Process synchronisation in MASCOT*

H. R. Simpson† and K. Jackson‡

The MASCOT approach to programming is concerned with the build up of software in a computer, its run time operation and the testing of individual facilities. A central feature of the approach is the manner in which certain synchronisation primitive procedures and control variables are used to achieve a high degree of modularity. This paper describes MASCOT software structure and synchronisation, and includes several examples to illustrate this style of programming.

(Received January 1978)

1. Introduction

The acronym MASCOT stands for 'Modular Appproach to Software Construction, Operation and Test' (Jackson and Simpson, 1975) and identifies the more important characteristics of a line of research started in 1972. Experience gained from the design and implementation of a large real time system had emphasised the need for software modularity in three separate but interdependent aspects which are represented by the words 'construction', 'operation' and 'test'. In this context the term 'software construction' is used to denote the process of converting program source text into the total software content of a computer; it embraces individual functions such as compilation, loading and linking which are sequentially combined to produce executable code resident in a machine. 'Operation' refers to the execution of constructed code at run time. 'Test' refers to the debugging and proving of facilities for the purpose of commissioning, approval, development and so on.

Modularity is concerned with the partitioning of software into smaller more manageable components. Modularity in the construction process comes about naturally by providing facilities which allow computer loads to be built up from many component parts; however care must be taken to ensure that the maximum degree of flexibility is retained. Modularity at run time is necessary so that processor time can be shared between competing parallel processes; it is only partially controllable since real time interrupts break into running programs in an unpredictable fashion. As for software testing, modularity is achieved by identifying and separating the individual facilities which are combined to make up the total system specification; to these must be added various facilities which are not available to system users but which provide support for higher levels of software.

In the initial stages the MASCOT work was designated as an 'approach' since it was intended to develop a style of programming rather than to produce tight specifications for standard facilities. However with the passage of time a considerable amount and variety of experimental work has been associated with MASCOT and a set of well defined standard facilities has evolved. We are now confident that these facilities represent a useful and balanced set and are applicable to a reasonably wide range of practical requirements.

If applied to its full extent MASCOT affects nearly every aspect of software design and implementation. However not all MASCOT features need be incorporated and there is scope for taking a subset which can be blended in with existing facilities. At the very least a MASCOT-like system will include the executive software for process scheduling. This takes the form of a small kernel providing certain basic operating system facilities for handling interactions and sharing processor time between parallel processes. Other more complex facilities may then be constructed using standard MASCOT system building techniques.

In this paper we concentrate on the process synchronisation aspects of MASCOT. Shrivastava (1975) has reviewed some important ideas relating to process synchronisation and many papers including, more recently, those by Kammerer (1977) and Ford and Hamacher (1977) give a good summary of the more significant contributions to work in this area. There are two main lines of approach: first, the use of P, V operations on semaphores (Dijkstra, 1968) introduced in order to provide mutual exclusion and second, the use of monitors for mutual exclusion supplemented by wait, signal operations on a condition variable to provide an explicit cross stimulation mechanism (Hoare, 1974). MASCOT has much in common with these techniques but there are significant differences, the most important being that both mutual exclusion and cross stimulation are treated as a coordinated set of facilities operating on a single type of control variable called a control queue. Also, in designing MASCOT we have attached great importance to practical considerations. All process scheduling operations are made as efficient as possible and the facilities provided are flexible yet safe and can be easily adopted for use in conjunction with design methods, languages and most hardware associated with current projects.

Sections 2 and 3 of the paper provide some background to MASCOT process synchronisation. These sections introduce the relevant terminology, and describe the structure of MASCOT software and the particularly simple form of kernel operating system which is required to schedule processes at run time. Section 4 describes the basic facilities for process synchronisation in terms of a control variable and the set of primitive procedures which can operate on it. Section 5 shows how this form of control variable and its associated primitive procedures can be used to construct the message passing system which is an integral part of the MASCOT approach. Section 6 extends the discussion to interrupt handling and shows how this aspect can be incorporated in the overall design. Section 7 gives some examples illustrating various techniques and solutions to standard problems. Section 8 compares MASCOT synchronisation with techniques using semaphores and monitors. Section 9 discusses some general points and gives the present state of MASCOT research and Section 10 summarises the main features and advantages of the MASCOT approach. Finally, for completeness two appendices are included outlining the MASCOT construction

*In the interval between submission and publication of this paper MASCOT has been further developed and refined resulting in some changes to the facilities and techniques described herein. Further details can be obtained from the authors.

†Department of Computing and Control, Imperial College, 180 Queens' Gate, London SW7 2BZ
‡Royal Signals and Radar Establishment, St. Andrews Road, Gt. Malvern, Worcs. WR14 3PS

and test facilities.

MASCOT is in fact language independent but in this paper we have found it convenient to use the conventions of ALGOL 68-R (Woodward and Bond, 1974) to express program text and data structures. Three additional types of procedure, the 'root procedure', 'access procedure' and 'response procedure' are used to denote particular MASCOT functions. Example programs have been chosen to illustrate points of technique and are not necessarily optimised for run time efficiency. There are a few minor changes to MASCOT as previously described (Jackson and Simpson, 1975) which have been brought about by recent development work.

2. Software structure

MASCOT software consists of 'subsystems' which run under the control of a 'kernel'. The subsystem is an important MASCOT concept. It provides a means of expressing software structure and organisation which is sufficiently general to be used in the formulation of the major part of a computer load.

The kernel software provides executive programs which exercise overall control over the allocation of processing time, both in response to external hardware interrupts and at the base (i.e. non-interrupt) level. It also contains certain key files, tables and lists (called the kernel data base) which are used both in the construction of the software and in the dynamic management of the execution sequence. A third important component of the kernel software consists of a set of 'primitive' procedures. These procedures allow subsystems to interact both with the kernel data base and with certain application dependent control variables lying outside the kernