in a procedure, LOC, declared local to the monitor MON.

```
      program INCORRECT;
       var GLOB;

        monitor MON1;

         procedure LOC;
          begin
           SAVE(GLOB);
****                ^ ONLY CURRENT MONITOR VARIABLES MAY BE SAVED
```

ONLY STARRED IDENTIFIERS ACCESSIBLE

Only an identifier (constant, variable, procedure or function) that is declared as starred (cf. section 1.1.1) may be accessed from outside the monitor in which it was declared. Any attempt to access an identifier that is declared local to a monitor but is not starred will result in this error.

Example of occurrence

In the following the variable I has not been declared as starred within the monitor MON, so when an attempt is made to access I outside the monitor MON this error results.

```
               monitor MON;
                var I;
                 ...
                begin   (*MON*)
                 I:=0;
                end;    (*MON*)


              begin   (*main program*)
               if MON.I = 1 then
         ****              ^ ONLY STARRED IDENTIFIERS ACCESSIBLE
```

STARRED IDENTIFIERS ONLY IN MONITORS

Starred identifiers (cf. section 1.1.1) may only be declared at the outer level of monitors. Any attempt to declare an identifier as starred elsewhere will cause this compilation error to occur.

Example of occurrence

An attempt to declare the variable WRONG as starred inside the procedure PROC, declared local to monitor MON, causes this error as WRONG is not being declared at the outer level of a monitor.

```
monitor MON;

 procedure PROC;
  var *WRONG;
 **** ^ STARRED IDENTIFIERS ONLY IN MONITORS
```

TOO MANY CONDITION VARIABLES

Only 25 condition variables (cf. section 1.3.1) may be declared per program. Any attempt to declare more than 25 condition variables per program will result in this error. Arrays of condition variables may be declared, but every array element counts against this limit.

Example of occurrence

In the following program segment an array, BUSY, of condition variables, of 25 elements is declared. When the next condition variable is declared this error results as the limit of 25 has now been exceeded.

```
       monitor PROBLEM;
        condition BUSY[1:25], FREE;
****                        ^ TOO MANY CONDITION VARIABLES
```

TOO MANY MONITOR DECLARATIONS

Only 15 monitors may be declared per program. Any attempt to declare more monitors will result in this error.

Example of occurrence

If 15 monitors had been declared prior to the declaration of monitor MON16, then this error occurs as follows:

```
              ...

              monitor MON16;
         **** ^ TOO MANY MONITORS
```

SAVE/RESTORE ONLY IN MONITOR PROC/FUNC

The operations SAVE(parameters) and RESORE to ensure the invariance of monitor variables (cf. section 1.4.2.1) may only be used within procedures or functions declared local to a monitor. Any attempt to use them elsewhere will result in this error.

Example of occurrence

If the operation SAVE is used in the body of a monitor (cf. section 1.2) MON then this error will result.

```
            monitor MON;
             var I;
             ...

            begin (*body of MON*)
             SAVE(I);
        **** ^ SAVE/RESTORE ONLY IN MONITOR PROC/FUNC
```

4.1.2 Run time errors

Once the program starts to execute, certain errors, related to incorrect usage of the features in chapter 1, which can not be detected at compile time, will cause the program to abort.

PRIORITY < Ø

If the value of the priority expression, specified in the operation QPWAIT(priority) on a condition variable (cf. section 1.3.2.2), evaluates to less then Ø then this run time error will occur.

Example of occurrence

If the program contained the following operation on the condition variable CONDVAR:

```
CONDVAR.QPWAIT(6-7);
```

then this error will occur when that instruction is executed.

4.2 Error and warning messages relating to chapter 2

4.2.1 Compilation errors

ACCEPT EXPECTED

An accept statement must follow a guard condition. If any other statement follows a guard condition (or NOGUARD condition) this error will occur.

Example of occurrence

Because a compound statement is used after the guard statement in the following select statement this error occurs.

```
              synchroniser SYNC;
               var SIZE;

                select
                  SIZE > 6: begin
          ****                ^ ACCEPT EXPECTED
```

ENTRY POINT CALL IN ILLEGAL POSITION

Entry points may only be "called" from a process that is executing concurrently with the synchroniser in which the corresponding entry point was declared. Any attempt to "call" an entry point from within a monitor, synchroniser, the body of the main program or from within an accept statement, will result in this error.

Example of occurrence

An entry point "call" to the entry point DEPOSIT, declared in the synchroniser SYNC is made from the body of the program thus resulting in this error.

```
          begin (*body of the main program*)
           SYNC.DEPOSIT(ITEM);
    ****              ^ENTRY POINT CALL IN ILLEGAL POSITION
```

ENTRY POINT EXPECTED

If some other identifier other then a previously declared entry point is used in an accept statement after the reserved word "accept", then this error will occur.

Example of occurrence

In the following segment of code the function call, to the function FIND, has been used in an accept statement, after the reserved word "accept", instead of an entry point, hence the error.

```
                function FIND;
                 begin
                  ...
                 end;
                ...

               accept FIND then
      ****             ^ ENTRY POINT EXPECTED
```

ENTRY POINTS ONLY IN SYNCHRONISERS

Entry points may only be declared at the outer level of a synchroniser. If entry points are declared anywhere else this error occurs.

Example of occurrence

In the following segment of code the entry point DEPOSIT has been declared in the procedure PROC, which in turn has been declared local to the synchroniser SYNC. As DEPOSIT is not declared at the outer level of SYNC this error occurs.

```
               synchroniser SYNC;

                procedure PROC;
                 var I;
                 entry DEPOSIT(ITEM);
           **** ^ ENTRY POINTS ONLY IN SYNCHRONISERS
```

NO NESTED ACCEPT STATEMENTS

Accept statements may not be nested *ie.* there may not be an accept statement within another accept statement. Any attempt to do so will result in this error.

Example of occurrence

Here the accept statement for the entry point DEPOSIT was nested in the accept statement for the entry point REMOVE, hence the error.

```
              entry DEPOSIT(ITEM), REMOVE(var ARTICLE);
               ...
               accept REMOVE(var ARTICLE) then
                begin
                 accept DEPOSIT(ITEM) then
          **** ^ NO NESTED ACCEPT STATEMENTS
```

NO NESTED SYNCHRONISERS

No synchronisers may be declared local to a synchroniser. If the synchroniser's declarations are nested then this error occurs.

Example of occurrence

Synchroniser SYNC2 has been declared local to synchroniser SYNC1 so this error results.

```
              synchroniser SYNC1;
               var SIZE;

                synchroniser SYNC2;
          **** ^ NO NESTED SYNCHRONISERS
```

ONLY ENTRY POINTS ACCESSIBLE

An entry point is the only part of a synchroniser that is accessible to other processes outside the synchroniser (unlike monitor identifiers (cf.section 1.1.1)). If an attempt is made to "access" a part of a sychroniser other than an entry point this error arises.

Example of occurrence

In this example an attempt was made to call the procedure PROC declared local to the synchroniser SYNC, from outside SYNC; hence the error.

```
                synchroniser SYNC;

                 procedure PROC;
                ...
                end;   (*SYNC*)

                procedure PROCES;
                 begin
                  SYNC.PROC;
         ****          ^ ONLY ENTRY POINTS ACCESSIBLE
```

SELECT ONLY IN SYNCHRONISER

Select statements may only be used within a synchroniser. (They may also be used in procedures / functions declared local to a synchroniser.) If used anywhere else this error occurs.

Example of occurrence

Here a select statement was used in the body of the main program; hence the error.

```
                begin   (*body of main program*)
                 select
            **** ^ SELECT ONLY IN SYNCHRONISER
```

SYNCHRONISER ONLY IN MAIN BLOCK

Synchronisers (cf. section 2) may only be declared at the outer level of a program. They may not be declared local to any procedures, functions, monitors or other synchronisers. If they are this error occurs.

Example of occurrence

The synchroniser SYNC has been declared local to the procedure PROC so this error occurred.

```
              procedure PROC;
               var I;

                 synchroniser SYNC;
          **** ^ SYNCHRONISER ONLY IN MAIN BLOCK
```

TOO MANY ENTRY POINTS

Only a maximum of 25 entry points are allowed per program. Any further entry point declarations will generate this error.

Example of occurrence

```
              entry REQUEST1,...,REQUEST26;
      ****                         ^ TOO MANY ENTRY POINTS
```

TOO MANY GUARD CONDITIONS

The maximum number of guard (and NOGUARD) conditions allowable per select statement is 20. If more than 20 guard conditions are used in a single select statement this error occurs.

Example of occurrence

When the 21st guard condition for the select statement is encountered this error occurs.

```
              select
               ...

               SIZE < 6: (*guard condition 21*) accept ...
          **** ^ TOO MANY GUARD CONDITIONS
```

4.2.2 Warning messages

Warning messages are used to inform the user of a possible problem that he / she may have inadvertantly overlooked. Their occurrence will not effect the compilation of the program, but may give the user some hint of unforseen problems.

These warning messages may be "turned off" by the use of the (*$W- *) compiler directive.

MISSING RESTORE

Following a conditioned PLOXY point (cf. section 1.4.2) if there is no RESTORE instruction this warning will be generated to inform the user that the RESTORE is missing ie. any variables that might have been SAVEd before the conditioned PLOXY point will not be RESTOREd so invariance can not guaranteed.

Example of occurrence

```
              save(I);
              BUSY.qwait;    (*conditioned PLOXY point*)
              I := 5;
   *WARNING* ^ MISSING RESTORE
```

MONITOR VARIABLES NOT INVARIANT

If there is no SAVE(parameters) before a conditioned PLOXY point (cf. section 1.4.2) then this warning message will be issued to warn the user that, because the SAVE(parameters) is missing the monitor's variables can not be guaranteed to be invariant after the conditioned PLOXY point.

Example of occurrence

```
              I := 5;
              BUSY.qsignal;    (*conditioned PLOXY point*)
   *WARNING* ^ MONITOR VARIABLES NOT INVARIANT
```

4.2.3 Run time errors

Once the program starts to execute, certain errors, related to incorrect usage of the features in chapter 2, which can not be detected at compile time, will cause the program to abort.

CONCURRENCY NOT IN OPERATION

This run time error occurs when an attempt is made to perform an accept statement (cf. section 2.2) while concurrency is not in operation.

The error will also occur if an attempt is made to perform a "call" to an entry point ie. a rendezvous request (cf. section 2.1.2) while concurrency is not in operation.

NO VALID SELECT GUARD

If all the guard conditions of a select statement evaluate to false and there is no ELSE clause to the select statement (cf. sections 2.3.2 & 2.3.3) then this run time error will occur.

Example of occurrence

The following select statement will cause this run time error if the value of the variable SIZE were -1.

```
select
 SIZE > Ø: accept DEPOSIT(ITEM) then
            begin
             (*some statements*)
            end;
 SIZE = Ø: accept PLACE(ITEM) then
            begin
             (*some statements*)
            end;
end;   (*select*)
```

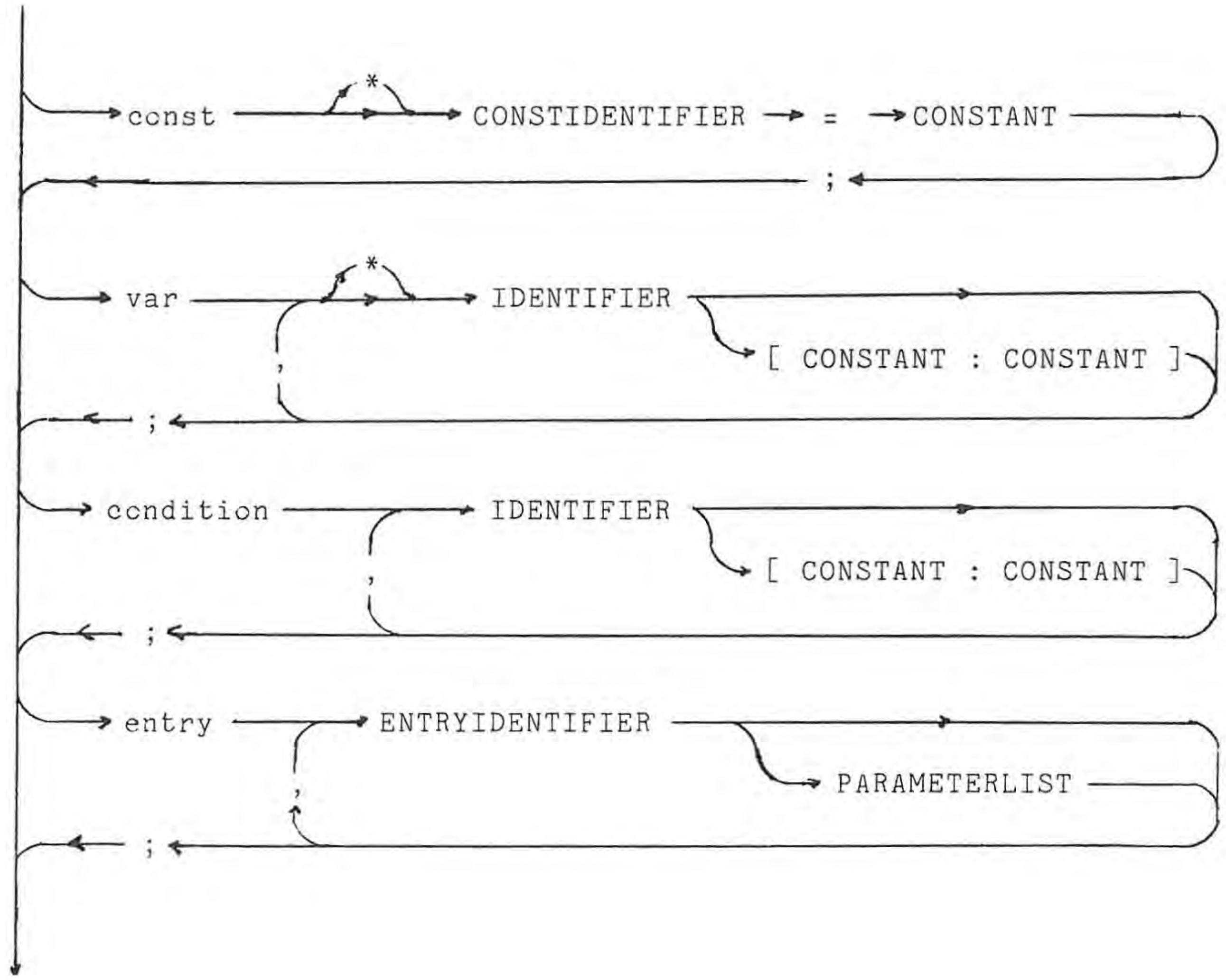
Syntax diagrams

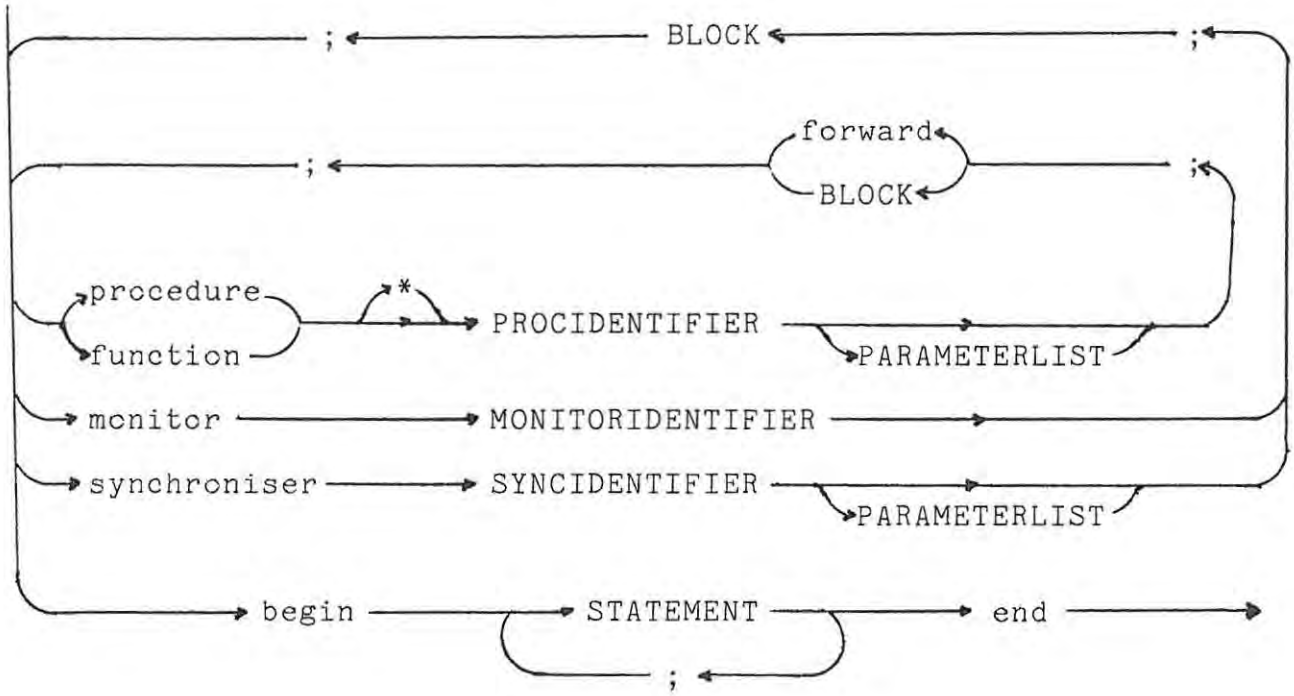
There are currently several versions of CLANG available. Below are the syntax diagrams for the latest release (CLANG 21.2C), which contains the monitor and synchroniser concepts. These syntax diagrams are known to be inadequate in several respects and should be studied in conjunction with the accompanying notes.

Syntax diagrams

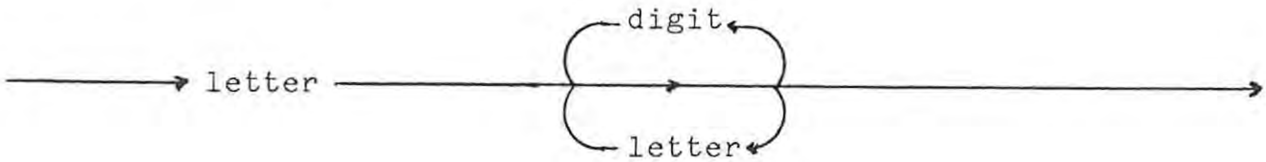
PROGRAM

BLOCK

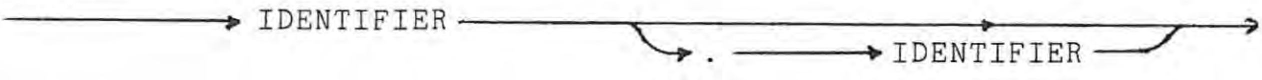




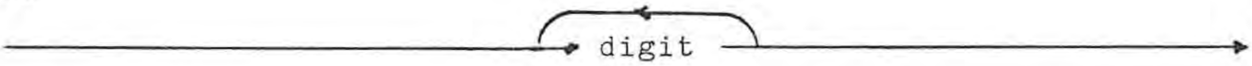
IDENTIFIER (including all semantic variations)



QUALIFIEDIDENTIFIER



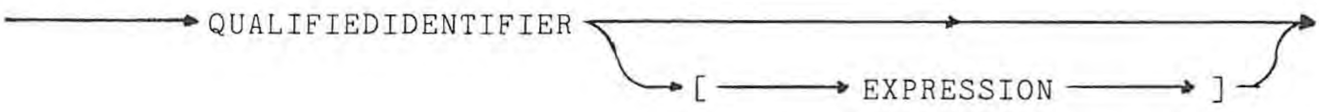
NUMBER



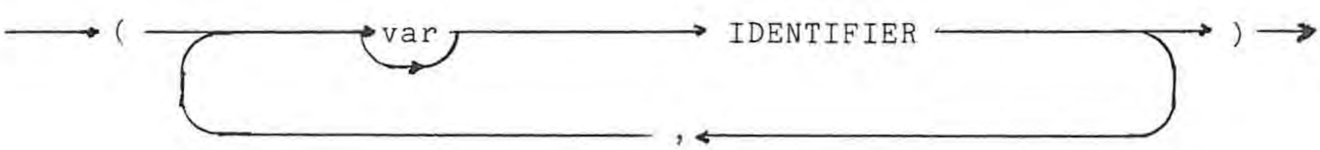
CONSTANT

STRING

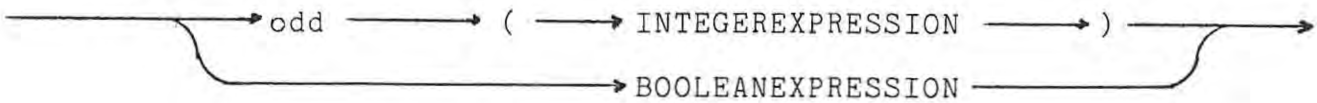
VARIABLE



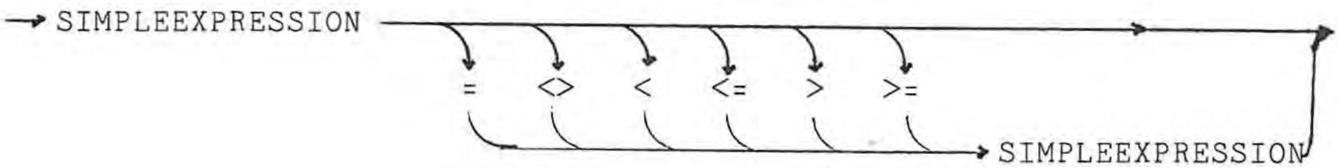
PARAMETERLIST



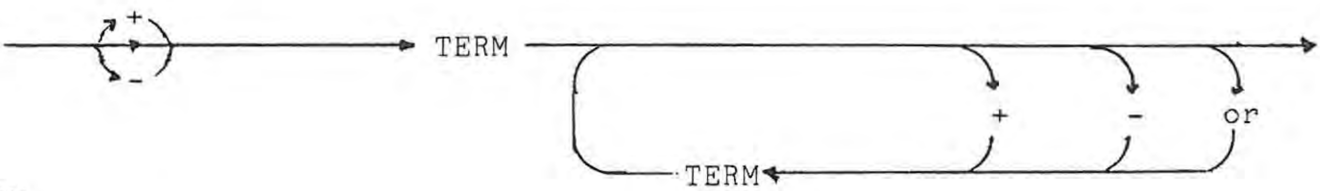
CONDITION



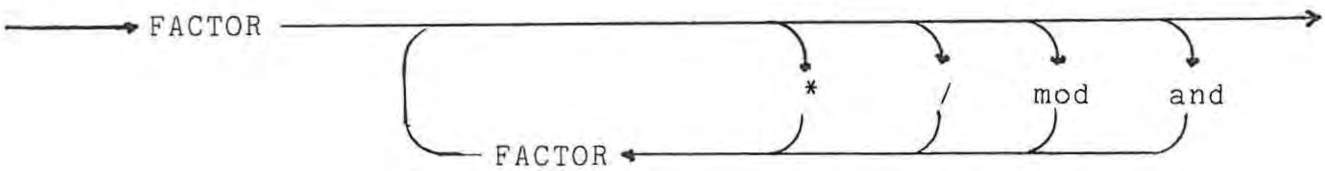
EXPRESSION



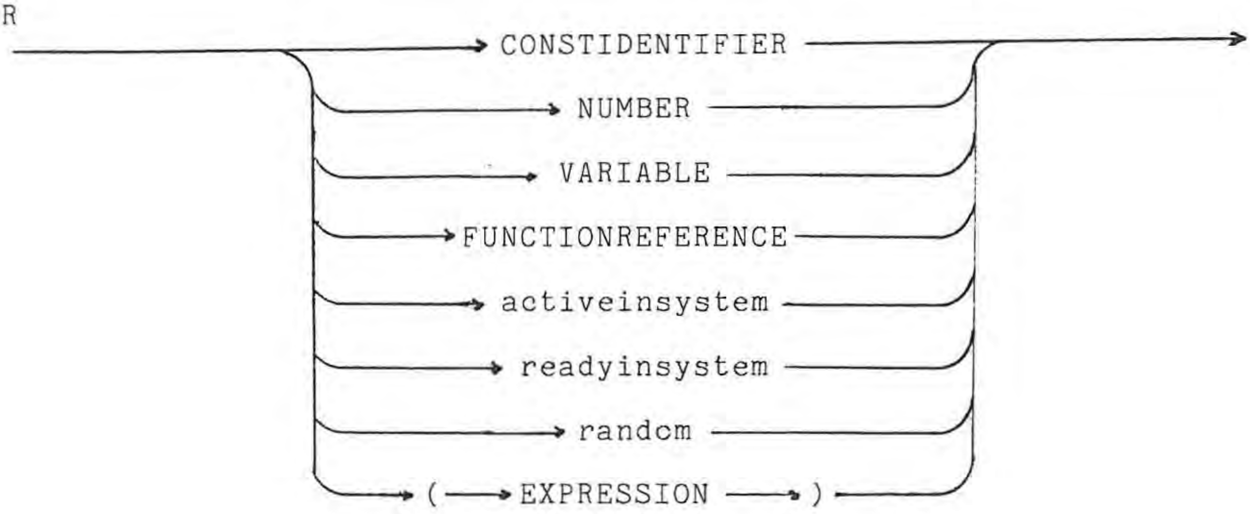
SIMPLEEXPRESSION



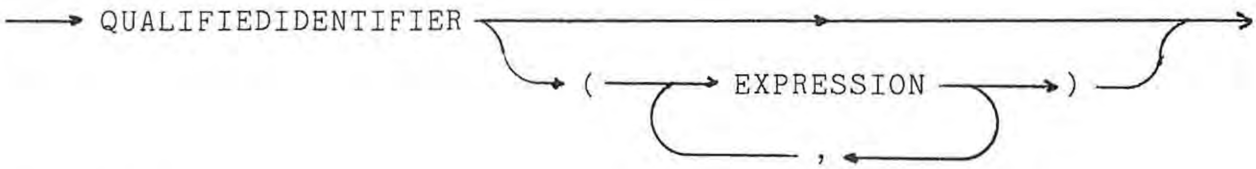
TERM



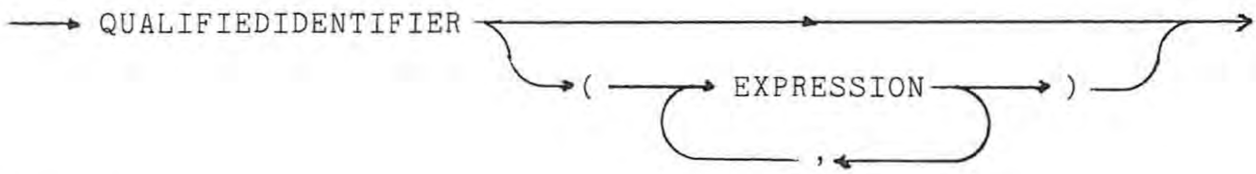
FACTOR



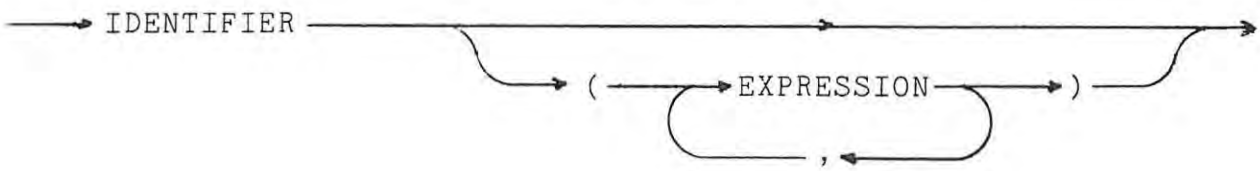
FUNCTIONREFERENCE



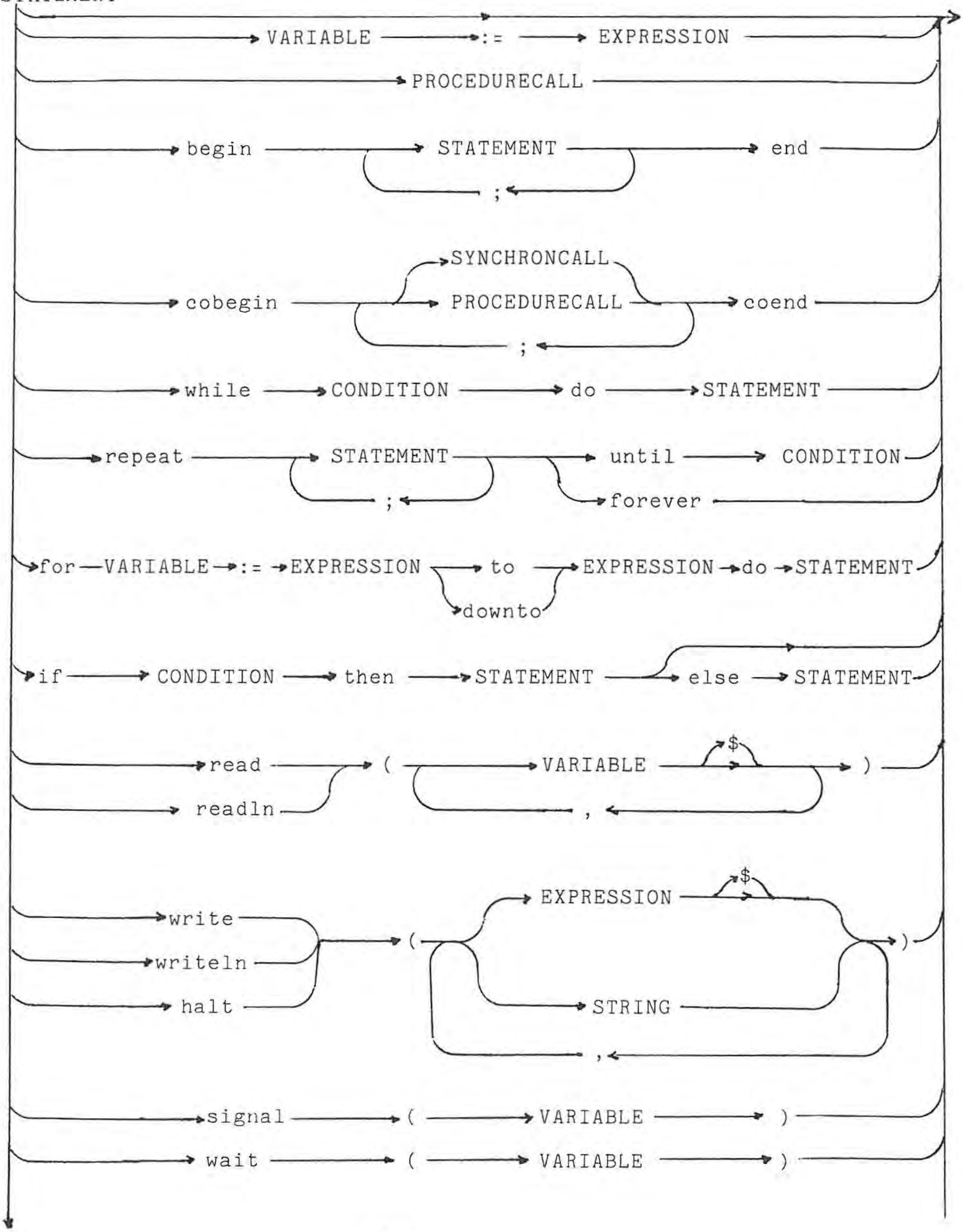
PROCEDURECALL

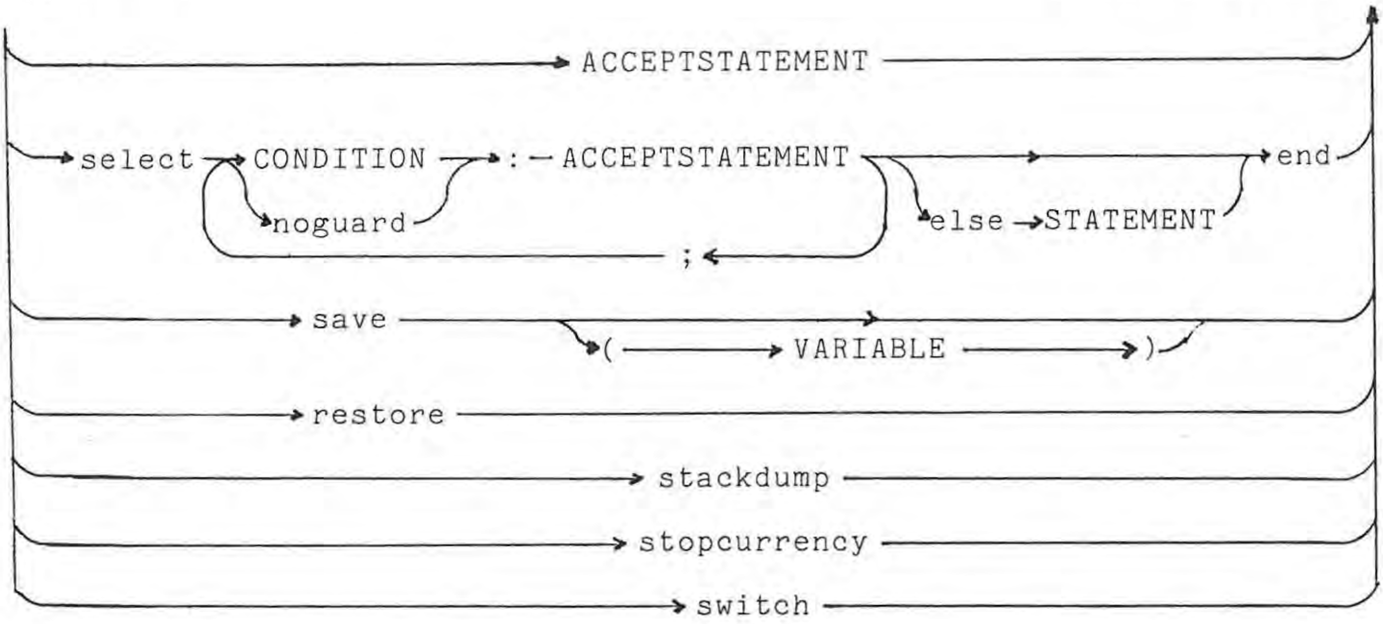


SYNCHRONCALL



STATEMENT


VARIABLE
:=
EXPRESSION
PROCEDURECALL
begin
STATEMENT
end
;
SYNCHRONCALL
cobegin
PROCEDURECALL
coend
;
while
CONDITION
do
STATEMENT
repeat
STATEMENT
until
CONDITION
;
forever
for
VARIABLE
:=
EXPRESSION
to
EXPRESSION
do
STATEMENT
downto
if
CONDITION
then
STATEMENT
else
STATEMENT
read
(
VARIABLE
$
)
readln
,
write
writeln
halt
(
EXPRESSION
$
)
STRING
,
signal
(
VARIABLE
)
wait
(
VARIABLE
)



ACCEPTSTATEMENT

Notes

(1) stackdump allows one to examine the runtime stack. It is used for debugging, but requires a knowledge of the underlying architecture.

(2) random produces a random integer, based on the time of day to seed the sequence. Thus programs using random will not produce the same results each time they run.

(3) It is not at present possible to pass complete arrays as parameters.

(4) Concurrency is introduced by the Cobegin..Coend construct. At present concurrent processes may only be launched from the main program, although they may call upon other procedures thereafter. A concurrent process is defined as a procedure or synchroniser, and may have parameters.

(5) Semaphores are simple integer variables - there is nothing at present to distinguish them from integers, and it is the programmer's responsibility not to abuse them. There are no associated queues.

(6) Recursion is fully supported.

(7) monitor and synchroniser blocks may only be declared in the

main program. It follows that they may not be nested, or contain instances of one another, contrary to what the syntax diagrams might suggest.

(8) Starred identifiers in monitor blocks are accessible outside the monitor in read-only mode, using the "dot" notation. Other identifiers in monitor blocks are totally inaccessible. Within monitor blocks the global variables of the main program block are read-only accessible.

(9) The **$** format descriptors in I/O statements specify whether the item is to be read/written in ASCII or INTEGER mode. Thus, for example, read(A$, B) will read a single character and assign to A the equivalent ASCII value, and will continue to read a single integer and assign it to B.

(10) accept statements may not be nested.

(11) The presence of a noguard option within a select statement renders the else clause redundant.

Reserved words

The list of reserved words is as follows. Those given in (brackets) are not currently used in a reserved sense, but are reserved for possible future extensions, and should probably not be used as identifiers.

accept	activeinsystem	and	begin
(boolean)	(char)	cobegin	coend
condition	const	do	downto
else	end	entry	(false)
for	forever	forward	function
halt	if	(init)	(integer)
mod	monitor	noguard	odd
or	procedure	(process)	program
qlength	qpwait	qsignal	queue
qwait	random	read	readln
readyinsystem	repeat	restore	save
select	(semaphore)	signal	stackdump
stopconcurrency	synchroniser	then	to
(true)	until	(value)	var
wait	while	write	writeln

Compiler directives

(*$S+*)	Suspend process switching for duration of read and write statements.
(*$S-*)	Allow process switching for duration of read and write statements. (DEFAULT)
(*$L+*)	Compiler listing on (DEFAULT)
(*$L-*)	Compiler listing off (except for error messages)
(*$T+*)	Request symbol table dump (*$T-*) suppress it. (DEFAULT)
(*$O+*)	Request object code dump (*$O-*) suppress it. (DEFAULT)
(*$W-*)	Suppress warning messages. (*$W+*) allow them. (DEFAULT)
(*$M+*)	Generate process tracing code for run-time debugging.
(*$M-*)	Suppress process tracing code generation. (DEFAULT)
(*$B+*)	Provide invariance of monitor variables when performing nested monitor calls. (DEFAULT)
(*$B-*)	No guarantee of invariance of monitor variables.

Restrictions

Restrictions	Sage IV
Maximum number of p-codes that can be generated	1500
Maximum number of concurrent processes that can run	10
Maximum number of parameters for any procedure/function	24
Maximum level to which procedures may be nested	5
Maximum number of active identifiers during compilation	100
Maximum memory available for variables in pseudo machine	3500
Maximum of monitors per program	15
Maximum number of condition variables per program	25
Maximum number of entry points per program	25
Maximum of guard conditions per select statement	20
Significant letters in identifiers	8

Arrays may not be passed as parameters

Fatal compilation errors

Program incomplete	Self evident
Symbol table overflow	Limited to 100 accessible
Procedures too deeply nested	Limit is 5
Program too long	Too many p-codes required
Too many parameters	Limited to 24 per procedure

Running instructions on the Apple/Horizon/Advantage/Sirius/Sage IV

1 Obtain a copy of **CLANG21.CODE**.

2 From the command level X(ecute CLANG21.

3 System prompts for names of Listing and Source files. The latter will usually have been prepared with the UCSD E(ditor.

4 After compilation, system prompts for names of Results and Data files. (All these files may default to CONSOLE:)

5 The system allows for repeated execution without recompilation. An executing program may be interrupted with the <ESC> key.

APPENDIX B

LISTING

```
unit TEXTFILES;
(**************************************************************************)
(* Various  machine  dependent, but  useful routines for the Sage IV   *)
(* UCSD  Pascal, developed  by  Pat Terry, 1982.  Version A- 2.Ø ucsd *)
(**************************************************************************)
interface
 var RANDMSD : INTEGER;
     INPUT: FILE;   (* untyped for BLOCKREAD in GETCH*)
 procedure TEXTINPUT( PROMPT: STRING);
 procedure TEXTOUTPUT(var output:text; PROMPT: STRING);
 function KEYPRESS: BOOLEAN;
 function RANDOM: INTEGER;

implementation
 type
  BYTES = Ø..255;
 var
  ALIAS: record
       case BOOLEAN of
        TRUE: (PT: ^INTEGER);
        FALSE: (INT: INTEGER)
      end;

procedure TEXTINPUT (*Open INPUT from console or named file*);
const
 ESCAPE = 27 (*ascii for <esc>*);
var
 FINISHED: BOOLEAN;
 FILENAME: STRING;
begin
 FINISHED := FALSE;
  repeat
    WRITE('What ',PROMPT,' file (<RET> for CONSOLE:
           -<ESC-RET> to abandon)? ');
    READLN(FILENAME);
    if LENGTH(FILENAME)=Ø
     then begin FINISHED := TRUE; RESET(INPUT,'CONSOLE:') end
     else begin
           if (FILENAME[1]=CHR(ESCAPE)) then EXIT(program);
           (*$I- turn off IO-checks *) RESET(INPUT,FILENAME);
           if IORESULT=Ø then FINISHED:=TRUE
             else  if  POS('.text',FILENAME)+POS('.TEXT',FILENAME)=Ø
                    then begin
                          FILENAME:=CONCAT(FILENAME,'.TEXT');
                          RESET(INPUT,FILENAME); FINISHED:=IORESULT=Ø
                         end
          end;
     if not FINISHED then
      begin WRITELN; WRITELN('No such file. Try again') end
    until FINISHED (*$I+ turn IO checks back on*);
 end (*TEXTINPUT*);
```

```
procedure TEXTOUTPUT (*Open OUTPUT to CONSOLE or named file*);
 const
  ESCAPE = 27 (*ascii for <esc>*);
 var
  FINISHED: BOOLEAN;
  FILENAME: STRING;
  CH: CHAR;
  begin
   repeat
    WRITE('What ',PROMPT,' file (<RET> for CONSOLE:
            -<ESC-RET> to abandon)? ');
    FINISHED := TRUE; READLN(FILENAME);
    if LENGTH(FILENAME)=Ø then  FILENAME:='CONSOLE:'
     else if FILENAME[1]='*' then FILENAME := 'PRINTER:' else
          begin
           if (FILENAME[1]=CHR(ESCAPE)) then EXIT(program);
           if  POS('.text',FILENAME)+POS('.TEXT',FILENAME)=Ø
              then FILENAME:=CONCAT(FILENAME,'.TEXT');
           (*$I- turn off IO-checks *) RESET(OUTPUT,FILENAME);
           if IORESULT=Ø
            then begin
                  WRITELN;
                  WRITELN('File already exists - okay to overwrite? ');
                  repeat
                   READ(KEYBOARD,CH)
                  until CH in ['Y','y','N','n'];
                  CLOSE(OUTPUT); FINISHED:=CH in ['Y','y']
                 end
          end
   until FINISHED (*$I+ turn IO checks back on*);
   REWRITE(OUTPUT,FILENAME)
 end (*OPENOUTPUT*);

function KEYPRESS (*check to see whether CONSOLE: is ready*);
var BUF : array [Ø..29] of INTEGER;
begin
  UNITSTATUS(1, BUF, 1); KEYPRESS := BUF[Ø] > Ø
end (*KEYPRESS*);

function RANDOM ;
var
  HIWORD,LOWORD: INTEGER;
begin
  TIME (HIWORD,LOWORD);
  RANDMSD := 259 * RANDMSD + LOWORD mod 56;
  if RANDMSD < Ø then RANDMSD := RANDMSD + MAXINT;
  RANDOM := RANDMSD
end (*RANDOM*);

begin
  RANDMSD := Ø;
end.
```

```
(*$S+*)
unit DECLARATIONS;
(**********************************************************************)
(* For the simple compiler with stack machine code generation *)
(* includes procedures, functions, value parameters and simple arrays *)
(* forward declarations, compound conditions, reference parameters, *)
(* simple concurrency, the monitor concept with starred identifiers *)
(* and condition variables, invariance of monitor variables is *)
(* provided by means of nested backup, save(parameters) and restore; *)
(* and the synchroniser concept including accept statements, and *)
(* select statements with optional else clauses.                     *)
(*                                                                    *)
(* Authors: P.D. Terry and A.G. Chalmers - June 1984   Release 21.2C *)
(**********************************************************************)

interface
uses (*$U :UNIT20A.CODE*) TEXTFILES;
const
  HIGHEST =127;  (*Ascii ord value*)
  LEVMAX = 5;    (*max static nesting*)
  CODEMAX = 1500;(*Max size of code*)
  PMAX = 24;     (*Max number of parameters*)
  PRMAX = 10;    (*concurrent processes*)
  MONMAX = 15;   (*maximum number of monitors*)
  CONDMAX = 25;  (*maximum number of condition variables*)
  DEFAULT = 10;  (*default priority for waiting processes*)
  ENTRYMAX =25;  (*Max no. of entry points per program*)
type
  FCT=(LIT, LDA, CAL, RET, STK, INT, IND, CBG, SFL, EFL, BRN, BZE,
       WGT, SIG, CND, SWP, NEG, ADD, SUB, MUL, DVD, MD , OD , EQL,
       NEQ, LSS, GEQ, GTR, LEQ, STO, HLT, INN, PRN, PRS, NL , LDX,
       RND, PRC, NC , INC, ACT, RDY, SWI, SMK, LMN, EXC, SAV, RES,
       CHK, ACC, EAC, LDE, SCL, SEL, QLN, QPW, QSG, QUE, QWT      );
  INSTRUCTION = packed record
                  F: FCT          (*Function code*);
                  L: 0 .. LEVMAX  (*Level*);
                  A: INTEGER      (*Address*)
                end;

var
  OBCODE, OUTPUT: TEXT;
  CH: CHAR                      (*Last character read*);
  ERRORS,OBLIST: BOOLEAN        (*position of last error*)
  NEXTADDRESS: INTEGER;         (*Code Location Counter*)
  CODE: array [0 .. CODEMAX] of INSTRUCTION (*Generated code*);
  MNEMONIC: array [FCT] of packed array [1..3] of CHAR; (*opcodes*)
  MONICOUNT:INTEGER;            (*For unique monitor number*)
  CONDCOUNT:INTEGER;            (*For unique condition variable number*)
  ENTRYCOUNT:INTEGER;           (*for the unique entry point numbers*)
  NOBACKUP,ASKBACKUP:BOOLEAN    (*automatic nested backup*)
  NOOFLINES:INTEGER;            (*no. of lines compiled*)
  MONIVARADR: array [1..MONMAX,1..2] of INTEGER;
                                (*addresses of monitor variables*)
function BREAKIN : BOOLEAN;
procedure LISTCODE;
```

```
implementation

function BREAKIN;
var CH: CHAR;
begin
  if KEYPRESS
   then begin READ(KEYBOARD,CH); BREAKIN := CH = CHR(27) end
   else BREAKIN := FALSE
end;


procedure LISTCODE;
 var
  I: INTEGER;
 begin
  TEXTOUTPUT (OBCODE,'OBJECT');
  for I := 0 to NEXTADDRESS - 1 do
   begin
    if BREAKIN then EXIT(program);
    with CODE[I] do
     begin
      WRITE(OBCODE, I:10, MNEMONIC[F]:4);
      if (F <= BZE) or (F>=LMN) then WRITE(OBCODE, ' ', L, ' ', A)
     end;
    WRITELN(OBCODE)
   end;
  CLOSE(OBCODE, LOCK)
 end;

end.   (*declarations*)
```

```
(*$S+*)
segment PROGRAMME;
unit COMPILER;
(*for simple concurrent language*)

interface
  uses (*$U :UNIT20A.CODE*) TEXTFILES,
       (*$U :DEC20A.CODE *) DECLARATIONS;
  procedure PROGRAMME;

implementation

procedure PROGRAMME;

const
  LOWEST = 0  (*ASCII ord value*);
  NORW = 52   (*Number of reserved words*);
  TXMAX = 100 (*Length of identifier table*);
  NMAX = 6    (*Max number of digits in numbers*);
  AL = 8      (*Length of identifiers*);
  LEVMAX = 5  (*Max static nesting*);
type
  SYMBOL =
    (NUL,  IDENT,  NUMBER, STRINGSYM, PLUS, MINUS, TIMES, SLASH, DOLLAR,
     ODDSYM,  ANDSYM,  ORSYM,  MODSYM,  EQLSYM,  NEQSYM, LSSSYM, LEQSYM,
     GTRSYM,  GEQSYM,  LPAREN, RPAREN, COMMA, SEMICOLON, PERIOD, LBRACK,
     RBRACK,  COLON, BECOMES, BEGINSYM, ENDSYM, IFSYM, THENSYM, READSYM,
     WHILESYM,  HALTSYM,  REPEATSYM, ELSESYM, UNTILSYM, STACKSYM, DOSYM,
     WRITESYM,  CONSTSYM,  VARSYM,  PROCSYM,  FORWARDSYM, FORSYM, TOSYM,
     DOWNTOSYM,  COBEGINSYM,  COENDSYM,  WAITSYM,  RANDSYM,  FOREVERSYM,
     SIGNALSYM,  CONDSYM,  QLENSYM,  QPWAITSYM,  QSIGNALSYM,  QUEUESYM,
     QWAITSYM,  SYNCSYM,  ENTRYSYM,  ACCEPTSYM,  NOGARDSYM,  SELECTSYM,
     ACTIVESYM, READYSYM, STOPCSYM, SAVESYM, RESTORESYM, SWITCHSYM);

OBJECT = (CONSTANT, VARIABLE, PROG, PROC, FUNC,MONI,CONDVAR,SYNC,EPOINT);
  SYMSET = set of SYMBOL;
  ALFA = packed array [1 .. 8] of CHAR;
  TRANSFERS = (NUMBERS, CHARS, STRINGS, NEWLINE, NEWCARD);
  TYPES = (INTS, BOOLS, NOTYPE);

var
  SYM: SYMBOL                 (*Last symbol read*);
  ID: ALFA                    (*Last identifier read*);
  NUM: INTEGER                (*Last number read*);
  CC: INTEGER                 (*Character pointer*);
  LL: INTEGER                 (*Line length*);
  CS: INTEGER                 (*start of last symbol*);
  ERRPOS: INTEGER             (*position of last error*);
  LISTING, TABLES: BOOLEAN (*Request tables*);
  CLEANIO: BOOLEAN            (*Request READ and WRITE to be indivisible*);
  PROCCALL: BOOLEAN           (*type of last statement*);
  LINE: array [1 .. 81] of CHAR          (*last line read*);
  STRINGTEXT: array [1 .. 80] of CHAR (*last string read*);
  WORD: array [1 .. NORW] of ALFA        (*reserved words*);
  WSYM: array [1 .. NORW] of SYMBOL   (*matching symbols*);
```

```
  SSYM: array [CHAR] of SYMBOL        (*one character symbols*);
  BLOCKBEGSYS, STATBEGSYS, FACBEGSYS, CONSTBEGSYS, RELOPSYS: SYMSET;
  TABLE: array [0 .. TXMAX] of record (*symbol table entries*)
                                 NAME: ALFA;
                                 KIND: OBJECT;
                                 LEVEL: 0 .. LEVMAX;
                                 MIN: INTEGER;
                                 SIZE: INTEGER;
                                 ADR: INTEGER;
                                 CANCHANGE, VARPARAM, DEFINED: BOOLEAN;
                                 REF: packed array [1..PMAX] of BOOLEAN;
                                 ACCESS,INSIDE:BOOLEAN;
                                 UNIQUE:INTEGER; (*monitor number*)
                               end;
  CODEISTOBEGENERATED: BOOLEAN;           (*Listing is not suppressed*)
  NEWGLOBALS:INTEGER;               (*So as not to lose monitor variables *)
  PRESENT,PREVIOUS:INTEGER;             (*For initialisation code sequence*)
  ENDOFMAINVAR:0..TXMAX;               (*Last mainblock var entry in TABLE*)
  STARTOFMAINVAR:0..TXMAX;   (*Table entry for start of main variables*)
  GLOBALADDRESS:INTEGER; (*Monitor variables referenced from main base*)
  MOREMONITORS:BOOLEAN;             (*To update the stack frame correctly*)
  MONCHK, NOWARN:BOOLEAN;                   (*so warnings are suppressed*)
  INMONITOR:BOOLEAN;                          (*no semaphores in monitors*)
  WANTEXCLUSIVITY:BOOLEAN;   (*so as only to ask for exclusivity after
                                          parameters have been loaded*)
  SYNCHRON,                 (*to ensure accepts,etc. in correct places*)
  ISACCEPT,                                 (*to prevent nested accepts*)
  ISELSECASE,                          (*To prevent select in else clause*)
  ISSAVE:BOOLEAN;                          (*for warning if save mmissing*)
  BLOCKNUMBER,BLENGTH:INTEGER;                     (*for indexing buffer*)
  BUFFER:PACKED ARRAY[0..1023] of CHAR;                  (*for new GETCH*)
  DONE:BOOLEAN;                           (*when we have finished reading*)
  MISSRESTORE:BOOLEAN;                  (*to warn that RESTORE is missing*)
  OFFSET:INTEGER;                 (*to determine the offset for each line*)

segment procedure HALT (S: STRING);
begin
  WRITELN; WRITELN ('Halted ',S); EXIT(program)
end (*HALT*);

(* +++++++++++++++++++++++++ Source handler +++++++++++++++++++++++ *)

segment procedure ERROR(ERRORCODE: INTEGER);
 var
   I: INTEGER;
 procedure ERR1;
  begin
   case ERRORCODE of
    0: WRITE(OUTPUT,'OUT OF RANGE');
    1: WRITE(OUTPUT,'STRING TOO LONG');
    2: WRITE(OUTPUT,'; EXPECTED');
    3: WRITE(OUTPUT,'INVALID SEQUENCE');
    4: WRITE(OUTPUT,'REDECLARED');
    5: WRITE(OUTPUT,'UNDECLARED');
    6: WRITE(OUTPUT,'IDENTIFIER EXPECTED');
```

```
    7: WRITE(OUTPUT,':= WRONG CONTEXT');
    8: WRITE(OUTPUT,'NUMBER EXPECTED');
    9: WRITE(OUTPUT,'= EXPECTED');
    10: WRITE(OUTPUT,'] EXPECTED');
    11: WRITE(OUTPUT,'UNEXPECTED SUBSCRIPT');
    12: WRITE(OUTPUT,'WRONG NUMBER OF PARAMETERS');
    13: WRITE(OUTPUT,', OR ) EXPECTED');
    14: WRITE(OUTPUT,'INVALID START TO FACTOR');
    15: WRITE(OUTPUT,'[ EXPECTED');
    16: WRITE(OUTPUT,'INVALID PROCEDURE REFERENCE');
    17: WRITE(OUTPUT,') EXPECTED');
    18: WRITE(OUTPUT,'( EXPECTED');
    19: WRITE(OUTPUT,': EXPECTED');
    20: WRITE(OUTPUT,'INVALID ASSIGNMENT');
    21: WRITE(OUTPUT,':= EXPECTED');
    22: WRITE(OUTPUT,'INVALID REFERENCE');
    23: WRITE(OUTPUT,'THEN EXPECTED');
    24: WRITE(OUTPUT,'END EXPECTED');
    25: WRITE(OUTPUT,'DO EXPECTED');
    26: WRITE(OUTPUT,'UNTIL EXPECTED');
    end (*case*)
 end (*ERR1*);

 procedure ERR2;
  begin
   case ERRORCODE of
    27: WRITE(OUTPUT,'INVALID FORMAT DESCRIPTOR');
    28: WRITE(OUTPUT,'CANNOT READ');
    29: WRITE(OUTPUT,'INVALID CONSTANT');
    30: WRITE(OUTPUT,': EXPECTED');
    31: WRITE(OUTPUT,'INVALID SUBRANGE');
    32: WRITE(OUTPUT,'INVALID SYMBOL AFTER A STATEMENT');
    33: WRITE(OUTPUT,'TYPE CONFLICT');
    34: WRITE(OUTPUT,'BEGIN EXPECTED');
    35: WRITE(OUTPUT,'INVALID SYMBOL AFTER BLOCK');
    36: WRITE(OUTPUT,'PROGRAM EXPECTED');
    37: WRITE(OUTPUT,'. EXPECTED');
    38: WRITE(OUTPUT,'DISAGREES WITH EARLIER LIST');
    39: WRITE(OUTPUT,'CANNOT ALTER - READ ONLY');
    40: WRITE(OUTPUT,'DECLARED AT WRONG LEVEL');
    41: WRITE(OUTPUT,'TO OR DOWNTO EXPECTED');
    42: WRITE(OUTPUT,'ONLY PROCEDURE CALLS ALLOWED');
    43: WRITE(OUTPUT,'COEND EXPECTED');
    44: WRITE(OUTPUT,'CONCURRENCY ONLY IN MAIN PROGRAM');
    45: WRITE(OUTPUT,'TOO MANY CONCURRENT PROCESSES');
    46: WRITE(OUTPUT,'MONITORS IN MAINBLOCK ONLY');
    47: WRITE(OUTPUT,'STARRED IDENTIFIERS ONLY IN MONITORS');
    48: WRITE(OUTPUT,'ONLY STARRED IDENTIFIERS ACCESSIBLE');
    50: WRITE(OUTPUT,'TOO MANY MONITOR DECLARATIONS');
    51: WRITE(OUTPUT,'CONDITION VARIABLES ONLY IN MONITORS');
    52: WRITE(OUTPUT,'INCORRECT CONDITION VARIABLE USAGE');
    53: WRITE(OUTPUT,'TOO MANY CONDITION VARIABLES');
    54: WRITE(OUTPUT,'NO SEMAPHORES IN MONITORS');
    55: WRITE(OUTPUT,'ONLY CURRENT MONITOR VARIABLES MAY BE SAVED');
    57: WRITE(OUTPUT,'SAVE/RESTORE ONLY IN MONITOR PROC/FUNC');
```

```
    60: WRITE(OUTPUT,'SYNCHRONISERS ONLY IN MAINBLOCK');
    61: WRITE(OUTPUT,'ENTRY POINTS ONLY IN SYNCRONISERS');
    62: WRITE(OUTPUT,'TOO MANY ENTRY POINTS');
    63: WRITE(OUTPUT,'ONLY ENTRY POINTS ACCESSABLE');
    64: WRITE(OUTPUT,'ENTRY POINT CALL IN ILLEGAL POSITION');
    65: WRITE(OUTPUT,'NO NESTED SYNCHRONISERS');
    66: WRITE(OUTPUT,'ENTRY POINT EXPECTED');
    67: WRITE(OUTPUT,'NO NESTED ACCEPT STATEMENTS');
    68: WRITE(OUTPUT,'SELECT ONLY IN SYNCHRONISER');
    69: WRITE(OUTPUT,'ACCEPT EXPECTED');
    70: WRITE(OUTPUT,'TOO MANY GUARD CONDITIONS');
    71: WRITE(OUTPUT,'SELECT STATEMENT IN ILLEGAL POSITION');
   end (*case*);
  end (*ERR2*);

begin (*ERROR*)
 ERRORS := TRUE; CODEISTOBEGENERATED := FALSE;
 if CS <> ERRPOS then
  begin
   if not LISTING then
    begin
     write(OUTPUT,'      ');
     for I := 1 to LL do WRITE(OUTPUT,LINE[I]); WRITELN(OUTPUT)
    end;
   WRITE(OUTPUT,'**** ', '^': CS+1+OFFSET);
   if ERRORCODE < 27 then ERR1 else ERR2;
   WRITELN(OUTPUT); ERRPOS := CS
  end
end (*ERROR*);

(* Include  files  *)
(*$I :DEC220A.TEXT *)
(*$I :COM20A.TEXT  *)
(*$I :COM220A.TEXT *)
(*$I :COM320A.TEXT *)
```

```
procedure GETCH;

  procedure READNEXTBLOCK;
   begin
    DONE:=BLOCKREAD(INPUT,BUFFER,2,BLOCKNUMBER)=0;
    BLOCKNUMBER:=BLOCKNUMBER+2;   (*read in two blocks at a time*)
   end;

begin
  if CC = LL
  then
    begin (*new line*)
      LL := 0; CC := 0; CS := 0; ERRPOS := -1; OFFSET:=0;
      NOOFLINES:=NOOFLINES+1; (*no. of lines compiled*)
      if LISTING then WRITE(OUTPUT, NEXTADDRESS:5, ' ');
      if BLENGTH=0 then READNEXTBLOCK;
      repeat
       if (BREAKIN) then HALT('IC');
       if (ord(BUFFER[BLENGTH])=16 (*DLE*)) then
        begin   (*offset left margin*)
         BLENGTH:=BLENGTH+1;
         OFFSET:=(ord(BUFFER[BLENGTH])-32);
         WRITE(OUTPUT,' ':OFFSET);
        end
       else
        begin
         if (ord(BUFFER[BLENGTH]) >=32) and (ord(BUFFER[BLENGTH]) <=126)
          then begin
           LL:=LL+1;
           LINE[LL]:=BUFFER[BLENGTH];
           if LISTING then WRITE(OUTPUT,LINE[LL]);
          end;
         end;   (*else*)
      BLENGTH:=BLENGTH+1;
      if BLENGTH > 1023 then
       begin
        BLENGTH:=0; READNEXTBLOCK;
       end;
     until ord(BUFFER[BLENGTH])=13;
     if LISTING then WRITELN(OUTPUT);
     LL:=LL+1;   (*get passed EOLN*)
     LINE[LL]:=' ';
     if BLENGTH > 1023 then BLENGTH:=0;
    end;   (*newline*)
  CC:=CC+1; CH:=LINE[CC];
 end;   (*GETCH*)
```

```
(* +++++++++++++++++++++++ Lexical Analyser +++++++++++++++++++++++ *)

procedure GETSYM;
var
  I, J, K, DEPTH: INTEGER;
  FOUND, ENDSTRING: BOOLEAN;

 function NOTLETTER: BOOLEAN;
   begin NOTLETTER := not(CH in ['A'..'Z','a'..'z']) end (*NOTLETTER*);

 function NOTDIGIT: BOOLEAN;
   begin NOTDIGIT := (CH < 'Ø') or (CH > '9') end (*NOTDIGIT*);

 function DIGIT: INTEGER;
   begin DIGIT := ORD(CH) - ORD('Ø') end (*DIGIT*);

 procedure OPTIONS;
   begin
    GETCH;
    case CH of
     'S','s' : begin GETCH; CLEANIO := CH = '+' end;
     'T','t' : begin GETCH; TABLES := CH = '+' end;
     'L','l' : begin GETCH; LISTING := CH = '+' end;
     'O','o' : begin GETCH; OBLIST := CH = '+' end;
     'W','w' : begin GETCH; NOWARN := CH = '-'  end;
     'M','m' : begin GETCH; MONCHK := CH = '+' end;
     'B','b' : begin
                GETCH; NOBACKUP := CH = '-'; ASKBACKUP:=NOBACKUP;
               end;
    end (*case*); GETCH
   end (*OPTIONS*);

 begin (*GETSYM*)
 repeat
   while CH = ' ' do GETCH (*Skip blanks*);
   FOUND := TRUE; CS := CC (*for error reporting*);
   SYM := SSYM[CH];
   case CH of
     'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M',
     'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z',
     'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p',
     'q','r','s','t','u','v','w','x','y','z':
       begin (*Identifier or reserved word*)
         K := 1; ID := '        ';
         repeat
           if CH in ['a'..'z'] then CH:= CHR(ORD(CH)-ORD('a')+ORD('A'));
           if K <= AL then begin ID[K] := CH; K := K + 1 end; GETCH
         until NOTLETTER and NOTDIGIT;
         I := 1; J := NORW;
         repeat (*Binary search*)
           K := (I + J) DIV 2;
           if ID <= WORD[K] then J := K - 1;
           if ID >= WORD[K] then I := K + 1
         until I > J;
```

```
        if I - 1 > J then SYM := WSYM[K] else SYM := IDENT
      end;
    'Ø', '1', '2', '3', '4', '5', '6', '7', '8', '9':
      begin (*number*)
        K := Ø; NUM := Ø; SYM := NUMBER;
        repeat
          if K <= NMAX then NUM := 1Ø * NUM + DIGIT; GETCH; K := K + 1
        until NOTDIGIT;
        if K > NMAX then ERROR(Ø)
      end ;
    ':':
      begin
         GETCH;
         if CH = '=' then begin SYM := BECOMES; GETCH end
         else SYM := COLON
       end;
     '<':
       begin
         GETCH;
         if CH = '=' then begin SYM := LEQSYM; GETCH end
         else
           if CH = '>' then begin SYM := NEQSYM; GETCH end
           else SYM := LSSSYM
       end;
     '>':
       begin
         GETCH;
         if CH = '=' then begin SYM := GEQSYM; GETCH end
         else SYM := GTRSYM
       end;
     '''':
       begin (*String*)
         NUM := Ø; GETCH; SYM := STRINGSYM; ENDSTRING := FALSE;
         repeat
           if CH = '''' then begin GETCH; ENDSTRING := CH <> '''' end;
           if not ENDSTRING then
             begin NUM := NUM + 1; STRINGTEXT[NUM] := CH; GETCH end
         until ENDSTRING or (CC = LL);
         if CC = LL then begin NUM := Ø; ERROR(1) end
       end;
     '(':
       begin
         GETCH;
         if CH = '*' then
           begin (*ignore comments (even nested) *)
             DEPTH := 1; FOUND := FALSE; GETCH;
             if CH = '$' then OPTIONS;
             repeat
               if CH = ';' then WRITELN(OUTPUT,'^':CC+6,'; INTENDED?');
               if CH = '*' then
                 begin (*end of comment?*)
                  GETCH;
                  if CH = ')' then begin DEPTH := DEPTH-1; GETCH end
                 end
```

```
               else
                 if CH = '(' then
                   begin (*nested comment?*)
                    GETCH;
                    if CH = '*' then begin DEPTH := DEPTH+1; GETCH end
                   end
                 else GETCH
             until DEPTH = 0
           end
         else SYM := LPAREN
       end;
      '*', '+', '-', '#', '=', '/', ')', '[', ']', ',', ';', '^', '&',
      '@', '$', '.', '\', ' ', '%', '?', '"', '!':
        (*Implementation defined*)
        begin SYM := SSYM[CH]; GETCH end;
    end (*case*);
  until FOUND
end (*GETSYM*);

procedure INITIALISE;
var
  C: CHAR;

procedure RESERVREST;
 begin
   WSYM[ 1]:= ACCEPTSYM ; WSYM[ 2]:= ACTIVESYM ; WSYM[ 3]:= ANDSYM     ;
   WSYM[ 4]:= BEGINSYM  ; WSYM[ 5]:= COBEGINSYM; WSYM[ 6]:= COENDSYM   ;
   WSYM[ 7]:= CONDSYM   ; WSYM[ 8]:= CONSTSYM  ; WSYM[ 9]:= DOSYM      ;
   WSYM[10]:= DOWNTOSYM ; WSYM[11]:= ELSESYM   ; WSYM[12]:= ENDSYM     ;
   WSYM[13]:= ENTRYSYM  ; WSYM[14]:= FORSYM    ; WSYM[15]:= FOREVERSYM;
   WSYM[16]:= FORWARDSYM; WSYM[17]:= PROCSYM   ; WSYM[18]:= HALTSYM    ;
   WSYM[19]:= IFSYM     ; WSYM[20]:= MODSYM    ; WSYM[21]:= PROCSYM    ;
   WSYM[22]:= NOGARDSYM ; WSYM[23]:= ODDSYM    ; WSYM[24]:= ORSYM      ;
   WSYM[25]:= PROCSYM   ; WSYM[26]:= PROCSYM   ; WSYM[27]:= QLENSYM    ;
   WSYM[28]:= QPWAITSYM ; WSYM[29]:= QSIGNALSYM; WSYM[30]:= QUEUESYM   ;
   WSYM[31]:= QWAITSYM  ; WSYM[32]:= RANDSYM   ; WSYM[33]:= READSYM    ;
   WSYM[34]:= READSYM   ; WSYM[35]:= READYSYM  ; WSYM[36]:= REPEATSYM  ;
   WSYM[37]:= RESTORESYM; WSYM[38]:= SAVESYM   ; WSYM[39]:= SELECTSYM  ;
   WSYM[40]:= SIGNALSYM ; WSYM[41]:= STACKSYM  ; WSYM[42]:= STOPCSYM   ;
   WSYM[43]:= SWITCHSYM ; WSYM[44]:= SYNCSYM   ; WSYM[45]:= THENSYM    ;
   WSYM[46]:= TOSYM     ; WSYM[47]:= UNTILSYM  ; WSYM[48]:= VARSYM     ;
   WSYM[49]:= WAITSYM   ; WSYM[50]:= WHILESYM  ; WSYM[51]:= WRITESYM   ;
   WSYM[52]:= WRITESYM  ;
 end;

 procedure RESERVEDWORDS;
  begin
   WORD[ 1]:= 'ACCEPT   '; WORD[ 2]:= 'ACTIVEIN'; WORD[ 3]:= 'AND      ';
   WORD[ 4]:= 'BEGIN    '; WORD[ 5]:= 'COBEGIN '; WORD[ 6]:= 'COEND    ';
   WORD[ 7]:= 'CONDITIO'; WORD[ 8]:= 'CONST    '; WORD[ 9]:= 'DO       ';
   WORD[10]:= 'DOWNTO   '; WORD[11]:= 'ELSE     '; WORD[12]:= 'END      ';
   WORD[13]:= 'ENTRY    '; WORD[14]:= 'FOR      '; WORD[15]:= 'FOREVER ';
   WORD[16]:= 'FORWARD '; WORD[17]:= 'FUNCTION'; WORD[18]:= 'HALT     ';
   WORD[19]:= 'IF       '; WORD[20]:= 'MOD      '; WORD[21]:= 'MONITOR ';
   WORD[22]:= 'NOGUARD '; WORD[23]:= 'ODD      '; WORD[24]:= 'OR       ';
```

```
    WORD[25]:= 'PROCEDUR'; WORD[26]:= 'PROGRAM '; WORD[27]:= 'QLENGTH ';
    WORD[28]:= 'QPWAIT  '; WORD[29]:= 'QSIGNAL '; WORD[30]:= 'QUEUE   ';
    WORD[31]:= 'QWAIT   '; WORD[32]:= 'RANDOM  '; WORD[33]:= 'READ    ';
    WORD[34]:= 'READLN  '; WORD[35]:= 'READYINS'; WORD[36]:= 'REPEAT  ';
    WORD[37]:= 'RESTORE '; WORD[38]:= 'SAVE    '; WORD[39]:= 'SELECT  ';
    WORD[40]:= 'SIGNAL  '; WORD[41]:= 'STACKDUM'; WORD[42]:= 'STOPCONC';
    WORD[43]:= 'SWITCH  '; WORD[44]:= 'SYNCHRON'; WORD[45]:= 'THEN    ';
    WORD[46]:= 'TO      '; WORD[47]:= 'UNTIL   '; WORD[48]:= 'VAR     ';
    WORD[49]:= 'WAIT    '; WORD[50]:= 'WHILE   '; WORD[51]:= 'WRITE   ';
    WORD[52]:= 'WRITELN ';
   RESERVREST;

  end (*RESERVEDWORDS*);

  procedure OPCODES;
   begin
     MNEMONIC[LIT]:= 'LIT'; MNEMONIC[LDA]:= 'LDA'; MNEMONIC[CAL]:= 'CAL';
     MNEMONIC[INT]:= 'INT'; MNEMONIC[BRN]:= 'BRN'; MNEMONIC[BZE]:= 'BZE';
     MNEMONIC[IND]:= 'IND'; MNEMONIC[RET]:= 'RET'; MNEMONIC[NEG]:= 'NEG';
     MNEMONIC[ADD]:= 'ADD'; MNEMONIC[SUB]:= 'SUB'; MNEMONIC[MUL]:= 'MUL';
     MNEMONIC[DVD]:= 'DVD'; MNEMONIC[ MD]:= 'MOD'; MNEMONIC[ OD]:= 'ODD';
     MNEMONIC[EQL]:= 'EQL'; MNEMONIC[NEQ]:= 'NEQ'; MNEMONIC[LSS]:= 'LSS';
     MNEMONIC[GEQ]:= 'GEQ'; MNEMONIC[GTR]:= 'GTR'; MNEMONIC[LEQ]:= 'LEQ';
     MNEMONIC[STK]:= 'STK'; MNEMONIC[STO]:= 'STO'; MNEMONIC[HLT]:= 'HLT';
     MNEMONIC[INN]:= 'INN'; MNEMONIC[PRN]:= 'PRN'; MNEMONIC[PRS]:= 'PRS';
     MNEMONIC[NL] := 'NL '; MNEMONIC[LDX]:= 'LDX'; MNEMONIC[SWP]:= 'SWP';
     MNEMONIC[SFL]:= 'SFL'; MNEMONIC[EFL]:= 'EFL'; MNEMONIC[CBG]:= 'CBG';
     MNEMONIC[CND]:= 'CND'; MNEMONIC[WGT]:= 'WGT'; MNEMONIC[SIG]:= 'SIG';
     MNEMONIC[RND]:= 'RND'; MNEMONIC[PRC]:= 'PRC'; MNEMONIC[NC ]:= 'NC ';
     MNEMONIC[INC]:= 'INC'; MNEMONIC[LMN]:= 'LMN'; MNEMONIC[EXC]:= 'EXC';
     MNEMONIC[QLN]:= 'QLN'; MNEMONIC[QPW]:= 'QPW'; MNEMONIC[QSG]:= 'QSG';
     MNEMONIC[QUE]:= 'QUE'; MNEMONIC[QWT]:= 'QWT'; MNEMONIC[CHK]:= 'CHK';
     MNEMONIC[ACC]:= 'ACC'; MNEMONIC[EAC]:= 'EAC'; MNEMONIC[SCL]:= 'SCL';
     MNEMONIC[LDE]:= 'LDE'; MNEMONIC[SEL]:= 'SEL'; MNEMONIC[ACT]:= 'ACT';
     MNEMONIC[RDY]:= 'RDY'; MNEMONIC[RES]:= 'RES'; MNEMONIC[SAV]:= 'SAV';
     MNEMONIC[SMK]:= 'SMK'; MNEMONIC[SWI]:= 'SWI';
    end (*OPCODES*);

  begin (*INITIALISE*)
     WRITELN(OUTPUT);
     WRITELN(OUTPUT, 'Toy Compiler Mark 21.2C m cv s spr nb');
     WRITELN(OUTPUT);
     RESERVEDWORDS; OPCODES;
     for C := CHR(LOWEST) to CHR(HIGHEST) do SSYM[C] := NUL;
     SSYM['+'] := PLUS    ; SSYM['-'] := MINUS  ; SSYM['*'] := TIMES;
     SSYM['/'] := SLASH   ; SSYM['('] := LPAREN ; SSYM[')'] := RPAREN;
     SSYM['='] := EQLSYM  ; SSYM[','] := COMMA  ; SSYM['.'] := PERIOD;
     SSYM['<'] := LSSSYM  ; SSYM['>'] := GTRSYM ; SSYM[';'] := SEMICOLON;
     SSYM['['] := LBRACK  ; SSYM[']'] := RBRACK ; SSYM[':'] := COLON;
     SSYM['$'] := DOLLAR  ;

     RELOPSYS := [EQLSYM, NEQSYM, GTRSYM, GEQSYM, LSSSYM, LEQSYM];
     BLOCKBEGSYS := [CONSTSYM, VARSYM, CONDSYM, PROCSYM, BEGINSYM,
                     FORWARDSYM, ENTRYSYM, SYNCSYM];
```

```
    STATBEGSYS := [IDENT, BEGINSYM, IFSYM, WHILESYM, REPEATSYM, HALTSYM,
                   FORSYM,  COBEGINSYM,  WAITSYM,  SIGNALSYM,  WRITESYM,
                   READSYM,      STACKSYM,      ACCEPTSYM,      SELECTSYM,
                   STOPCSYM,SAVESYM, RESTORESYM];

    FACBEGSYS := [IDENT, NUMBER, LPAREN, RANDSYM,ACTIVESYM,READYSYM];

    CONSTBEGSYS := [PLUS, MINUS, IDENT, NUMBER];

    LISTING := TRUE; OBLIST := FALSE; TABLES := FALSE;
    ERRORS := FALSE; CLEANIO := FALSE;  NOWARN:=FALSE;
    MONCHK := FALSE; NOBACKUP := FALSE; ASKBACKUP:=NOBACKUP;

    (************Initialise code generator**********)
    NEXTADDRESS := Ø; CODEISTOBEGENERATED := TRUE;

    (****************** Initialise lexical analyser ******************)
    CC := Ø; LL := Ø; ERRPOS := Ø; CH := ' '; DONE:=FALSE;NOOFLINES:=Ø;
    BLENGTH:=Ø; BLOCKNUMBER:=2;   (*skip passed header information*)

    GETSYM;

    PRESENT:=Ø;              NEWGLOBALS:=Ø;           STARTOFMAINVAR:=2;
    ENDOFMAINVAR:=1;         MOREMONITORS:=FALSE;     MONICOUNT:=Ø;
    CONDCOUNT:=Ø;            INMONITOR:=FALSE;        SYNCHRON:=FALSE;
    ENTRYCOUNT:=Ø;           WANTEXCLUSIVITY:=FALSE;ISACCEPT:=FALSE;
    ISELSECASE:=FALSE;       ISSAVE := FALSE;         MISSRESTORE :=FALSE;
    PROCCALL:=FALSE;
 end (*INITIALISE*);
```

```
(* ++++++++++++++++++++++++++++ Analyser ++++++++++++++++++++++++++++ *)

procedure ACCEPT(EXPECTED: SYMBOL; ERRORCODE: INTEGER);
  begin
   if SYM = EXPECTED then GETSYM else ERROR(ERRORCODE)
  end (*ACCEPT*);

procedure BLOCK(FOLLOWERS: SYMSET; LEV,TX: INTEGER; BLOCKKIND: OBJECT;
                COMPLETING: BOOLEAN; BLOCKENTRY: INTEGER);

  var
    STARTBLOCK: INTEGER   (*Start address*);
    ADDRESS: INTEGER      (*Variable address index*);
    I,TX0: INTEGER        (*Initial symbol table entry*);
    PARAMS: INTEGER       (*Number of Parameters*);
    STAR:BOOLEAN;         (*Used for starred identifiers*)
    SIZEREQUIRED:INTEGER; (*get extra stack space for select statement*)

(* ++++++++++++++++++++++++++ Code Generator ++++++++++++++++++++++++ *)

   procedure GEN (X: FCT; Y,Z: INTEGER);
   (*Code generator*)
    begin
     if NEXTADDRESS > CODEMAX then HALT('LL');
     if CODEISTOBEGENERATED
     then
      begin
       with CODE[NEXTADDRESS] do begin F := X; L := Y; A := Z end;
       NEXTADDRESS := NEXTADDRESS + 1
      end
    end;

   procedure EMIT(X: FCT);
   (*code generator with no address field*)
    begin GEN(X, 0, 0) end;

   procedure OBTAINEXCLUSIVITY(U:INTEGER);
    begin GEN(EXC,0,U); end;

   procedure LEAVINGMONITOR(U:INTEGER);
    begin GEN(LMN,0,U); end;

   procedure CONDVARCODE(X:FCT; B:INTEGER);
    begin GEN(X,0,B); end;

   procedure EMITACCEPT(U:INTEGER);
    begin GEN(ACC,0,U); end;

   procedure EMITENDACCEPT(OFFSET,ADR:INTEGER);
    begin GEN(EAC,OFFSET,ADR); end;

   procedure ENTRYPARAMETER(OFFSET,ADR:INTEGER);
   (*load address for entry point parameter*)
    begin GEN(LDE,OFFSET,ADR); end;
```

```
procedure SYNCCALL(OFFSET,ADR:INTEGER);
 (*calling an entry point*)
 begin GEN(SCL,OFFSET,ADR); end;

procedure EMITSELECT(OFFSET,ADR:INTEGER);
 begin GEN(SEL,OFFSET,ADR); end;

procedure SAVEVARIABLES(U:INTEGER);
 begin GEN(SAV,Ø,U); end;

procedure RESTOREVARIABLES(U:INTEGER);
 begin GEN(RES,Ø,U); end;

procedure SAVEMARKER;
 begin EMIT(SMK); end;

procedure NEGATEINTEGER;
 begin EMIT(NEG) end;

procedure BINARYINTEGEROP(OP: SYMBOL);
 begin
  case OP of
   TIMES: EMIT(MUL);
   SLASH: EMIT(DVD);
   PLUS:  EMIT(ADD);
   MINUS: EMIT(SUB);
   MODSYM: EMIT(MD)
  end
 end;

procedure BINARYBOOLEANOP(OP: SYMBOL);
 begin
  case OP of
   ANDSYM:EMIT(MUL);
   ORSYM: EMIT(ADD);
  end
 end;

procedure COMPARISON(OP: SYMBOL);
 begin
  case OP of
   EQLSYM: EMIT(EQL);
   NEQSYM: EMIT(NEQ);
   LSSSYM: EMIT(LSS);
   LEQSYM: EMIT(LEQ);
   GTRSYM: EMIT(GTR);
   GEQSYM: EMIT(GEQ)
  end
 end;
```

```
procedure INPUTOPERATION(OP: TRANSFERS);
 begin
  case OP of
   NUMBERS: EMIT(INN);
   STRINGS,NEWLINE: (*not used*);
   CHARS: EMIT(INC);
   NEWCARD: EMIT(NC);
  end
 end;

procedure STACKSTRING;
 var I: INTEGER;
 begin
  for I := 1 to NUM do GEN(LIT, 0, ORD(STRINGTEXT[I]));
  GEN(LIT, 0, NUM)
 end;

procedure OUTPUTOPERATION(OP: TRANSFERS);
 begin
  case OP of
   STRINGS: begin STACKSTRING; EMIT(PRS) end;
   NUMBERS: EMIT(PRN);
   NEWLINE: EMIT(NL);
   CHARS: EMIT(PRC);
   NEWCARD: ;
  end
 end;

procedure STACKCONSTANT(NUM: INTEGER);
 begin GEN(LIT, 0, NUM) end;

procedure STACKADDRESS(OFFSET, ADR: INTEGER);
 begin GEN(LDA, OFFSET, ADR) end;

procedure DEREFERENCE;
 begin EMIT(LDX) end;

procedure SUBSCRIPT(LIMIT: INTEGER);
 begin GEN(IND, 0, LIMIT-1) end;

procedure ASSIGN;
 begin EMIT(STO) end;

procedure OPENSTACKFRAME(SIZE: INTEGER);
 begin GEN(INT, 0, SIZE) end;

procedure STORELABEL(var LAB: INTEGER);
 begin LAB := NEXTADDRESS end;

procedure JUMP(LAB: INTEGER);
 begin GEN(BRN, 0, LAB) end;

procedure JUMPONFALSE(LAB: INTEGER);
 begin GEN(BZE, 0, LAB) end;
```

```
procedure STARTFORLOOP (UP: BOOLEAN);
 begin if UP then GEN(SFL, 0, 0) else GEN(SFL, 2, 0) end;

procedure ENDFORLOOP (UP: BOOLEAN; LAB: INTEGER);
 begin if UP then GEN(EFL, 0, LAB) else GEN(EFL, 2, LAB) end;

procedure STARTPROCESSES;
 begin EMIT(CBG) end;

procedure STOPPROCESSES;
 begin EMIT(CND) end;

procedure CODEFORSIGNAL;
 begin EMIT(SIG) end;

procedure CODEFORWAIT;
 begin EMIT(WGT) end;

procedure CODEFORRANDOM;
 begin EMIT(RND) end;

procedure CODEFORACTIVE;
 begin EMIT(ACT) end;

procedure RDYCODE;
 begin EMIT(RDY) end;

procedure TOGGLESWITCHING;
 begin EMIT(SWP) end;

procedure PROCESSTRACE;
 begin EMIT(CHK) end;

procedure PROCSWITCH;
 begin EMIT(SWI) end;

procedure ENTERBLOCK(var LAB: INTEGER);
 begin STORELABEL(LAB); JUMP(0) end;

procedure LEAVEBLOCK(BLOCKKIND: OBJECT; LEV: INTEGER;
                      PARAMS: INTEGER);
 begin
  case BLOCKKIND of
   PROG: EMIT(HLT);
   FUNC,PROC,SYNC: GEN(RET, LEV, PARAMS+1);
   MONI:JUMP(STARTBLOCK);
  end
 end;

procedure CALL(OFFSET, ADR: INTEGER);
 begin GEN(CAL, OFFSET, ADR) end;

procedure CODEFORODD;
 begin EMIT(OD) end;
```

```
    procedure CODEFORDUMP(LEVEL: INTEGER);
     begin GEN(STK, Ø, LEVEL) end;

    procedure BACKPATCH(LOCATION, ADR: INTEGER);
     begin CODE[LOCATION].A := ADR end;


(* ++++++++++++++++++++++++ DECLARATIONS PART ++++++++++++++++++++ *)

    procedure TEST(ALLOWED, BEACONS: SYMSET; ERRORCODE: INTEGER);
      begin
        if not (SYM in ALLOWED) then
          begin
            ERROR(ERRORCODE);
            while not (SYM in ALLOWED + BEACONS) do GETSYM
          end
     end (*TEST*);

    procedure SKIP(EXCESS:SYMBOL);
     (*to skip passed excess symbols*)
     begin
      while SYM=EXCESS do
       begin
        ERROR(3);
        GETSYM;
       end;
     end;

    procedure LISTABLE;
     var
       I: INTEGER;
     begin (*List symbol table for a block*)
       FOR I := 1 TO TX do
         with TABLE[I] do
           begin
             WRITE(OUTPUT,I: 1Ø, NAME: 9);
             case KIND of
               CONSTANT: WRITE(OUTPUT,'CONSTANT': 1Ø);
               VARIABLE: WRITE(OUTPUT,'VARIABLE': 1Ø);
               PROG:     WRITE(OUTPUT,'PROGRAM' : 1Ø);
               FUNC:     WRITE(OUTPUT,'FUNCTION': 1Ø);
               PROC:     WRITE(OUTPUT,'PROCEDURE':1Ø);
               MONI:     WRITE(OUTPUT,'MONITOR'  :1Ø);
               CONDVAR:  WRITE(OUTPUT,'CONDITION':1Ø);
               SYNC:     WRITE(OUTPUT,'SYNC PROC':1Ø);
               EPOINT:   WRITE(OUTPUT,'EPOINT'   :1Ø);
             end (*case*);
            if DEFINED then WRITE(OUTPUT,' DEF')
              else WRITE(OUTPUT,' UND');
            if ACCESS then WRITE(OUTPUT,' ACCESS')
             else WRITE(OUTPUT,' NOTACC');
            if CANCHANGE then WRITE(OUTPUT,' CAN ')
             else WRITE(OUTPUT,' CANT');
            if INSIDE then WRITE(OUTPUT,' IN ')
             else WRITE(OUTPUT,' OUT');
```

```
        WRITELN(OUTPUT,' U=',UNIQUE:2, LEVEL:4, ' ',
                        ADR, ' ', SIZE, ' ', MIN)
        end
    end (*LISTABLE*);

  procedure ISTARRED;
   begin   (*check for starred identifiers*)
    STAR:=FALSE;
    if SYM=TIMES then
     begin
      if BLOCKKIND=MONI then STAR:=TRUE else ERROR(47);
      GETSYM;
     end;
   end;   (*ISTARRED*)

  procedure ENTER(OBJ: OBJECT);
    var
      I : INTEGER;
    begin (*Enter object into table*)
      for I := TXØ + 1 to TX do if TABLE[I].NAME=ID then ERROR(4);
      TX := TX + 1;
      if TX <= TXMAX
      then with TABLE[TX] do
       begin
        NAME := ID; KIND := OBJ; LEVEL := LEV; SIZE := 1; MIN := Ø;
        CANCHANGE := TRUE; DEFINED := FALSE; VARPARAM := FALSE;
        INSIDE:=FALSE; ACCESS:=STAR;
        if KIND=MONI then
         begin
          MONICOUNT:=MONICOUNT+1; UNIQUE:=MONICOUNT;
          if MONICOUNT > MONMAX then ERROR(5Ø); (*too many monitors*)
         end
        else
         if KIND in [PROC,FUNC] then
          UNIQUE := TABLE[TXØ].UNIQUE
         else
          UNIQUE:=Ø;
       end
      else HALT('YY') (*symbol table overflow*)
    end (*ENTER*);

  procedure SEARCHFORWARD(var T:INTEGER);
   var TØ:INTEGER;
   begin
    if TABLE[T].KIND=SYNC then
      TØ:=TABLE[T].MIN  (*as size field used for parameters*)
    else
      TØ:=TABLE[T].SIZE;
    if TØ=Ø then TØ:=TX; (*still in the monitor*)
    T:=T+1;
    while (T<=TØ) and (TABLE[T].NAME<>ID) do T:=T+1;
    IF T > TØ then T:=Ø;
   end;   (*SEARCHFORWARD*)
```

```
procedure MONITORIDENTIFIERS(var T:INTEGER);
 var M:INTEGER;
  begin
   M:=T;
   if SYM <> PERIOD then ERROR(37)
   else
    begin
     GETSYM;
     SKIP(PERIOD);
     if SYM = IDENT then
       begin
        SEARCHFORWARD(T);
        GETSYM;
        if not(TABLE[T].ACCESS) then
         begin
          if (TABLE[TXØ].UNIQUE <> TABLE[M].UNIQUE) then ERROR(48);
         end;
       end
     else ERROR(6);
     WANTEXCLUSIVITY:=((TABLE[T].KIND IN [PROC,FUNC]) and
                       (TABLE[T].UNIQUE <> TABLE[TXØ].UNIQUE))
    end;   (*else*)
  end;   (*MONITORIDENTIFIERS*)

 procedure SYNCENTRYPT(var T:INTEGER);
  begin
   if SYM=PERIOD then
    begin
     GETSYM;
     SKIP(PERIOD);
     if SYM=IDENT then
      begin
       SEARCHFORWARD(T);
       GETSYM;
       if TABLE[T].KIND <> EPOINT then ERROR(63);
      end
     else ERROR(6);
    end
   else
    begin
      if (SYM<>LPAREN) and (SYM<>SEMICOLON) then ERROR(37);
    end;
 end;   (*syncentrypt*)

 function SEARCH(ID: ALFA): INTEGER;
  var I: INTEGER;
      FOUND:BOOLEAN;
  begin (*Find identifier in table*)
    TABLE[Ø].NAME := ID; I := TX; GETSYM;
    repeat
     FOUND:=TRUE;
     while TABLE[I].NAME <> ID do I := I - 1;
```

```
      if (TABLE[I].INSIDE) then
       begin
        I:=I-1;
        FOUND:=FALSE;
       end;
      until (I=Ø) or (FOUND);
     SEARCH := I;
   end (*SEARCH*);

  function POSITION(ID: ALFA): INTEGER;
   var I: INTEGER;
    begin (*find identifier in table, insert if missing*)
     I := SEARCH(ID);
     if TABLE[I].KIND = SYNC then
       SYNCENTRYPT(I)
     else
      if TABLE[I].KIND=MONI then
        MONITORIDENTIFIERS(I)
      else
        if TABLE[I].KIND <> CONDVAR then
         begin
          SKIP(PERIOD); (*confused*)
          if SYM=IDENT then GETSYM;
         end;
      if I = Ø then begin ERROR(5); ENTER(VARIABLE); I := TX end;
      POSITION := I
   end (*POSITION*);

 procedure GETCONSTANT(var C: INTEGER; FOLLOWERS: SYMSET);
  var I, SIGN: INTEGER;
   begin (*parse constants, numeric or named, signed or unsigned*)
    TEST(CONSTBEGSYS, FOLLOWERS, 3);
    if SYM in CONSTBEGSYS then
     begin
       SIGN := 1; C := Ø;
       if SYM in [PLUS,MINUS] then
         begin if SYM = MINUS then SIGN := -1; GETSYM end;
       if SYM = IDENT then
         begin
          I := POSITION(ID);
          if I <> Ø then with TABLE[I] do
           if KIND <> CONSTANT then ERROR(29) else C := SIGN * ADR
         end
       else
       if SYM = NUMBER then
         begin C := SIGN * NUM; GETSYM end
       else ERROR(8);
     end;
    TEST(FOLLOWERS, [], 3)
  end (*GETCONSTANT*);
```

```
  procedure CONSTDECLARATION(FOLLOWERS: SYMSET);
   begin
    GETSYM;
    ISTARRED;
    TEST([IDENT], FOLLOWERS, 6);
    repeat
     while SYM = IDENT do
       begin
         ENTER(CONSTANT);GETSYM;
         if BLOCKKIND=PROG then
          begin
           STARTOFMAINVAR:=STARTOFMAINVAR+1;
           ENDOFMAINVAR:=ENDOFMAINVAR+1;
          end;
         if SYM in [EQLSYM, BECOMES]
         then
           begin
             if SYM = BECOMES then ERROR(7); GETSYM;
             GETCONSTANT(TABLE[TX].ADR, FOLLOWERS+[COMMA,SEMICOLON]);
             TABLE[TX].DEFINED := TRUE (*value is obviously known*)
           end
         else ERROR(9);
         ACCEPT(SEMICOLON, 2);
        ISTARRED;
       end (*while*);
     TEST(FOLLOWERS, [IDENT] + STATBEGSYS + BLOCKBEGSYS, 3)
    until SYM <> IDENT (*silly error?*)
   end (*CONSTDECLARATION*);

 procedure VARDECLARATION(FOLLOWERS: SYMSET);

   procedure ENTERVARIABLE;
    begin
     if SYM = IDENT then
      begin
       ENTER(VARIABLE); GETSYM; TABLE[TX].ADR := ADDRESS;
       if SYM = LBRACK then (*Array declaration*)
        with TABLE[TX] do
         begin
          GETSYM; GETCONSTANT(MIN, FOLLOWERS+[COLON,RBRACK]);
          ACCEPT(COLON, 3Ø);
          GETCONSTANT(SIZE, FOLLOWERS+[RBRACK]);
          SIZE := SIZE - MIN + 1;
          if SIZE <= Ø then ERROR(31);
          ACCEPT(RBRACK, 1Ø)
         end;
       ADDRESS := ADDRESS + TABLE[TX].SIZE;
       if BLOCKKIND=PROG then ENDOFMAINVAR:=ENDOFMAINVAR+1;
     end
     else ERROR(6)
    end (*ENTERVARIABLE *);
```

```
  begin (*VARDECLARATION*)
   GETSYM;
   ISTARRED;
   TEST([IDENT], FOLLOWERS, 6);
   if BLOCKKIND=MONI then MONIVARADR[MONICOUNT,1]:=ADDRESS;
   repeat
    if SYM = IDENT then
     begin
       ENTERVARIABLE;
       while SYM = COMMA do begin GETSYM; ISTARRED; ENTERVARIABLE end;
     end;
    ACCEPT(SEMICOLON, 2); ISTARRED;
    TEST(FOLLOWERS, [IDENT] + STATBEGSYS + BLOCKBEGSYS, 3)
   until SYM <> IDENT (*Silly error?*);
   if BLOCKKIND=MONI then MONIVARADR[MONICOUNT,2]:=ADDRESS-1;
  end (*VARDECLARATION*);

 procedure CONDDECLARATION(FOLLOWERS:SYMSET);

  procedure ENTERCONDITION;
   begin
    if SYM=IDENT then
     begin
      ENTER(CONDVAR);
      GETSYM;
      CONDCOUNT:=CONDCOUNT+1;
      if CONDCOUNT > CONDMAX then ERROR(53);
      TABLE[TX].UNIQUE:=CONDCOUNT;
      TABLE[TX].ADR:=TABLE[TXØ].UNIQUE;
      TABLE[TX].DEFINED:=TRUE;   (*so no warning appears*)
      if SYM=LBRACK then (*array declaration*)
       with TABLE[TX] do
        begin
         GETSYM; GETCONSTANT(MIN,FOLLOWERS+[COLON,RBRACK]);
         ACCEPT(COLON,3Ø);
         GETCONSTANT(SIZE,FOLLOWERS+[RBRACK]);
         SIZE:=SIZE - MIN + 1;
         if SIZE <= Ø then ERROR(31);
         CONDCOUNT:=CONDCOUNT + SIZE - 1;
         ACCEPT(RBRACK,1Ø);
        end;   (*with*)
     end
    else
     ERROR(6);
   end;   (*entercondition*)

  begin  (*conddeclaration*)
   GETSYM; TEST([IDENT],FOLLOWERS,6);
   repeat
    if SYM=IDENT then
     begin
      ENTERCONDITION;
      while SYM=COMMA do begin GETSYM; ENTERCONDITION; end;
     end;
```

```
      ACCEPT(SEMICOLON,2);
      TEST(FOLLOWERS,[IDENT]+STATBEGSYS+BLOCKBEGSYS,3);
     until SYM <> IDENT;
    end;   (*conddeclaration*)

   procedure PARDECLARATION(FOLLOWERS: SYMSET; PROCENTRY: INTEGER);
    var I: INTEGER;

    procedure ENTERPARAMETER;
     var REFERENCE: BOOLEAN;
      begin
       REFERENCE := FALSE (*Assume passing by value wanted*);
       if SYM = VARSYM then begin GETSYM; REFERENCE := TRUE end;
       if SYM = IDENT then
        begin
         ENTER(VARIABLE); PARAMS := PARAMS + 1;
         if PARAMS > PMAX then HALT('PP') (*too many for ref array*);
         if COMPLETING then
          begin (*make sure no conflict with earlier declaration*)
           if TABLE[PROCENTRY].REF[PARAMS] <> REFERENCE then ERROR(38)
          end
         else
          TABLE[PROCENTRY].REF[PARAMS] := REFERENCE (*type of passing*);
         (*value parameters initialised*);
         TABLE[TX].DEFINED := not REFERENCE
         TABLE[TX].VARPARAM := REFERENCE; GETSYM
        end
       else ERROR(6)
      end;

   begin (*PARDECLARATION*)
    GETSYM; TEST([IDENT, VARSYM], FOLLOWERS, 6);
    repeat
     if SYM in [VARSYM, IDENT] then
      begin
       ENTERPARAMETER;
       while SYM = COMMA do begin GETSYM; ENTERPARAMETER end
      end;
     ACCEPT(RPAREN, 13); TEST(FOLLOWERS, [IDENT, VARSYM], 3)
    until not (SYM in [VARSYM, IDENT]);
    for I := 1 to PARAMS do (*Parameters have negative offsets*)
     TABLE[TX - I + 1].ADR := -I;
   end (*PARDECLARATION*);

  procedure PROCDECLARATION;
   var PROCKIND: OBJECT;
       COMPLETING: BOOLEAN;
       I: INTEGER;

   begin
    if ID = 'FUNCTION' then PROCKIND := FUNC
    else if ID = 'MONITOR ' then PROCKIND := MONI
         else PROCKIND := PROC;
   if (BLOCKKIND<>PROG) and (PROCKIND= MONI) then
     ERROR(46);   (*Declared in the wrong place*)
```

```
   GETSYM; COMPLETING := FALSE;
   ISTARRED;
   if SYM = IDENT then
    begin
     I := SEARCH(ID);
     if (I <> Ø) and (TABLE[I].LEVEL = LEV) and (PROCKIND<> MONI)
     then (*should be forward*)
      begin
       if TABLE[I].DEFINED then ERROR(4) (*redeclared*);
       BACKPATCH(TABLE[I].ADR, NEXTADDRESS);
       COMPLETING := TRUE
      end
     else ENTER(PROCKIND)
    end
   else ERROR(6);
   if PROCKIND=MONI then
     begin
      TABLE[TX].DEFINED:=TRUE; INMONITOR:=TRUE;
      BLOCK(FOLLOWERS, LEV, TX, PROCKIND, COMPLETING, I);
     end
   else
     BLOCK(FOLLOWERS, LEV+1, TX, PROCKIND, COMPLETING, I);
   if PROCKIND=MONI then
     begin
      TX:=TX+NEWGLOBALS;    (* Don't lose Monitor variables*)
      NEWGLOBALS:=Ø; INMONITOR:=FALSE;
     end;
   TEST([SEMICOLON], FOLLOWERS, 2);
   if SYM =  SEMICOLON then GETSYM
  end (*PROCDECLARATION*);

procedure SYNCDECLARATION; (*declaring of SYNCHRON procedures*)
 var I:INTEGER;
  begin
   if SYNCHRON then ERROR(65);
   if (BLOCKKIND<>PROG) then ERROR(6Ø); (*only in mainblock*)
   GETSYM;
   if SYM=IDENT then
    begin
     I:=SEARCH(ID);
     ENTER(SYNC);
    end
   else ERROR(6);
  SYNCHRON:=TRUE; (*dealing with a SYNCHRON procedure*)
  COMPLETING:=FALSE;
  BLOCK(FOLLOWERS,LEV+1,TX,SYNC,COMPLETING,I);
  SYNCHRON:=FALSE;
  TX:=TX+NEWGLOBALS;  (*adjust symbol table*)
  NEWGLOBALS:=Ø;
  TEST([SEMICOLON],FOLLOWERS,2);
  if SYM = SEMICOLON then GETSYM;
 end;   (*syncdeclaration*)
```

```
procedure ENTRYDECLARATION(FOLLOWERS:SYMSET);
 var TOP:INTEGER;

   procedure ENTERENTRY;
    var SAFE:INTEGER;
     begin
      if SYM=IDENT then
       begin
        ENTER(EPOINT);
        TABLE[TX].SIZE:=Ø;   (*no parameters as yet*)
        GETSYM;
        ENTRYCOUNT:=ENTRYCOUNT+1;
        if ENTRYCOUNT > ENTRYMAX then ERROR(62);
        TABLE[TX].UNIQUE:=ENTRYCOUNT;
        TOP:=TX;
        if SYM=LPAREN then
         begin
          COMPLETING:=FALSE;
          SAFE:=PARAMS;
          PARAMS:=Ø;
          PARDECLARATION(BLOCKBEGSYS +[SEMICOLON,COMMA],TX);
          TABLE[TOP].SIZE:=PARAMS;
          TABLE[TOP].ADR:=Ø;
          PARAMS:=SAFE;
         end;
        TX:=TOP; (*don't want to enter the parametrs at this stage*)
       end
      else ERROR(6);
    end;   (*enterentry*)

  begin (*entrydeclaration*)
   GETSYM;
   TEST([IDENT],FOLLOWERS,6);
   repeat
    if SYM=IDENT then
     begin
      ENTERENTRY;
      while SYM=COMMA do
       begin
        GETSYM; ENTERENTRY;
       end;
      end;   (*if*)
    ACCEPT(SEMICOLON,2);
    TEST(FOLLOWERS,[IDENT]+STATBEGSYS+BLOCKBEGSYS,3);
   until SYM <> IDENT;
  end;   (*entrydeclaration*)
```

```
procedure COMPOUNDSTATEMENT(FOLLOWERS: SYMSET); FORWARD;

  procedure STATEMENT(FOLLOWERS: SYMSET);
    var  I, TESTLABEL, STARTLOOP, THENLABEL: INTEGER;
         ETYPE: TYPES;
         HALTING: BOOLEAN;
         AREWRITING:BOOLEAN;   (*for problems of monitor functions*)

   procedure EXPRESSION(FOLLOWERS: SYMSET; var EXPTYPE:TYPES);FORWARD;

    procedure ADDRESSFOR(I: INTEGER);
     var ETYPE: TYPES;
     begin (*load address for identifier at table entry I *)
      with TABLE[I] do
       begin
        if UNIQUE=99 then  (*entry point parameter*)
         ENTRYPARAMETER(LEVEL,ADR)
        else
         STACKADDRESSFOR(LEVEL,ADR);
        if SYM = LBRACK then (*subscript*)
         begin if SIZE = 1 then ERROR(11); GETSYM;
          EXPRESSION([RBRACK] + FOLLOWERS, ETYPE);
          if not (ETYPE in [NOTYPE, INTS]) then ERROR(33);
          STACKCONSTANT(MIN); BINARYINTEGEROP(MINUS);
          SUBSCRIPT(SIZE); ACCEPT(RBRACK,1Ø)
         end
        else if SIZE > 1 then ERROR(15);
        if VARPARAM then DEREFERENCE
       end
     end (*ADDRESSFOR*);

    procedure CONDUNIQUE(I:INTEGER);
     var ETYPE:TYPES;
      begin
       with TABLE[I] do
        begin   (*with*)
         STACKCONSTANT(UNIQUE);
         if SYM=LBRACK then (*subscript*)
          begin
           if SIZE=1 then ERROR(11); GETSYM;
           EXPRESSION([RBRACK]+FOLLOWERS,ETYPE);
           if not(ETYPE in [NOTYPE,INTS]) then ERROR(33);
           STACKCONSTANT(MIN);
           BINARYINTEGEROP(MINUS);
           SUBSCRIPT(SIZE);
           ACCEPT(RBRACK,1Ø);
          end
         else
          if SIZE > 1 then ERROR(15);
        end;   (*with*)
      end;   (*condunique*)
```

```
procedure PARAMETERS(FORMAL: INTEGER; FOLLOWERS: SYMSET;
                     PRENTRY: INTEGER);
  var PTYPE: TYPES;
      I, ACTUAL: INTEGER;
   begin
    ACTUAL := Ø;
    if SYM = LPAREN then
     begin
      repeat
       GETSYM;
       if ACTUAL >= FORMAL then ERROR(12)
       else
        begin
         ACTUAL := ACTUAL+1; if ACTUAL > PMAX then HALT('PP');
         if TABLE[PRENTRY].REF[ACTUAL] then (*var parameter*)
          if SYM <> IDENT then ERROR(6) else
           begin
            I := POSITION(ID);
            if I <> Ø then with TABLE[I] do
             if KIND <> VARIABLE then ERROR(22) else ADDRESSFOR(I)
           end
         else
          begin (*value parameter*)
           EXPRESSION(FOLLOWERS+[COMMA,RPAREN], PTYPE);
           if not (PTYPE in [NOTYPE, INTS]) then ERROR(33)
          end;
        end;
       TEST ([COMMA, RPAREN], FOLLOWERS, 13)
      until SYM <> COMMA;
      ACCEPT(RPAREN, 13)
     end;
    if ACTUAL < FORMAL then ERROR(12)
   end;

procedure EXPRESSION;
  var RELOP: SYMBOL;
      FTYPE: TYPES;

  procedure CHECKTYPE(var A: TYPES; B,C: TYPES);
   begin (*check A and B are of type C*)
    if (A <> C) or (B <> C) then
     begin
      if (A <> NOTYPE) and (B <> NOTYPE) then ERROR(33); A:= NOTYPE
    end
   end (*CHECKTYPE*);

  procedure SIMPLEEXPRESSION(FOLLOWERS: SYMSET; var STYPE: TYPES);
   var ADDOP: SYMBOL;
       FTYPE: TYPES;

   procedure TERM(FOLLOWERS: SYMSET; var TERMTYPE: TYPES);
     var MULOP: SYMBOL;
         FTYPE: TYPES;
```

```
procedure FACTOR(FOLLOWERS: SYMSET; var FACTYPE: TYPES);
 var I: INTEGER;
  begin
   TEST(FACBEGSYS, FOLLOWERS, 14);
   FACTYPE := INTS;
   while SYM in FACBEGSYS do
     begin
       case SYM of
         IDENT:
           begin
             I := POSITION(ID);
             if I <> Ø
             then with TABLE[I] do
               case KIND of
                CONSTANT: STACKCONSTANT(ADR);
                VARIABLE:
                 begin
                  ADDRESSFOR(I); DEREFERENCE;
                  if ((not(DEFINED)) and (LISTING))
                     and (not(NOWARN)) then
                   WRITELN(OUTPUT,'*WARNING* ',
                        '^':(abs(CS-5+OFFSET)), 'UNDEFINED?')
                 end;
                CONDVAR: begin
                          CONDUNIQUE(I);
                          if SYM <> PERIOD then ERROR(37)
                          else
                           begin
                            GETSYM;
                            SKIP(PERIOD);
                            if not(SYM in
                                   [QUEUESYM,QLENSYM]) then
                              ERROR(52)
                            else
                             begin
                              case SYM of
                               QLENSYM : CONDVARCODE(QLN,Ø);
                               QUEUESYM: begin
                                          EXPTYPE:= BOOLS;
                                          CONDVARCODE(QUE,Ø);
                                         end;
                              end;  (*case*)
                             end;   (*else*)
                            GETSYM;
                           end;    (*else*)
                         end;   (*condvar*)
```

```
                    FUNC: begin
                            OPENSTACKFRAME(1) (*for value*);
                            PARAMETERS(SIZE, FOLLOWERS, I);
                            if WANTEXCLUSIVITY then
                             begin
                              if AREWRITING then TOGGLESWITCHING;
                              WANTEXCLUSIVITY:=FALSE;
                              OBTAINEXCLUSIVITY(UNIQUE);
                              if AREWRITING then TOGGLESWITCHING;
                             end;
                            if (UNIQUE = TABLE[TXØ].UNIQUE)
                               and (ACCESS) then
                              (*calling from the same monitor*)
                              CALL(LEVEL,(ADR + CODEMAX))
                            else
                              CALL(LEVEL,ADR);
                           end;
                    PROG: ERROR(14);
                    PROC: ERROR(16)
                   end (*case*)
               end;
             NUMBER    : begin STACKCONSTANT(NUM); GETSYM end;
             RANDSYM   : begin CODEFORRANDOM; GETSYM end;
             ACTIVESYM: begin CODEFORACTIVE; GETSYM; end;
             READYSYM : begin RDYCODE; GETSYM; end;
             LPAREN: begin
                      GETSYM;
                      EXPRESSION([RPAREN] + FOLLOWERS, FACTYPE);
                      ACCEPT(RPAREN, 17)
                     end;
             end (*case*);
           TEST(FOLLOWERS, FACBEGSYS, 3)
         end (*while*)
   end (*FACTOR*);

  begin (*TERM*)
    FACTOR(FOLLOWERS + [TIMES,SLASH,MODSYM,ANDSYM], TERMTYPE);
    while SYM in [TIMES, SLASH, MODSYM, ANDSYM] do
     begin
       MULOP := SYM; GETSYM;
       FACTOR(FOLLOWERS + [TIMES,SLASH,MODSYM,ANDSYM], FTYPE);
       if MULOP = ANDSYM then
        begin
         BINARYBOOLEANOP(MULOP);
         CHECKTYPE(TERMTYPE, FTYPE, BOOLS)
        end (*ANDSYM*)
       else
        begin
         BINARYINTEGEROP(MULOP);
         CHECKTYPE(TERMTYPE, FTYPE, INTS)
        end (*other mulops*)
     end (*while*)
  end (*TERM*);
```

```
      begin (*SIMPLEEXPRESSION*)
        if SYM in [PLUS, MINUS] then
          begin
            ADDOP := SYM; GETSYM;
            TERM(FOLLOWERS + [PLUS, MINUS, ORSYM], STYPE);
            if not (STYPE in [NOTYPE,INTS]) then ERROR(33);
            if ADDOP = MINUS then NEGATEINTEGER
          end
        else TERM(FOLLOWERS + [PLUS, MINUS, ORSYM], STYPE);
        while SYM in [PLUS, MINUS, ORSYM] do
          begin
            ADDOP := SYM; GETSYM;
            TERM(FOLLOWERS + [PLUS, MINUS, ORSYM], FTYPE);
             if ADDOP = ORSYM then
              begin
               BINARYBOOLEANOP(ADDOP);
               CHECKTYPE(STYPE, FTYPE, BOOLS)
              end (*ORSYM*)
             else
              begin
               BINARYINTEGEROP(ADDOP);
               CHECKTYPE(STYPE, FTYPE, INTS)
              end (*other addops*)
          end (*while*)
      end (*SIMPLEEXPRESSION*);


    begin (*EXPRESSION*)
      SIMPLEEXPRESSION(RELOPSYS + FOLLOWERS, EXPTYPE);
      if SYM in RELOPSYS then
      begin
        RELOP := SYM; GETSYM; SIMPLEEXPRESSION(FOLLOWERS, FTYPE);
        CHECKTYPE(EXPTYPE, FTYPE, INTS); COMPARISON(RELOP);
        EXPTYPE := BOOLS
      end
    end (*EXPRESSION*);

      procedure ACCEPTSTATEMENT;
       var I,TOP,UNIQ,LC:INTEGER;
           SAFE:INTEGER;
        begin
         if ISACCEPT then ERROR(67);
         ISACCEPT:=TRUE; GETSYM;
         I:=POSITION(ID);
         with TABLE[I] do
          begin (*with*)
           if KIND <> EPOINT then ERROR(66)
           else
            begin
             EMITACCEPT(UNIQUE); UNIQ:=UNIQUE;
            end;
          end;   (*with*)
```

```
        TOP:=TX; (*Current top of symbol table*)
        SAFE:=PARAMS;
        if SYM=LPAREN then
         begin
          COMPLETING:=TRUE;  (*like a forward procedure*)
          PARAMS:=0;
          PARDECLARATION(BLOCKBEGSYS+[THENSYM],I);
          COMPLETING:=FALSE;
          for LC:=(TOP+1) to TX do
           begin
            TABLE[LC].UNIQUE:=99; (*sentinal-entry point parameter*)
           end;
         end
        else
         if TABLE[I].SIZE > 1 then ERROR(12);
       if SYM = THENSYM then GETSYM
       else begin ERROR(23); if SYM=DOSYM then GETSYM; end;
       STATEMENT(FOLLOWERS+[ELSESYM(*in select*)]);
       ISACCEPT:=FALSE;
       STACKCONSTANT(UNIQ);
       EMITENDACCEPT(LEV,PARAMS+1);
       TX:=TOP; (*collapse level of symbol table*)
       PARAMS:=SAFE;
     end;   (*acceptstatement*)


    procedure CONDITION(FOLLOWERS: SYMSET);
     var ETYPE: TYPES;
      begin
        if SYM = ODDSYM
        then
          begin
            GETSYM; ACCEPT(LPAREN, 18);
            EXPRESSION(FOLLOWERS + [RPAREN], ETYPE); CODEFORODD;
            if not (ETYPE in [NOTYPE, INTS]) then ERROR(33);
            ACCEPT(RPAREN, 17)
          end
        else
          begin
            EXPRESSION(FOLLOWERS, ETYPE);
            if not (ETYPE in [NOTYPE, BOOLS]) then ERROR(33)
          end
      end (*CONDITION*);
```

```
procedure SELECTSTATEMENT;
 const MAXGUARD=20; (*max. no. of guards per select*)
 var ENDSELECT: ARRAY[1..MAXGUARD] of INTEGER;
     START,STOP,SUB,LC,NEXTG:INTEGER;
     ISNOGUARD:BOOLEAN;
  begin
   ISNOGUARD:=FALSE;
   if ISELSECASE then ERROR(70);
   if not(SYNCHRON) then ERROR(68);
   SUB:=0; (*no. of guard conditions*)
   START:=ADDRESS+1;
   GETSYM;
   while (SYM <> ENDSYM) and (SYM<>ELSESYM) do
    begin   (*while*)
     ADDRESS:=ADDRESS+1;
     if SYM = NOGARDSYM then
      begin
       ISNOGUARD:=TRUE;
       STACKADDRESS(LEV,ADDRESS);
       STACKCONSTANT(1);
       ASSIGN;
       GETSYM;
      end
     else
      begin
       STACKADDRESS(LEV,ADDRESS);
       CONDITION(FOLLOWERS + [COLON,ACCEPTSYM]);
       ASSIGN;
      end;
     if SYM <> COLON then ERROR(19);
     GETSYM;
     ADDRESS:=ADDRESS+1;
     STACKADDRESS(LEV,ADDRESS);
     (*address of accept statement*)
     STACKCONSTANT(NEXTADDRESS+3);
     (*it  must  be  nextaddress+3  to
       take into account the STO & BRN*)
     ASSIGN;
     TEST([ACCEPTSYM],FOLLOWERS + [NOGARDSYM,IDENT],69);
     STORELABEL(NEXTG);
     JUMP(-1);                          (*backpatch later*)
     SUB := SUB + 1;          (*another guard condition*)
     if SUB > MAXGUARD then ERROR(70);      (*too many*)
     ACCEPTSTATEMENT;
     if SYM=SEMICOLON then GETSYM
     else if (not (SYM in [ENDSYM,ELSESYM])) then ERROR(2);
     STORELABEL(ENDSELECT[SUB]);
     (*accepts continue after select*)
     JUMP(-1);   (*jump to after the select statement*)
     BACKPATCH(NEXTG,NEXTADDRESS); (*guards evaluated first*)
    end;  (*while*)
```

```
      if SYM=ELSESYM then
       begin
         ADDRESS:=ADDRESS+1;
         STACKADDRESS(LEV,ADDRESS);
         STACKCONSTANT(2);
         ASSIGN;
         if (ISNOGUARD) and not(NOWARN) then
          WRITELN(OUTPUT,'*WARNING*','^':(abs(CS-5+OFFSET)),
                         'ELSE REDUNDANT');
         ADDRESS:=ADDRESS+1;
         STACKADDRESS(LEV,ADDRESS);
         STACKCONSTANT(NEXTADDRESS+3);  (*address of statements*)
         ASSIGN;
         STORELABEL(NEXTG);
         JUMP(-1);
         GETSYM;    (*get elsesym*)
         ISELSECASE:=TRUE;
         STATEMENT(FOLLOWERS);
         SUB:=SUB+1;
         if SUB>MAXGUARD then ERROR(70);
         STORELABEL(ENDSELECT[SUB]);
         JUMP(-1);  (*backpatched*)
         ISELSECASE:=FALSE;
         ACCEPT(SEMICOLON,2);
       end; (*else clause*)
      BACKPATCH(NEXTG,NEXTADDRESS); (*last guard must branch here*)
     GETSYM;                              (*get rid of the endsym*)
    STACKCONSTANT(START);              (*start of guard conditions*)
    EMITSELECT(LEV,ADDRESS);
    for LC:= 1 to SUB do
     begin
      BACKPATCH(ENDSELECT[LC],NEXTADDRESS);
      (*all accepts continue after the select statement*)
     end;
    BACKPATCH(SIZEREQUIRED,ADDRESS+1);
    (*grab a bigger portion of stack*)
  end;   (*SELECTSTATEMENT*)

  procedure IFSTATEMENT;
    begin
     GETSYM; CONDITION([THENSYM, DOSYM] + FOLLOWERS);
     if SYM = THENSYM then GETSYM
     else begin ERROR(23); if SYM = DOSYM then GETSYM end;
     STORELABEL(TESTLABEL);
     JUMPONFALSE(0) (*Incomplete*);
     STATEMENT(FOLLOWERS + [ELSESYM]);
     if SYM <> ELSESYM then BACKPATCH(TESTLABEL,NEXTADDRESS)
     else
       begin
         GETSYM; STORELABEL(THENLABEL); JUMP(0) (*incomplete*);
         BACKPATCH(TESTLABEL, NEXTADDRESS);
         STATEMENT(FOLLOWERS); BACKPATCH(THENLABEL, NEXTADDRESS)
       end (*else parse*);
   end (*IFSTATEMENT*);
```

```
procedure WHILESTATEMENT;
 begin
   STORELABEL(TESTLABEL); GETSYM;
   CONDITION([DOSYM] + FOLLOWERS);
   STORELABEL(STARTLOOP); JUMPONFALSE(0) (*Incomplete*);
   ACCEPT(DOSYM, 25); STATEMENT(FOLLOWERS);
   JUMP(TESTLABEL); BACKPATCH(STARTLOOP,NEXTADDRESS);
 end (*WHILESTATEMENT*);

procedure REPEATSTATEMENT;
 begin
   STORELABEL(STARTLOOP);
   GETSYM; STATEMENT([SEMICOLON,FOREVERSYM,UNTILSYM] + FOLLOWERS);
   while SYM in [SEMICOLON] + STATBEGSYS do
     begin
       ACCEPT(SEMICOLON, 2);
       STATEMENT([SEMICOLON, FOREVERSYM, UNTILSYM] + FOLLOWERS)
     end;
   if SYM = FOREVERSYM then begin JUMP(STARTLOOP); GETSYM end
    else
     begin
       ACCEPT(UNTILSYM, 26); CONDITION(FOLLOWERS);
       JUMPONFALSE(STARTLOOP)
     end;
 end (*REPEATSTATEMENT*);

procedure OUTPUTSTATEMENT;
 var ENDING:BOOLEAN;
  begin
   AREWRITING:=FALSE;   (*for monitor function call*)
   if CLEANIO then begin TOGGLESWITCHING; AREWRITING:=TRUE; end;
   ENDING:= ID='WRITELN ';
   HALTING := SYM = HALTSYM; GETSYM;
   if SYM = LPAREN then
     begin
       repeat
         GETSYM;
         if SYM <> STRINGSYM then
          begin
           EXPRESSION(FOLLOWERS + [COMMA, RPAREN,COLON], ETYPE);
           (*Boolean expressions can be output as 0 or 1 *)
           if SYM = DOLLAR then   (*deal with formatter*)
             begin OUTPUTOPERATION(CHARS); GETSYM; end
           else OUTPUTOPERATION(NUMBERS)
          end
         else
          begin OUTPUTOPERATION(STRINGS); GETSYM end (*String*)
       until SYM <> COMMA;
       ACCEPT(RPAREN, 13)
     end;
   if MONCHK then PROCESSTRACE;
   if ENDING then OUTPUTOPERATION(NEWLINE);
   if CLEANIO then begin TOGGLESWITCHING; AREWRITING:=FALSE; end;
   if HALTING then LEAVEBLOCK(PROG, LEV, 0)
 end (*OUTPUTSTATEMENT*);
```

```
     procedure INPUTSTATEMENT;
      var ENDING:BOOLEAN;
       begin
        if CLEANIO then TOGGLESWITCHING;
        ENDING:= ID='READLN  '; GETSYM;
        if SYM <> LPAREN then ERROR(18)
        else
          begin
            repeat
              GETSYM;
              if SYM <> IDENT then ERROR(6)
              else
                begin
                  I := POSITION(ID);
                  if I <> 0 then with TABLE[I] do
                   begin
                    if KIND <> VARIABLE then ERROR(28)
                    else
                     begin
                      if not CANCHANGE then ERROR(39);
                      ADDRESSFOR(I);
                      DEFINED := TRUE;     (*known at run time*)
                      if SYM=DOLLAR then (*deal with formatter*)
                       begin INPUTOPERATION(CHARS); GETSYM; end
                      else INPUTOPERATION(NUMBERS)
                     end
                   end
                end
            until SYM <> COMMA;
            ACCEPT(RPAREN, 13);
          end;
        if ENDING then INPUTOPERATION(NEWCARD);
        if CLEANIO then TOGGLESWITCHING;
      end (*INPUTSTATEMENT*);

     procedure SEMASTATEMENT;
      var WAITSEM: BOOLEAN;
       begin
        if INMONITOR then ERROR(54)
        else
        begin
         WAITSEM := SYM = WAITSYM; GETSYM;
         if SYM <> LPAREN then ERROR(18)
         else
           begin
             GETSYM;
             if SYM <> IDENT then ERROR(6)
             else
               begin
```

```
                I := POSITION(ID);
                if I <> Ø then with TABLE[I] do
                 begin
                  if KIND <> VARIABLE then ERROR(28)
                  else
                   begin
                    if not CANCHANGE then ERROR(39);
                    ADDRESSFOR(I);
                    if WAITSEM then CODEFORWAIT else CODEFORSIGNAL;
                   end
                 end
              end;
           ACCEPT(RPAREN, 13);
         end;
      end;   (*else*)
     end (*SEMASTATEMENT*);

    procedure CONCSTATEMENT;
     var NPR: INTEGER (*Count number of processes*);
      begin
       NPR := Ø; STORELABEL(STARTLOOP);
       if (LEV <> 1) or (BLOCKKIND=MONI) then ERROR(44);
       GETSYM; STARTPROCESSES;
       STATEMENT([SEMICOLON, COENDSYM] + FOLLOWERS);
       if PROCCALL then NPR:=NPR+1 else ERROR(42);
       PROCCALL:=FALSE;  (*in case next statement not procedure call*)
       while SYM in [SEMICOLON] + STATBEGSYS do
        begin
         ACCEPT(SEMICOLON, 2);
         STATEMENT([SEMICOLON, COENDSYM] + FOLLOWERS);
         if PROCCALL then NPR:=NPR+1
         else if SYM <> COENDSYM then ERROR(42);
         PROCCALL:=FALSE;(*in case next statement not procedure call*)
        end;
       BACKPATCH(STARTLOOP, NPR);
       ACCEPT(COENDSYM, 43); STOPPROCESSES;
       if NPR > PRMAX then ERROR(45) (*too many*);
      end (*CONCSTATEMENT*);
```

```
    procedure FORSTATEMENT;
     var I: INTEGER;
         UP: BOOLEAN;
         NOTALTER:BOOLEAN;
      begin
       GETSYM; I := Ø (*Index into table*);
       if SYM = IDENT then
        begin
         I := POSITION(ID);
         if I <> Ø then with TABLE[I] do
          if KIND = VARIABLE then
           begin
            if not CANCHANGE or VARPARAM then ERROR(39);
            if (LEV <> LEVEL) then ERROR(4Ø) (*Must be local*);
            ADDRESSFOR(I)
           end
          else ERROR(22)
        end
       else ERROR(6) (*identifier?*);
       TEST([BECOMES], [TOSYM, DOWNTOSYM, DOSYM] + FOLLOWERS, 21);
       TABLE[I].DEFINED := TRUE;
       if SYM = BECOMES then
        begin
         GETSYM; EXPRESSION([TOSYM,DOWNTOSYM,DOSYM] + FOLLOWERS, ETYPE);
         if not (ETYPE in [NOTYPE, INTS]) then ERROR(33)
        end;
       TEST([TOSYM, DOWNTOSYM] , [DOSYM] + FOLLOWERS, 41);
       if SYM in [TOSYM, DOWNTOSYM] then
        begin
         UP := SYM = TOSYM; GETSYM;
         EXPRESSION([DOSYM] + FOLLOWERS, ETYPE);
         if not (ETYPE in [NOTYPE, INTS]) then ERROR(33)
        end;
       ACCEPT(DOSYM, 25); STORELABEL(STARTLOOP);
       STARTFORLOOP(UP); STORELABEL(TESTLABEL);
       NOTALTER:=TABLE[I].CANCHANGE;
       TABLE[I].CANCHANGE := FALSE; STATEMENT(FOLLOWERS);
       ENDFORLOOP(UP, TESTLABEL); BACKPATCH(STARTLOOP, NEXTADDRESS);
       TABLE[I].CANCHANGE := NOTALTER; TABLE[I].DEFINED := FALSE
      end (*FORSTATEMENT*);

  procedure PRIORITYWAIT;
   begin
    GETSYM;   (*should be a lparen*)
    if SYM=LPAREN then
     begin
      GETSYM;
      EXPRESSION(FOLLOWERS+[RPAREN],ETYPE);
      if ETYPE<>INTS then ERROR(33);
      (*priority should be at top of stack*)
      if SYM <> RPAREN then ERROR(17);
     end
    else
     ERROR(18);
   end;   (*prioritywait*)
```

```
procedure FORSTATEMENT;
 var I: INTEGER;
     UP: BOOLEAN;
     NOTALTER:BOOLEAN;
  begin
   GETSYM; I := Ø (*Index into table*);
   if SYM = IDENT then
    begin
     I := POSITION(ID);
     if I <> Ø then with TABLE[I] do
      if KIND = VARIABLE then
       begin
        if not CANCHANGE or VARPARAM then ERROR(39);
        if (LEV <> LEVEL) then ERROR(4Ø) (*Must be local*);
        ADDRESSFOR(I)
       end
      else ERROR(22)
    end
   else ERROR(6) (*identifier?*);
   TEST([BECOMES], [TOSYM, DOWNTOSYM, DOSYM] + FOLLOWERS, 21);
   TABLE[I].DEFINED := TRUE;
   if SYM = BECOMES then
    begin
     GETSYM; EXPRESSION([TOSYM,DOWNTOSYM,DOSYM] + FOLLOWERS, ETYPE);
     if not (ETYPE in [NOTYPE, INTS]) then ERROR(33)
    end;
   TEST([TOSYM, DOWNTOSYM] , [DOSYM] + FOLLOWERS, 41);
   if SYM in [TOSYM, DOWNTOSYM] then
    begin
     UP := SYM = TOSYM; GETSYM;
     EXPRESSION([DOSYM] + FOLLOWERS, ETYPE);
     if not (ETYPE in [NOTYPE, INTS]) then ERROR(33)
    end;
   ACCEPT(DOSYM, 25); STORELABEL(STARTLOOP);
   STARTFORLOOP(UP); STORELABEL(TESTLABEL);
   NOTALTER:=TABLE[I].CANCHANGE;
   TABLE[I].CANCHANGE := FALSE; STATEMENT(FOLLOWERS);
   ENDFORLOOP(UP, TESTLABEL); BACKPATCH(STARTLOOP, NEXTADDRESS);
   TABLE[I].CANCHANGE := NOTALTER; TABLE[I].DEFINED := FALSE
  end (*FORSTATEMENT*);

procedure PRIORITYWAIT;
 begin
  GETSYM;   (*should be a lparen*)
  if SYM=LPAREN then
   begin
    GETSYM;
    EXPRESSION(FOLLOWERS+[RPAREN],ETYPE);
    if ETYPE<>INTS then ERROR(33);
    (*priority should be at top of stack*)
    if SYM <> RPAREN then ERROR(17);
   end
  else
   ERROR(18);
 end;   (*prioritywait*)
```

```
procedure SAVERESTOREVARIABLES;

  procedure SAVEPARAMETERS;
   var I,LC:INTEGER;
       ETYPE: TYPES;
       WHOLEARAY:BOOLEAN; (*whether saving whole array with save*)
    begin
     I:=POSITION(ID);
     if I <> 0 then
      begin
       with TABLE[I] do
        begin
         if (KIND=VARIABLE) and (not(INSIDE)) and (CANCHANGE)
            and (LEVEL=1) then
             begin (*load address for identifier at table entry I*)
              WHOLEARAY:=TRUE;
              STACKADDRESSFOR(LEVEL,ADR);
              if SYM = LBRACK then (*subscript*)
               begin
                WHOLEARAY:=FALSE;
                if SIZE = 1 then ERROR(11);
                GETSYM;
                EXPRESSION([RBRACK] + FOLLOWERS, ETYPE);
                if not (ETYPE in [NOTYPE, INTS]) then ERROR(33);
                STACKCONSTANT(MIN); BINARYINTEGEROP(MINUS);
                SUBSCRIPT(SIZE); ACCEPT(RBRACK,10)
               end;   (*if SYM=LBRACK*)
              if SIZE > 1 then (*array*)
               begin
                if WHOLEARAY then (*save the whole array*)
                 begin
                  for LC:= 1 to (SIZE-1) do
                    STACKADDRESSFOR(LEVEL,ADR+LC);
                 end;   (*if WHOLEARRAY*)
               end;   (*if SIZE > 1*)
             end   (*if legitimate*)
         else
          ERROR(55); (*only current monitor variables saved*)
        end;   (*with*)
      end;   (*if I<>0*)
    end;   (*SAVEPARAMETERS*)

 begin   (*SAVERESTOREVARIABLES*)
  if not(TABLE[TX0].KIND in [PROC,FUNC]) then
    ERROR(57)   (*only in proc/func*)
  else
   begin
    if TABLE[TX0].UNIQUE < 1 then ERROR(57)   (*only in monitors*)
    else
     begin
      if SYM=SAVESYM then
       begin
        MISSRESTORE:=TRUE;
        ISSAVE:=TRUE;
        GETSYM;
```

```
        if SYM <> LPAREN then ERROR(18)
        else
         begin
          SAVEMARKER; GETSYM;
          if SYM <> IDENT then ERROR(6)
          else
           begin   (*else*)
            SAVEPARAMETERS;
            while SYM = COMMA do
             begin
              GETSYM;
              if SYM <> IDENT then ERROR(6)
              else SAVEPARAMETERS;
             end;   (*while*)
            ACCEPT(RPAREN,17);
           end;   (*else*)
         end;   (*if SYM=LPAREN*)
        SAVEVARIABLES(TABLE[TXØ].UNIQUE);
       end   (*if SYM=SAVESYM*)
      else
       begin
        MISSRESTORE:=FALSE;
        RESTOREVARIABLES(TABLE[TXØ].UNIQUE);
        GETSYM;
       end;
     end;
   end;
 end;   (*SAVERESTOREVARIABLES*)
```

```
begin (*STATEMENT*)
  if SYM in STATBEGSYS
  then
    case SYM of
      IDENT:
        begin
          I := POSITION(ID);
          if I <> 0
          then with TABLE[I] do
          case KIND of
           FUNC, VARIABLE:
            begin
             if KIND = VARIABLE then ADDRESSFOR(I)
             else
              if LEV > LEVEL
               then STACKADDRESS(LEVEL+1, -SIZE-1) else ERROR(20);
             if not CANCHANGE then ERROR(39);
             if SYM = BECOMES then GETSYM
             else begin ERROR(21); if SYM = EQLSYM then GETSYM end;
             EXPRESSION(FOLLOWERS, ETYPE);
             if not (ETYPE in [NOTYPE, INTS]) then ERROR(33);
             DEFINED := TRUE (*Will get value at run time*);
             ASSIGN
            end;
           SYNC: begin
                   PROCCALL:=TRUE;
                   if (BLOCKKIND<>PROG) then ERROR(60);
                   PARAMETERS(SIZE,FOLLOWERS,I);
                   CALL(LEVEL,ADR);
                  end;
           PROC:
            begin
             PROCCALL:=NOT(INSIDE);
             PARAMETERS(SIZE, FOLLOWERS, I);
             if WANTEXCLUSIVITY then
              begin
               WANTEXCLUSIVITY:=FALSE;
               OBTAINEXCLUSIVITY(UNIQUE);
              end;
             if (UNIQUE = TABLE[TX0].UNIQUE) and (ACCESS) then
              (*calling starred procedure from inside the same
                monitor so set up flag to ignore the next  LMN
                instruction*)
               CALL(LEVEL,(ADR + CODEMAX))
             else
              CALL(LEVEL, ADR);
            end;
           EPOINT: begin
                     if (TABLE[TX0].UNIQUE<>0)  (*ie. in monitor*)
                      or (SYNCHRON) or (BLOCKKIND=PROG)
                      or (ISACCEPT) then
                       ERROR(64);   (*illegal position*)
                     PARAMETERS(SIZE,FOLLOWERS,I);
                     SYNCCALL(LEVEL,UNIQUE);
                    end;
```

```
          CONDVAR: begin
                     CONDUNIQUE(I);
                     if SYM <> PERIOD then ERROR(37)
                      else
                       begin
                        GETSYM;
                        SKIP(PERIOD);
                        if not(SYM in
                           [QPWAITSYM,QSIGNALSYM,QWAITSYM]) then
                         ERROR(52)
                        else
                         begin
                          if not(ISSAVE) and not(NOWARN) then
                           WRITELN(OUTPUT,'*WARNING*',
                                 '^':(abs(CS-5+OFFSET)),
                                 'MONITOR VARIABLES NOT INVARIANT');
                          case SYM of
                           QWAITSYM:begin
                                     STACKCONSTANT(DEFAULT);
                                     CONDVARCODE(QWT,Ø);
                                    end;
                           QPWAITSYM:begin
                                      PRIORITYWAIT;
                                      CONDVARCODE(QPW,Ø);
                                     end;
                           QSIGNALSYM: CONDVARCODE(QSG,ADR);
                          end;   (*case*)
                          ISSAVE:=FALSE;
                         end;   (*else*)
                        GETSYM;
                       end;   (*else*)
                     end;   (*condvar*)
           CONSTANT, PROG: ERROR(22);
         end
       end (*IDENT*);
     IFSYM      : IFSTATEMENT;
     BEGINSYM   : COMPOUNDSTATEMENT(FOLLOWERS);
     WHILESYM   : WHILESTATEMENT;
     REPEATSYM  : REPEATSTATEMENT;
     FORSYM     : FORSTATEMENT;
     COBEGINSYM: CONCSTATEMENT;
     HALTSYM, WRITESYM  : OUTPUTSTATEMENT;
     READSYM    : INPUTSTATEMENT;
     WAITSYM,SIGNALSYM  : SEMASTATEMENT;
     ACCEPTSYM : ACCEPTSTATEMENT;
     SELECTSYM : SELECTSTATEMENT;
     SAVESYM,RESTORESYM : SAVERESTOREVARIABLES;
     STOPCSYM: begin GEN(RET,Ø,-1(*sentinal*)); GETSYM; end;
     STACKSYM: begin CODEFORDUMP(LEV); GETSYM end;
     SWITCHSYM: begin PROCSWITCH; GETSYM end;
   end (*case *);
  TEST(FOLLOWERS, [], 32)
 end (*STATEMENT*);
```

```
procedure COMPOUNDSTATEMENT;
    begin
      ACCEPT(BEGINSYM, 34);
      STATEMENT([SEMICOLON, ENDSYM] + FOLLOWERS);
      while SYM in [SEMICOLON] + STATBEGSYS do
        begin
          ACCEPT(SEMICOLON, 2);
          STATEMENT([SEMICOLON, ENDSYM] + FOLLOWERS)
        end;
      if MISSRESTORE then writeln(OUTPUT,'*WARNING*',
                              ' ':(abs(CS-5+OFFSET)),'MISSING RESTORE');
      ACCEPT(ENDSYM, 24);
    end (*COMPOUNDSTATEMENT*);

begin (*BLOCK*)
  PARAMS := Ø; TXØ := TX;
  if BLOCKKIND=MONI then ADDRESS:=GLOBALADDRESS
  else ADDRESS := 3 (*First variable has offset 3*);
  ENTERBLOCK(STARTBLOCK);

  if LEV > LEVMAX then HALT('YY') (*too deeply nested*);

  case BLOCKKIND of
    PROC,FUNC,SYNC: begin
                     if SYM = LPAREN then
                      if COMPLETING then
                       PARDECLARATION(BLOCKBEGSYS + [SEMICOLON],
                                      BLOCKENTRY)
                      else
                       PARDECLARATION(BLOCKBEGSYS + [SEMICOLON], TXØ);
                     ACCEPT(SEMICOLON, 2)
                    end;
    MONI:   begin
             for I:= STARTOFMAINVAR to ENDOFMAINVAR do
              (*To make variables in main block read only to Monitors*)
               TABLE[I].CANCHANGE:=FALSE;
             PREVIOUS:=PRESENT;
             PRESENT:=STARTBLOCK;
             ACCEPT(SEMICOLON,2);
            end;
    PROG:  begin PREVIOUS:=PRESENT; PRESENT:=STARTBLOCK; end;
  end;   (*case*)

  TEST(BLOCKBEGSYS, FOLLOWERS, 3);
  if not COMPLETING then
   begin TABLE[TXØ].ADR := STARTBLOCK; TABLE[TXØ].SIZE := PARAMS end
  else if PARAMS <> TABLE[BLOCKENTRY].SIZE then ERROR(12);
  if SYM = FORWARDSYM then
   begin
    if BLOCKKIND in [SYNC,MONI] then ERROR(3);
    BACKPATCH(STARTBLOCK,-1) (*sentinel address*);
    if COMPLETING then ERROR(3); GETSYM
   end
```

```
  else
   begin (*normal block*)
    if BLOCKKIND = PROC then
     if COMPLETING then TABLE[BLOCKENTRY].DEFINED := TRUE
     else TABLE[TX0].DEFINED := TRUE;
    repeat
     if SYM = CONSTSYM then
       CONSTDECLARATION([CONDSYM ,VARSYM ,PROCSYM,
                         BEGINSYM ,ENTRYSYM ,SYNCSYM]);
     if SYM = VARSYM then
       VARDECLARATION([PROCSYM, BEGINSYM,CONDSYM,ENTRYSYM,SYNCSYM]);
     if SYM = CONDSYM then
       begin
         if BLOCKKIND<>MONI then ERROR(51);
         CONDDECLARATION([PROCSYM,BEGINSYM]);
       end;
     if SYM=ENTRYSYM then
       begin
        if BLOCKKIND<>SYNC then ERROR(61);
        ENTRYDECLARATION([PROCSYM,SYNCSYM,BEGINSYM]);
       end;
     if (BLOCKKIND=PROG) then GLOBALADDRESS:=ADDRESS;
     while (SYM = PROCSYM) or (SYM=SYNCSYM) do
      begin
       if SYM=SYNCSYM then SYNCDECLARATION else  PROCDECLARATION;
      end;
     if TABLES then LISTABLE (*for demonstration purposes*);
     TEST([BEGINSYM], FOLLOWERS + BLOCKBEGSYS + STATBEGSYS, 34)
    until SYM in STATBEGSYS+FOLLOWERS;

    if (BLOCKKIND=PROG) or (BLOCKKIND=MONI) then
     begin
      BACKPATCH(PREVIOUS,NEXTADDRESS);
      PREVIOUS:=PRESENT;
      TABLE[TX0].SIZE:=TX;
     end
    else
      BACKPATCH(STARTBLOCK,NEXTADDRESS) (*Jump to code for this block*);
   if BLOCKKIND=SYNC then TABLE[TX0].MIN:=TX;
   (*for searching forward, so we know where the synchroniser ends*)
   if ((BLOCKKIND=PROG) or (BLOCKKIND=MONI)) and (MOREMONITORS) then
     begin
      if (BLOCKKIND <> PROG) then OPENSTACKFRAME(ADDRESS-GLOBALADDRESS);
     end
   else
    begin
     if SYNCHRON then STORELABEL(SIZEREQUIRED);
     OPENSTACKFRAME(ADDRESS) (*Reserve space for variables*);
    end;

   COMPOUNDSTATEMENT(FOLLOWERS);
```

```
    if (BLOCKKIND=MONI) then
     begin   (*to make variables read only*)
      for I:= (TX0+1) TO TX do
       begin   (*for*)
        if TABLE[I].KIND=VARIABLE then TABLE[I].CANCHANGE:=FALSE;
        TABLE[I].INSIDE:=TRUE;
       end;    (*for*)
      for I:= STARTOFMAINVAR to ENDOFMAINVAR do TABLE[I].CANCHANGE:=TRUE;
      NEWGLOBALS:=TX-TX0;
      GLOBALADDRESS:=ADDRESS;
      MOREMONITORS:=TRUE;
     end;    (* to make variables read only*)
     if (BLOCKKIND=SYNC) then
      begin
       for I:=(TX0+1) to TX do
        TABLE[I].INSIDE:=TRUE;(*can't access them*)
       NEWGLOBALS:=TX-TX0;
      end;

      if TABLE[TX0].ACCESS then (*For leaving a monitor procedure*)
       begin
        I:=TX0;
        while (TABLE[I].KIND<>MONI) and (I<>0) do I:=I-1;
        (*the I <> 0 is for incorrectly declared starred procedures
          that will generate compile  errors but  prevents a  value
          range error here*)
        LEAVINGMONITOR(TABLE[I].UNIQUE);
       end;
     LEAVEBLOCK(BLOCKKIND, LEV, PARAMS)
    end (*normal block*);

  TEST(FOLLOWERS + [SEMICOLON], [], 35);
  for I  := TX0 to TX do with TABLE[I ] do
   if (not(DEFINED)) and (not(NOWARN)) then
      WRITELN(OUTPUT, 'WARNING ', NAME, ' may not be defined');
 end (*BLOCK*);

 begin   (*PROGRAMME*)
   INITIALISE;
   ACCEPT(PROCSYM, 36);
   if SYM = IDENT then GETSYM else ERROR(6);
   with TABLE[1] do
     begin   (*Enter program name*)
      NAME := ID; KIND := PROG; LEVEL := 0; SIZE := 0; MIN := 0;
      ADR := 0; CANCHANGE := FALSE; DEFINED := TRUE; INSIDE:=TRUE;
      ACCESS:=FALSE; UNIQUE:=0;
     end;
   with TABLE[0] do INSIDE:=FALSE;
   ACCEPT(SEMICOLON, 2);
   BLOCK([PERIOD], 1, 1, PROG, TRUE, 1); (*Analyse program*)
   if SYM <> PERIOD then ERROR(37);
 end (*PROGRAMME*);

end (*COMPILER unit*).
```

```
(*$S+*)
program CONCOMPILER(INPUT, OUTPUT, OBCODE);

uses (*$U :UNIT20A.CODE *) TEXTFILES,
     (*$U :DEC20A.CODE  *) DECLARATIONS,
     (*$U :INIT20A.CODE *) COMPILER;

(*++++++++++++++++++++++++++++ Interpreter +++++++++++++++++++++++++++*)

segment procedure INTERPRET;
 const
  STACKMAX = 3500;     (*max size of the stack*)
  STEPMAX = 8          (*max before switch*);
  MONMAX1= 16;         (*MONMAX + 1*)
  PRMAX1 = 11;         (*PRMAX + 1*)
 type
  TYPEOFQUE=(NORMAL,TEMPORARY); (*Getfirst temporary-monitorque normal*)
  PTYPE = 0 .. PRMAX1;
  PTYPE2= 0..20;  (* 2*PRMAX *)
  LINK=^DESCRIPTOR;
  DESCRIPTOR= record
               NUMBER:INTEGER;   (*holds the VALUE for variable backup*)
               NEXT:LINK;
               PRIORITY:INTEGER;     (*the ADDRESS for variable backup*)
               UNIK:INTEGER; (*only for variable backup-unique monitor*)
              end;
  QUEUES=ARRAY[1..MONMAX] of LINK;
var
  INPRINPUT:TEXT;                      (*input file for the interpretter*)
  PS: (RUNNING, FINISHED, STKCHK, DATCHK, EOFCHK, DIVCHK, INXCHK,
       PRCCHK, DEDCHK, SEMCHK, PRICHK, CONCHK, SELCHK) (*Status*);
  S: array [0..STACKMAX] of INTEGER     (*Stack memory*);
  L1, L2, L3: INTEGER                 (*work variables*);
  INCR     (*stack increment as processes are launched*),
  OLDT  (*preserve top-of-stack*): INTEGER ;
  NPR                 (*Number of concurrent processes*),
  PROCACTIVE              (*number of active processes*),
  PREVPROC(*previous process*),CURPR (*current process*) : PTYPE;
  STEPS : INTEGER      (*number of steps before switch*);
  SWITCHING,                (*whether switching or not*)
  PROCTRACING, TRACING,                 (*for debugging*)
  PFLAG : BOOLEAN             (*concurrent call flag*);
  AVAILABLEMONITORS:SET of 1..MONMAX;
  ELEMENT:LINK;
  MONITORQUE:QUEUES;  (*queue waiting for execlusivity*)
  ENTRYQUE: ARRAY[1..ENTRYMAX] of LINK;
  NEXTAVAIL:LINK;               (*for the CREATE and DESTROY routines*)
  GETFIRST:QUEUES;               (*temp. queue for signalling process*)
  CONDVARQUE: ARRAY[1..CONDMAX] OF LINK;  (*condition variable queues*)
  HEAP: ^INTEGER;                       (*to mark and release the heap*)
```

```
  PTAB : array [PTYPE] of
                record
                  P, B, T: INTEGER(*Prog. counter, base, stack pointer*);
                  DISPLAY: array [1..LEVMAX] of INTEGER;
                  STACKEND: INTEGER;
                  SUSPEND: INTEGER;           (*Ø or index of semaphore*)
                  ACTIVE: BOOLEAN;                (*process active flag*)
                  EXCLUSSET,HELDSET: SET of 1..MONMAX;   (*exclus. held*)
                  NOOFELEMENTS:Ø..MONMAX;   (*no. of exclusivities held*)
                  SKIP: INTEGER; (*skip the next LMN instruction or not*)
                  RENDEZ:INTEGER;  (*which rendezvous we are performing*)
                  VARSTACK: LINK;   (*queue for backing up of variables*)
                  SAVEMARK: Ø..STACKMAX;
                 end (*PTAB*);

procedure INPRTEXTINPUT (var INPRINPUT:TEXT; PROMPT:STRING);
(*Open INPRINPUT from console or named file*)
 const ESCAPE = 27 (*ascii for <esc>*);
 var FINISHED: BOOLEAN;
     FILENAME: STRING;
  begin
   FINISHED := FALSE;
   repeat
    WRITE('What ',PROMPT,' file (<RET> for CONSOLE:
          -<ESC-RET> to abandon)? ');
    READLN(FILENAME);
    if LENGTH(FILENAME)=Ø then
     begin FINISHED := TRUE; RESET(INPRINPUT,'CONSOLE:') end
     else begin
           if (FILENAME[1]=CHR(ESCAPE)) then EXIT(program);
           (*$I- turn off IO-checks *) RESET(INPRINPUT,FILENAME);
           if IORESULT=Ø then FINISHED:=TRUE
             else  if  POS('.text',FILENAME)+POS('.TEXT',FILENAME)=Ø
                     then begin
                           FILENAME:=CONCAT(FILENAME,'.TEXT');
                           RESET(INPRINPUT,FILENAME);
                           FINISHED:=IORESULT=Ø
                          end
          end;
     if not FINISHED then
      begin WRITELN; WRITELN('No such file. Try again') end
    until FINISHED (*$I+ turn IO checks back on*);
 end (*INPRTEXTINPUT*);

 procedure CREATE(var AVAIL:LINK);
 (*act as NEW unless space has been recovered*)
  begin
   if NEXTAVAIL=NIL then new(AVAIL)
   else begin AVAIL:=NEXTAVAIL; NEXTAVAIL:=NEXTAVAIL^.NEXT; end;
  end; (*CREATE*)
```

```
procedure DESTROY(CURRENT:LINK);
 (*instead of DISPOSE-as not supported*)
 begin
  CURRENT^.NEXT:=NEXTAVAIL; NEXTAVAIL:=CURRENT;
 end;   (*DESTROY*)

procedure CHOOSEPROCESS;
(*from previous process search circularly for an active,
  unsuspended process*)

  procedure ALTER;
   begin
    CURRPR:=CURRPR+1;
    if CURRPR > PRMAX then CURRPR:=1;
   end;

begin
  ALTER;
  while (CURRPR<>PREVPROC) and ((not(PTAB[CURRPR].ACTIVE))or
                                (PTAB[CURRPR].SUSPEND<>Ø)) do
   begin ALTER; end;
  if (CURRPR=PREVPROC) and (not PTAB[CURRPR].ACTIVE) then PS:=DEDCHK
  else
   begin
    PREVPROC:=CURRPR;
    STEPS:= random mod STEPMAX + 1;
   end;
  if TRACING then
   WRITELN('Choose   ',CURRPR, ' for next ',STEPS, ' steps');
end (*CHOOSEPROCESS*);

procedure DECTBY(I:INTEGER);
(*Decrement stack pointer*)
 begin with PTAB[CURRPR] do T := T-I end;

procedure INCTBY(I:INTEGER);
(*Increment stack pointer*)
 begin
  with PTAB[CURRPR] do
   begin  T := T+I; if T > STACKEND-3 then PS := STKCHK end
 end;

procedure CHECKDATA;
(*Check "numeric" data for validity*)
 begin
   while not EOF(INPRINPUT) and (INPRINPUT^=' ') do GET(INPRINPUT);
  if EOF(INPRINPUT) then PS := EOFCHK
  else
   if ((INPRINPUT^<'Ø') or (INPRINPUT^>'9')) and (INPRINPUT^<>'+')
      and (INPRINPUT^<>'-') then PS := DATCHK
 end;
```

```
procedure POSTMORTEM;
  begin
   WRITELN(OUTPUT); WRITE(OUTPUT,'**** ');
   case PS of
    DIVCHK: WRITE(OUTPUT,'Division by zero');
    EOFCHK: WRITE(OUTPUT,'No more data');
    DATCHK: WRITE(OUTPUT,'Invalid data');
    STKCHK: WRITE(OUTPUT,'Stack overflow');
    INXCHK: WRITE(OUTPUT,'Subscript out of range');
    PRCCHK: WRITE(OUTPUT,'Missing routine');
    DEDCHK: WRITE(OUTPUT,'Deadlock');
    SEMCHK: WRITE(OUTPUT,'Semaphore with no concurrent processes');
    PRICHK: WRITE(OUTPUT,'Priority < 0');
    CONCHK: WRITE(OUTPUT,'Concurrency not in operation');
    SELCHK: WRITE(OUTPUT,'No valid Select guard');
   end;
   WRITELN(OUTPUT, ' at ', PTAB[CURPR].P-1:1, ' in process ',CURPR:1)
  end (*POSTMORTEM*);

 procedure STACKDUMP(MAX: INTEGER);
  var LOOP: INTEGER;
   begin (*Dump stack and display - useful for debugging*)
    with PTAB[CURPR] do
     begin
       WRITELN(OUTPUT);
       WRITELN(OUTPUT,'Stack dump at ', P-1:1, ' T= ', T:1, ' B= ',B:1,
               ' Return address= ', S[B+2]:1, ' Process= ', CURPR:1);
       WRITE(OUTPUT,'Display ');
       for LOOP := 1 to MAX do WRITE(OUTPUT, DISPLAY[LOOP], ' ');
       WRITELN(OUTPUT);
       for LOOP := 0 to T do
        begin
         WRITE(OUTPUT,LOOP:4, ':', S[LOOP]:5);
         if (LOOP+1) mod 8=0 then WRITELN(OUTPUT);
        end;
       WRITELN(OUTPUT)
     end (*with*)
  end (*STACKDUMP*);

 procedure SIGNAL;
  begin
   if CURPR = 0 then PS := SEMCHK else
   with PTAB[CURPR] do
    begin
      L1 := S[T]; DECTBY(1); L2 := PRMAX+1; L3 := RANDOM mod L2;
      while (L2 >= 0) and (PTAB[L3].SUSPEND <> L1) do
        begin L3 := (L3+1) mod (PRMAX+1); L2 := L2 - 1 end;
      if L2 < 0 then S[L1] := S[L1] + 1
      else begin PROCACTIVE:=PROCACTIVE+1; PTAB[L3].SUSPEND := 0;end;
    end;
  end (*SIGNAL*);
```

```
procedure WAIT;
 begin
  if CURPR = Ø then PS := SEMCHK else
  with PTAB[CURPR] do
   begin
     L1 := S[T]; DECTBY(1);
     if S[L1] > Ø then S[L1] := S[L1] - 1
      else
       begin SUSPEND := L1; STEPS := Ø;PROCACTIVE:=PROCACTIVE-1; end;
   end;
 end (*WAIT*);

procedure UNSTACKVARIABLES(PR:PTYPE; U:INTEGER);
 var PNT:LINK;
  begin
   with PTAB[PR] do
    begin   (*with ptab*)
     while (VARSTACK<>NIL) and (U=VARSTACK^.UNIK) do
      begin   (*restore variables*)
       PNT:=VARSTACK;
       S[VARSTACK^.PRIORITY] := VARSTACK^.NUMBER;
       VARSTACK := VARSTACK^.NEXT;
       DESTROY(PNT);
      end;
     while VARSTACK<>NIL do
      begin  (*clear queue - missing restore*)
       PNT := VARSTACK;
       VARSTACK := VARSTACK^.NEXT;
       DESTROY(PNT);
      end;
    end;   (*with ptab*)
  end;  (*unstackvariables*)

procedure DEQUEUEPROCESS(U:INTEGER);
 var
    P1,POINT,LASTP:LINK;
    LC:1..MONMAX;   (*loopcounter*)

    procedure READYPROCESS;
      begin
       with POINT^ do
        begin
         with PTAB[NUMBER] do
          begin
           if (VARSTACK<>NIL) and (VARSTACK^.UNIK=Ø) then
            (*nested backup*) UNSTACKVARIABLES(NUMBER,Ø);
           PROCACTIVE:=PROCACTIVE+1;
           ACTIVE:=TRUE;
           HELDSET:= [];
          end;   (*with*)
        end;   (*with POINT^*)
      end;   (*READYPROCESS*)
```

```
     procedure UPDATEQUEUE(QUE:TYPEOFQUE);
      var DISP:LINK;
       begin
        if QUE=TEMPORARY then
         begin
          DISP:=GETFIRST[U];
          GETFIRST[U]:=DISP^.NEXT;(*act like a stack F.I.L.O*)
         end
        else
         begin
          DISP:=MONITORQUE[U];
          MONITORQUE[U]:=DISP^.NEXT;
         end;
        DESTROY(DISP);
        PTAB[POINT^.NUMBER].HELDSET:= PTAB[POINT^.NUMBER].HELDSET+[U];
       end;

 begin (*DEQUEUEPROCESS*)
  if GETFIRST[U] <> NIL then
   begin
    POINT:=GETFIRST[U];
    with POINT^ do
     begin
      with PTAB[NUMBER] do
       begin
        UPDATEQUEUE(TEMPORARY);
        if HELDSET=EXCLUSSET then
         READYPROCESS;
       end;
     end;
    end   (*getfirst<>nil*)
   else
    begin
     POINT:=MONITORQUE[U];
     with POINT^ do   (*P is not nil *)
      begin   (*with*)
       with PTAB[NUMBER] do
        begin   (*with ptab[number]*)
         if PRIORITY=0 then
          begin   (*if*)
           if EXCLUSSET= (HELDSET + [U]) then
            begin
             UPDATEQUEUE(NORMAL);
             READYPROCESS;
            end
           else
             UPDATEQUEUE(NORMAL);
          end   (*if*)
```

```
else
 begin   (*else1*)
  if EXCLUSSET=[] then
   begin
    UPDATEQUEUE(NORMAL);
    EXCLUSSET:=EXCLUSSET+[U];
    NOOFELEMENTS:=NOOFELEMENTS+1;
    READYPROCESS;
   end
  else
   begin   (*else2*)
    for LC:=1 to MONICOUNT do
     begin   (*for*)
      if LC in EXCLUSSET then
       begin
        if LC in AVAILABLEMONITORS then
         begin
          AVAILABLEMONITORS:=AVAILABLEMONITORS - [LC];
          HELDSET:= HELDSET + [LC];
         end
        else
         begin
          P1:=MONITORQUE[LC]; LASTP:=P1;
          while (P1 <> NIL) and (P1^.PRIORITY=Ø) do
           begin
            LASTP:=P1;
            P1:=P1^.NEXT;
           end;   (*while*)
          CREATE(ELEMENT);
          ELEMENT^.PRIORITY:=Ø; ELEMENT^.NEXT:= NIL;
          ELEMENT^.NUMBER:= NUMBER; (*p^.number*)
          if LASTP=P1 then (*ie. at the front of queue*)
           begin
            ELEMENT^.NEXT:=P1;
            MONITORQUE[LC]:= ELEMENT;
           end
          else
           begin
            LASTP^.NEXT:=ELEMENT;
            ELEMENT^.NEXT:= P1;
           end;
         end;
       end;   (*if lc in exclusset*)
     end;   (*for*)
     if EXCLUSSET = HELDSET then
       begin
        UPDATEQUEUE(NORMAL);
        EXCLUSSET:=EXCLUSSET+[U];
        NOOFELEMENTS:=NOOFELEMENTS+1;
        READYPROCESS;
       end
```

```
              else
                begin
                 EXCLUSSET:=EXCLUSSET+[U];(*for further priorities*)
                 NOOFELEMENTS:=NOOFELEMENTS+1;
                 UPDATEQUEUE(NORMAL);
                end;
           end;   (*else2*)
         end;   (*else1*)
       end;   (*with ptab[number]*)
     end;   (*with point^*)
    end;   (*getfirst=nil*)
  end;   (*DEQUEUEPROCESS*)

procedure QUEUEPROCESS(U,PR:INTEGER);
 var POINT,LASTP:LINK;
  begin
   CREATE(ELEMENT);
   ELEMENT^.NUMBER:=PR; ELEMENT^.NEXT:= NIL;
   ELEMENT^.PRIORITY:=MONMAX1 - PTAB[PR].NOOFELEMENTS;
   PTAB[PR].HELDSET:= []; (*release all monitors*)
   POINT:=MONITORQUE[U]; LASTP:=POINT;
   while (POINT <> NIL) and (POINT^.PRIORITY <= ELEMENT^.PRIORITY) do
    begin
     LASTP:=POINT;
     POINT:= POINT^.NEXT;
    end;
   if POINT=LASTP then (*ie. queue empty or at the beginning*)
    begin
     MONITORQUE[U]:= ELEMENT;
     ELEMENT^.NEXT:=POINT;
    end
   else
    if POINT = NIL then (*ie. at the end of the queue*)
     LASTP^.NEXT:= ELEMENT
    else   (*ie. in the interior of the queue*)
     begin
      LASTP^.NEXT:=ELEMENT;
      ELEMENT^.NEXT:= POINT;
     end;
  end;   (*QUEUEPROCESS*)

procedure STACKVARIABLES(U:INTEGER; CW:BOOLEAN);
 var LC: Ø..MONMAX;
     AD: INTEGER;
     PNT:LINK;
  begin   (*stackvariables*)
   with PTAB[CURPR] do
    begin   (*with ptab*)
```

```
      if CW (*conditioned wait*) then
       begin (*perform the conditioned as well as nested backup*)
        for AD := T downto SAVEMARK do
         begin
          PNT:=VARSTACK;
          CREATE(VARSTACK);
          VARSTACK^.PRIORITY:=S[AD]; (*address*)
          VARSTACK^.NUMBER:= S[S[AD]]; (*value*)
          VARSTACK^.UNIK:=U;
          VARSTACK^.NEXT:=PNT;
         end;   (*for ad*)
        T:=SAVEMARK-1;
       end;  (*if cw*)
     if  not(NOBACKUP) then
      for LC:=1 to MONICOUNT do
       begin   (*nested backup*)
        if (LC<>U) and (LC in EXCLUSSET) then
         begin (*for conditioned backup - not all of present monitor*)
          for AD:= MONIVARADR[LC,1] to MONIVARADR[LC,2] do
           begin
            PNT:=VARSTACK;
            CREATE(VARSTACK);
            VARSTACK^.PRIORITY:=AD;  (*address*)
            VARSTACK^.NUMBER:= S[AD];(*value*)
            VARSTACK^.UNIK:=U;
            VARSTACK^.NEXT:=PNT;
           end;   (*for ad*)
         end;  (*if lc*)
       end;   (*for lc*)
     end;   (*with ptab*)
   end;   (*stackvariables*)

  procedure RELEASEEXCLUSIVITIES(PR:PTYPE);
   var LC:1..MONMAX;
    begin
     with PTAB[PR] do
      begin
       PROCACTIVE:=PROCACTIVE-1;
       ACTIVE:=FALSE;
       STEPS:=Ø;
       for LC:=1 to MONICOUNT do
        begin   (*for*)
         if LC in EXCLUSSET then
          begin   (*if*)
           if (MONITORQUE[LC] = NIL) and (GETFIRST[LC] = NIL) then
            AVAILABLEMONITORS:=AVAILABLEMONITORS + [LC]
           else
            DEQUEUEPROCESS(LC);
          end;   (*if*)
        end;   (*for*)
      end;   (*with*)
    end;   (*releaseexclusivities*)
```

```
procedure EXCLUSIVITY(U:INTEGER);
 var LC:1..MONMAX;
  begin
   if CURPR<>0 then (*ie. concurrency active*)
    begin
     with PTAB[CURPR] do
      begin   (*with*)
      if U in AVAILABLEMONITORS then
       begin
        NOOFELEMENTS:=NOOFELEMENTS+1;
        EXCLUSSET:=EXCLUSSET + [U];
        AVAILABLEMONITORS:=AVAILABLEMONITORS - [U];
       end
      else
       begin
        if (EXCLUSSET<>[]) then
         (*nested monitor call - backup monitor variables*)
         STACKVARIABLES(0,FALSE);
        QUEUEPROCESS(U (*to index the monitor array*),CURPR);
        RELEASEEXCLUSIVITIES(CURPR);
       end;
      end;   (*with*)
    end   (*if curpr<>0*)
  end;   (*EXCLUSIVITY*)

 procedure LEAVEMONITOR(U:INTEGER);
  begin
   if CURPR <> 0 then     (*ie. concurrency active*)
    begin
     with PTAB[CURPR] do
      begin   (*with*)
       if SKIP > 0 then   (*ignore LMN instruction*)
        SKIP:= SKIP - 1
       else
        begin
         EXCLUSSET:=EXCLUSSET - [U];
         NOOFELEMENTS:=NOOFELEMENTS-1;
         if (MONITORQUE[U] = NIL) and (GETFIRST[U] = NIL) then
           AVAILABLEMONITORS:=AVAILABLEMONITORS + [U]
         else
          begin
           DEQUEUEPROCESS(U);
          end;
        end;   (*else - ie.SKIP > 0*)
      end;   (*with*)
    end;   (*if curpr<>0*)
  end;   (*LEAVEMONITOR*)
```

```
  procedure LENGTHOFQUEUE(C:INTEGER);
   var COUNT:INTEGER;
       LAST:LINK;
    begin
     COUNT:=0; LAST:=CONDVARQUE[C];
     while LAST <> NIL do
      begin
       COUNT:=COUNT+1;
       LAST:=LAST^.NEXT;
      end;
     INCTBY(1); S[PTAB[CURPR].T]:=COUNT;
    end;   (*lengthofqueue*)

  procedure CONDWAIT(PRIOR,C:INTEGER);
   var POINT,LASTPOINT:LINK;
       LC:1..MONMAX;
    begin
     CREATE(ELEMENT);
     with ELEMENT^ do
      begin
       NEXT:=NIL; PRIORITY:=PRIOR;
       if PRIORITY<0 then PS:=PRICHK;
       NUMBER:=CURPR; POINT:=CONDVARQUE[C]; LASTPOINT:=POINT;
       while (POINT<>NIL) and (POINT^.PRIORITY<=PRIORITY) do
        begin
         LASTPOINT:=POINT; POINT:=POINT^.NEXT;
        end;
       if LASTPOINT=POINT then
        begin CONDVARQUE[C]:=ELEMENT; NEXT:=POINT; end
       else
        begin LASTPOINT^.NEXT:=ELEMENT; NEXT:=POINT; end;
      end;   (*with element*)
     RELEASEEXCLUSIVITIES(CURPR);
    end;   (*condwait*)

 procedure CONDSIGNAL(U,C:INTEGER);
  var LC:INTEGER;   (*loop counter*)
      POINT:LINK;
      PR:PTYPE;

  procedure STOREPROCESS(NUM:PTYPE; POS:INTEGER);
   begin
    CREATE(ELEMENT);
    ELEMENT^.NUMBER:=NUM; ELEMENT^.PRIORITY:=0; (*dummy*)
    ELEMENT^.NEXT:=GETFIRST[POS]; (*act like a stack F.I.L.O*)
    GETFIRST[POS]:=ELEMENT;
   end;

  procedure RESTARTPROCESS;
   begin
    with PTAB[PR] do
     begin PROCACTIVE:=PROCACTIVE+1; HELDSET:=[]; ACTIVE:=TRUE; end;
   end;
```

```
procedure REMOVEITEM;
 var DISP:LINK;
  begin
   DISP:=CONDVARQUE[C];
   CONDVARQUE[C]:=DISP^.NEXT;
   DESTROY(DISP);
  end;

begin   (*condsignal*)
 if CONDVARQUE[C]<>NIL then
  begin   (*signal a process & temporarily suspend itself*)
   PR:=CONDVARQUE[C]^.NUMBER;
   with PTAB[PR] do
    begin
     PROCACTIVE:=PROCACTIVE-1;
     PTAB[CURPR].ACTIVE:=FALSE; STEPS:=Ø;   (*suspend*)
     for LC:=1 to MONICOUNT do
      begin   (*for*)
         if (LC in PTAB[CURPR].EXCLUSSET) then
           PTAB[CURPR].HELDSET:=PTAB[CURPR].HELDSET+[LC];
           (*so the signalling process will know when to continue*)
         if (LC in EXCLUSSET) and (LC in AVAILABLEMONITORS) then
          begin
           AVAILABLEMONITORS:=AVAILABLEMONITORS-[LC];
           HELDSET:=HELDSET+[LC];
          end;
         if (LC in EXCLUSSET) and (LC in PTAB[CURPR].EXCLUSSET) then
          begin
           HELDSET:=HELDSET + [LC];
           PTAB[CURPR].HELDSET:=PTAB[CURPR].HELDSET-[LC];
           STOREPROCESS(CURPR,LC);
          end;
     end;   (*for*)
    if EXCLUSSET=HELDSET then
      RESTARTPROCESS
   else
    begin   (*else1*)
     if GETFIRST[U]^.NEXT <> NIL then
      begin
       POINT:=GETFIRST[U]^.NEXT;
       while POINT<>NIL do
        begin   (*while*)
         for LC:= 1 to MONICOUNT do
          begin
           if (LC in EXCLUSSET) and
              (LC in PTAB[POINT^.NUMBER].HELDSET) then
            begin
             HELDSET:=HELDSET + [LC];
             PTAB[POINT^.NUMBER].HELDSET:=PTAB[POINT^.NUMBER].HELDSET
                                          -[LC];
             STOREPROCESS(POINT^.NUMBER,LC);
            end;   (*if*)
          end;   (*for*)
```

```
          if EXCLUSSET=HELDSET then
           POINT:=NIL   (*jump out of while loop*)
          else
           POINT:=POINT^.NEXT;
         end;   (*while*)
        end;   (*if getfirst[u]<>nil*)
     if EXCLUSSET=HELDSET then
       RESTARTPROCESS
     else
      begin   (*else2*)
       if (MONITORQUE[U]=NIL) or (MONITORQUE[U]^.PRIORITY<>0) then
        begin
         for LC:=1 to MONICOUNT do
          begin
           if not(LC in HELDSET) and (LC in EXCLUSSET) then
            STOREPROCESS(PR,LC);
          end;   (*for*)
        end
       else
        begin  (*get as many exclusivities as possible*)
         POINT:=MONITORQUE[U];
         while (POINT<>NIL) and (POINT^.PRIORITY=0) do
          begin
           for LC:=1 to MONICOUNT do
            begin
             if (LC in EXCLUSSET)and(LC in PTAB[POINT^.NUMBER].HELDSET)
             then
              begin
               HELDSET:=HELDSET + [LC];
               PTAB[POINT^.NUMBER].HELDSET:=PTAB[POINT^.NUMBER].HELDSET
                                                            - [LC];
               STOREPROCESS(POINT^.NUMBER,LC);
              end;
            end;   (*for*)
           if HELDSET=EXCLUSSET then
            POINT:=NIL (*jump out*)
           else
           POINT:=POINT^.NEXT;
          end;   (*while*)
         if EXCLUSSET=HELDSET then RESTARTPROCESS
         else
          begin
           for LC:= 1 to MONICOUNT do
            if not(LC in HELDSET) and (LC in EXCLUSSET) then
              STOREPROCESS(PR,LC);
          end;  (*else*)
       end;   (*else get as many exclusivities as possible*)
      end;   (*else2*)
     end;   (*else1*)
    end;   (*with*)
   REMOVEITEM;
  end;   (*if condvarque[c]<>nil*)
 end;   (*condsignal*)

 (*$I :CNC220A.TEXT *)   (*include file*)
```

```
procedure ACCEPTBLOCK(U:INTEGER); (*deals with accept statements*)
 var PT:LINK;
  begin
   if CURPR=0 then
    PS:=CONCHK   (*concurrency inactive*)
   else
    begin
     if ENTRYQUE[U]=NIL then
      begin   (*suspend process*)
       CREATE(ENTRYQUE[U]);
       with ENTRYQUE[U]^ do
        begin
         NUMBER:=CURPR+PRMAX;  (*fiddle-SYNCHRON procedure suspended*)
         PRIORITY:=PTAB[CURPR].P;       (*address of accept statement*)
         NEXT:=NIL;                           (*only process on queue*)
        end;   (*with*)
       PROCACTIVE:=PROCACTIVE-1;
       PTAB[CURPR].ACTIVE:=FALSE; (*suspend process*)
       STEPS:=0; (*switch processor*)
      end   (*if entry point queue empty*)
    else
     begin
      PTAB[CURPR].RENDEZ:=U;   (*ready to deal with rendezvous*)
     end;
    end;   (*else*)
  end; (*acceptblock*)

procedure ENDACCEPTBLOCK(I:INSTRUCTION);
 var PT:LINK;
     U:INTEGER;
  begin
   with I do
    begin   (*with*)
     U:=S[PTAB[CURPR].T];             (*unique no. on top of stack*)
     DECTBY(1);
     PTAB[CURPR].RENDEZ:=0;   (*no longer dealing with rendezvous*)
     PT:=ENTRYQUE[U]; (*can't be nil*)
     ENTRYQUE[U]:=PT^.NEXT;
     with PTAB[PT^.NUMBER] do
      begin
       T:=B-A;                 (*same as RETURN-reset stack segment*)
       B:=S[B+1];
       PROCACTIVE:=PROCACTIVE+1;
       ACTIVE:=TRUE;  (*reactivate*)
      end;
     DESTROY(PT);
    end;   (*with I*)
  end;   (*endacceptblock*)
```

```
procedure RENDEZVOUS(I:INSTRUCTION);
 var PT,LPT,ELEMENT:LINK;
     IMPLICITSIGNAL:BOOLEAN;
     LC:INTEGER;   (*loop counter*)
     INDEX:INTEGER;
  begin
   with I,PTAB[CURPR] do
    begin  (*with*)
     S[T+2]:=B; S[T+3]:=P;
     B:=T+1;
     T:=T+3;    (*alter the stack-same as the INT instr*)
     if T > STACKEND-3 then PS:=STKCHK;
     IMPLICITSIGNAL:=FALSE;
     PT:=ENTRYQUE[A]; LPT:=PT;
     while (PT <> NIL) do
      begin
       if PT^.NUMBER > PRMAX then IMPLICITSIGNAL:=TRUE;
       LPT:=PT;
       PT:=PT^.NEXT;
      end;    (*while*)
     CREATE(ELEMENT);
     ELEMENT^.PRIORITY:=0;  (*open to alteration?*)
     ELEMENT^.NEXT:=NIL;
     ELEMENT^.NUMBER:=CURPR;
     if PT=LPT then  (*queue empty*)
      ENTRYQUE[A]:=ELEMENT
     else
      begin
       LPT^.NEXT:=ELEMENT;
       if IMPLICITSIGNAL then
        begin  (*reactivate synchroniser at head of queue*)
         PROCACTIVE:=PROCACTIVE+1;
         INDEX:=ENTRYQUE[A]^.NUMBER-PRMAX;
         PTAB[INDEX].ACTIVE:=TRUE;
         PTAB[INDEX].RENDEZ:=A;
         PTAB[INDEX].P:=ENTRYQUE[A]^.PRIORITY;
         if PTAB[INDEX].HELDSET <> [] then
          (*sync suspended on a select-remove from all other queues*)
          begin
           for LC:= 1 to ENTRYCOUNT do
            begin
             if LC in PTAB[INDEX].HELDSET then
              begin (*remove from queue*)
               PT:=ENTRYQUE[LC];
               ENTRYQUE[LC]:=ENTRYQUE[LC]^.NEXT;
               DESTROY(PT);
              end; (*if*)
             end;  (*for*)
            PTAB[INDEX].HELDSET:=[];
           end  (*if HELDSET<>[]*)
```

```
            else
             begin
              PT:=ENTRYQUE[A];
              ENTRYQUE[A]:=ENTRYQUE[A]^.NEXT;
              DESTROY(PT);
             end;  (*ie. HELDSET=[]*)
          end;   (*if*)
        end;   (*else*)
      PROCACTIVE:=PROCACTIVE-1;
      ACTIVE:=FALSE;    (*suspend*)
      STEPS:=0;         (*process switch*)
    end;   (*with*)
  end; (*rendezvous*)

 procedure SELECTACCEPT(I:INSTRUCTION);
  const MAXGUARD=20;         (*max. no. of guard conditions per select*)
  var START,STOP:INTEGER;
      SUB,LC,LC1,LC2:INTEGER;
      SELTABLE:ARRAY[1..MAXGUARD] of INTEGER; (*table of valid guards*)
      FOUND:BOOLEAN;
   begin
    with PTAB[CURPR]do
     begin
      START:=S[T]; DECTBY(1);
      STOP:=I.A;
      LC:=START;
      SUB:=0;
      while LC < STOP do
       begin  (*find valid guards*)
        if S[DISPLAY[I.L] + LC] = 1 then (*valid*)
         begin
          SUB:=SUB+1;
          SELTABLE[SUB]:=S[DISPLAY[I.L]+LC+1];(*address of the accept*)
         end;
        LC:=LC+2; (*move to the next guard result*)
       end; (*while*)
       if SUB <> 0 then
       begin (*some valid guards*)
        LC:= (RANDOM mod SUB) +1;   (*choose an arbitrary valid guard*)
        LC1:=0;
        FOUND:=FALSE;
        while not(FOUND) do
         begin (*will the accepts cause a delay?*)
          with CODE[SELTABLE[LC]] do
           begin
            if ENTRYQUE[A]=NIL then (*will cause a delay*)
             begin
              LC:=LC+1;
              LC1:=LC1+1;
              if LC>SUB then LC:=1; (*circular search*)
              if LC1=SUB then FOUND:=TRUE;
             end
```

```
          else (*found an accept that won't cause a delay*)
           begin
            P:=SELTABLE[LC]; (*set PC to start address of accept*)
            FOUND:=TRUE;
           end;
         end;  (*with CODE[SELTABLE[LC]]*)
       end; (*while not found*)
      if LC1=SUB then (*all accepts cause a delay*)
       begin
        LC2 := (random mod SUB) + 1;
        (*start at random place in SELTABLE - for identical accepts
          only one chosen for queueing - arbitrarily*)
        LC1:=0;
        while LC1 < SUB do
         begin (*suspend sync proc on all queues*)
          with CODE[SELTABLE[LC2]] do
           begin
            CREATE(ENTRYQUE[A]);  (*queue is nil*)
            with ENTRYQUE[A]^ do
             begin
              NUMBER:=CURPR+PRMAX;   (*fiddle-synchroniser*)
              PRIORITY:=SELTABLE[LC2]; (*address of accept*)
              NEXT:=NIL;
             end; (*with ENTRYQUE[A]*)
            HELDSET:=HELDSET+[A];
           end;  (*with CODE*)
           LC1:=LC1+1; LC2:=LC2+1;
           if LC2 > SUB then LC2:=1;
          end; (*while*)
         PROCACTIVE:=PROCACTIVE-1;
         ACTIVE:=FALSE; (*suspend*)
         STEPS:=0; (*switch*)
       end;  (*if LC=LC1*)
     end  (*if SUB<>0*)
    else
     begin (* Is there an ELSE clause ?*)
      if S[DISPLAY[I.L]+STOP-1]=2 then
       begin (*Yes - else clause to the select statement*)
        P:=S[DISPLAY[I.L]+STOP]; (*start address*)
       end
      else
       PS:=SELCHK;   (*run time error*)
     end;
   end; (*with PTAB[CURPR]*)
  end;  (*selectaccept*)
```

```
procedure CALL (I: INSTRUCTION);
 begin
  with I, PTAB[CURPR] do
   begin
     if A > CODEMAX then
      (*set up at compile time so as to skip the next LMN instruction
        when calling starred procedure from inside the same monitor*)
      begin
       A := A - CODEMAX;
       if CURPR <> Ø then SKIP := SKIP + 1;(*ie. only if concurrency*)
      end;
     if not PFLAG then
      begin
        S[T+1] := DISPLAY[L+1]; S[T+2] := B; S[T+3] := P;
        B := T+1; P := A; DISPLAY[L+1] := B;
      end
     else
      begin (*mark for subsequent concurrent entry*)
        NPR := NPR + 1; PROCACTIVE:=PROCACTIVE+1;
        with PTAB[NPR] do
         begin
           B := PTAB[CURPR].T+1; P := A; DISPLAY[L+1] := B; T := B-1;
           S[T+1] := DISPLAY[L+1]; S[T+2] := B; S[T+3] := Ø(*fiddle*);
           STACKEND := T + INCR; ACTIVE := TRUE;
         end;
        INCTBY(INCR)
      end (*else*)
   end (*with*)
 end (*CALL*);

procedure RETURN(I: INSTRUCTION);
begin
  with I,PTAB[CURPR] do
   begin
     if A=-1 then (*stop concurrency*)
      begin
       NPR:=Ø; PTAB[Ø].ACTIVE:=TRUE; (*reactivate main program*)
       PTAB[Ø].T:=OLDT;
      end
     else
      begin
       T := B-A; DISPLAY[L] := S[B]; P := S[B+2]; B := S[B+1];
       if P = Ø then
        begin
          NPR := NPR - 1; PROCACTIVE:=PROCACTIVE-1;
          ACTIVE := FALSE; STEPS := Ø;
          PTAB[Ø].ACTIVE := NPR = Ø;
          if PTAB[Ø].ACTIVE then PTAB[Ø].T := OLDT
        end
      end;
   end (*with*)
end (*RETURN*);
```

```
 procedure TRACEPROCESSES;   (*for debugging-used with $M+ directive*)
  var I : INTEGER;
      A : array[BOOLEAN] of CHAR;
   begin
    if PROCTRACING then
      begin
        A[FALSE] := 'I'; A[TRUE] := 'A';
        WRITE(OUTPUT, ' -- Cur Pr = ',CURPR, ' ');
        for I := Ø to PRMAX do
          begin WRITE(OUTPUT, A[PTAB[I].ACTIVE]);
                if PTAB[I].SUSPEND= Ø then WRITE(OUTPUT,'    -')
                 else WRITE(OUTPUT,PTAB[I].SUSPEND:3,'-');
          end;
      end;
    WRITELN(OUTPUT);
   end (*TRACEPROCESSES*);

 procedure NEXTSTEP;
  var LOOP: INTEGER;
      I : INSTRUCTION; (*current*)

   procedure MORE;   (*NEXTSTEP too long*)
    var PRIOR,C:INTEGER;
     begin
      if CURPR=Ø then PS:=SEMCHK
      else   (*only if concurrency is active*)
       with PTAB[CURPR] do
        begin
         C:=S[T]; DECTBY(1);
         with I do
          case F of
           QLN: LENGTHOFQUEUE(C);
           QUE: begin INCTBY(1); S[T]:=ORD(CONDVARQUE[C] <> NIL); end;
           QPW,QWT:begin
                    PRIOR:=C;
                    C:=S[T]; DECTBY(1);
                    CONDWAIT(PRIOR,C);
                   end;
           QSG:CONDSIGNAL(A,C);
          end;   (*case*)
        end;   (*with ptab[curpr]*)
     end;  (*more*)
```

```
begin (*$R-*)   (*nextstep*)
 with PTAB[CURPR] do
  begin
   I := CODE[P]; P := P+1 (*fetch*);
   with I do (*execute*)
   begin
    if F>=QLN then MORE
    else
     begin
     case F of
      NEG: S[T] := -S[T];
      ADD: begin DECTBY(1); S[T] := S[T]+S[T+1] end;
      SUB: begin DECTBY(1); S[T] := S[T]-S[T+1] end;
      MUL: begin DECTBY(1); S[T] := S[T]*S[T+1] end;
      DVD:
       begin
        DECTBY(1);
        if S[T+1]=0 then PS := DIVCHK else S[T] := S[T] div S[T+1]
       end;
      MD:
       begin
        DECTBY(1);
        if S[T+1]=0 then PS := DIVCHK else S[T] := S[T] mod S[T+1]
       end;
      OD : S[T] := ORD(ODD(S[T]));
      EQL: begin DECTBY(1); S[T] := ORD(S[T] = S[T+1]) end;
      NEQ: begin DECTBY(1); S[T] := ORD(S[T] <> S[T+1]) end;
      LSS: begin DECTBY(1); S[T] := ORD(S[T] < S[T+1]) end;
      GEQ: begin DECTBY(1); S[T] := ORD(S[T] >= S[T+1]) end;
      GTR: begin DECTBY(1); S[T] := ORD(S[T] > S[T+1]) end;
      LEQ: begin DECTBY(1); S[T] := ORD(S[T] <= S[T+1]) end;
      STK: STACKDUMP(A);
      PRN: begin WRITE(OUTPUT,S[T]); DECTBY(1) end;
      PRS:
       begin
        for LOOP := T-S[T] to T-1 do WRITE(OUTPUT,CHR(S[LOOP]));
        DECTBY(S[T]+1)
       end;
      NL : WRITELN(OUTPUT);
      INN:
       begin CHECKDATA; if PS=RUNNING then READ(INPRINPUT,S[S[T]]);
             DECTBY(1) end;
      LIT: begin INCTBY(1); S[T] := A end;
      LDA: begin INCTBY(1); S[T] := DISPLAY[L] + A end;
      LDX: S[T] := S[S[T]];
      IND: if (S[T] < 0) or (S[T] > A) then PS := INXCHK
           else begin DECTBY(1); S[T] := S[T] + S[T+1] end;
      STO: begin S[S[T-1]] := S[T]; DECTBY(2) end;
      INT: INCTBY(A);
      HLT: PS := FINISHED;
      BRN: if A < 0 then PS := PRCCHK (*missing code*) else P := A;
      BZE: begin if S[T]=0 then P := A; DECTBY(1) end;
```

```
SFL:
 begin
  L1 := S[T-1];
  if (1 - L) * (L1 - S[T]) <= Ø
   then S[S[T-2]] := L1 else begin DECTBY(3); P := A end
 end;
EFL:
 begin
  L1 := S[S[T-2]] + 1 - L;
  if (1 - L) * (L1 - S[T]) <= Ø
   then begin S[S[T-2]] := L1; P := A end else DECTBY(3)
 end;
RND: begin INCTBY(1); S[T] := RANDOM end;
RDY: begin INCTBY(1); S[T] := PROCACTIVE; end;
ACT: begin INCTBY(1); S[T] := NPR; end;
SWI: begin STEPS := Ø; end;
WGT: WAIT;
SIG: SIGNAL;
CBG:
 begin
  PFLAG := TRUE; OLDT := T;
  INCR := (STACKMAX - T) div A - PMAX;
  if INCR <= Ø then PS := STKCHK
 end;
CND: begin PFLAG := FALSE; PTAB[Ø].ACTIVE := FALSE;
           CURPR:=(random mod NPR) +1;
           STEPS:=(random mod STEPMAX)+1;
           PREVPROC:=CURPR;
     end;
SWP: SWITCHING := not SWITCHING;
CAL: CALL(I);
RET: RETURN(I);
PRC: begin WRITE(OUTPUT,CHR(S[T] mod HIGHEST)); DECTBY(1); end;
NC : if not EOF(INPRINPUT) then READLN(INPRINPUT)
      else PS:=EOFCHK;
INC: if EOF(INPRINPUT) then PS:=EOFCHK
     else
      begin
       READ(INPRINPUT,CH); S[S[T]] := ord(CH); DECTBY(1);
      end;
SMK: begin SAVEMARK:=T+1; end;
SAV: STACKVARIABLES(A,TRUE);
RES: UNSTACKVARIABLES(CURPR,A);
EXC: EXCLUSIVITY(A);
LMN: LEAVEMONITOR(A);
CHK: TRACEPROCESSES;
ACC: ACCEPTBLOCK(A);
EAC: ENDACCEPTBLOCK(I);
SCL: RENDEZVOUS(I);
SEL: SELECTACCEPT(I);
```

```
      LDE: begin   (*for entry point parameters*)
             if PTAB[CURPR].RENDEZ=0 then PS:=CONCHK
             else
              begin
               INCTBY(1);
               S[T]:=PTAB[ENTRYQUE[PTAB[CURPR].RENDEZ]^.NUMBER].B+A;
              end;  (*else*)
           end;   (*LDE*)
     end (*case*)
   end;   (*else*)
  end;   (*with I*)
 end (*with PTAB*)
end (*NEXTSTEP*);

 begin (*INTERPRET*)(*$R-*)
  MARK(HEAP);
  NEXTAVAIL:= NIL;   (*for the CREATE and DESTROY routines*)
  S[0] := 0; S[1] := 0; S[2] := 0;  PS := RUNNING;
  WRITE('Trace? '); READLN(CH); TRACING := CH in ['Y','y'];
  WRITE('Process trace? '); READLN(CH); PROCTRACING := CH in ['Y','y'];
  if ASKBACKUP then
   begin
    WRITE('Nested Backup?');  (*used with $B- directive*)
    READLN(CH); NOBACKUP:= not(CH in ['Y','y']);
   end;
  WRITELN('Memory available ',MEMAVAIL);
  TEXTOUTPUT(OUTPUT,'RESULTS');
  INPRTEXTINPUT(INPRINPUT,'DATA');
  with PTAB[0] do (*start main program*)
   begin
    (*initialise main stack frame*)
    T := -1; P := 0; B := 0; DISPLAY[1] := 0
    SUSPEND := 0; ACTIVE := TRUE; STACKEND := STACKMAX; RENDEZ:=0;
   end;
  for CURPR := 1 to PRMAX do (*all processes inactive*)
   with PTAB[CURPR] do
    begin ACTIVE := FALSE; DISPLAY[1] := 0; SUSPEND := 0;
          EXCLUSSET:=[]; HELDSET:=[]; NOOFELEMENTS:=0;
          SKIP:= 0; RENDEZ:=0; VARSTACK:=NIL; SAVEMARK:=0;
    end;
  AVAILABLEMONITORS:= [];
  for L1:= 1 TO MONICOUNT do
   begin
    AVAILABLEMONITORS:=AVAILABLEMONITORS + [L1];
    MONITORQUE[L1]:= NIL;
    GETFIRST[L1]:=NIL;
   end;
  for L1:=1 to CONDCOUNT do CONDVARQUE[L1]:=NIL;
  for L1:=1 to ENTRYCOUNT do ENTRYQUE[L1]:=NIL;
  CURPR := 0; PFLAG := FALSE; NPR := 0; STEPS := 0; SWITCHING := TRUE;
  PROCACTIVE:=0;
```

```
   repeat
    if TRACING then
     with PTAB[CURPR] do
      WRITELN(OUTPUT,CURPR,'/',P,' ',MNEMONIC[CODE[P].F]);
    NEXTSTEP;
    if BREAKIN then PS := FINISHED;
    if PS = RUNNING then
     if PTAB[Ø].ACTIVE then CURPR := Ø
      else if SWITCHING then
       if STEPS = Ø then CHOOSEPROCESS else STEPS := STEPS - 1;
   until PS <> RUNNING;
   if PS <> FINISHED then
    begin
     POSTMORTEM; PROCTRACING:=TRUE;
     WRITELN(OUTPUT); TRACEPROCESSES;
    end;
   CLOSE(INPRINPUT);
   RELEASE(HEAP);
  end (*INTERPRET*);


  begin (*MAIN PROGRAM*)
   TEXTINPUT('SOURCE'); TEXTOUTPUT(OUTPUT,'LISTING');
   PROGRAMME;
   CLOSE (OUTPUT,LOCK);
   if ERRORS then WRITELN('Compilation errors')
   else
    begin
     WRITELN('[',NOOFLINES,'] Lines Compiled Correctly');
     if OBLIST then LISTCODE;
     CLOSE(INPUT);
     while TRUE do
      begin
       WRITELN('Executing');
       INTERPRET;
       CLOSE(OUTPUT,LOCK); CLOSE(INPUT)
      end
    end
end (*COMPILER*).
```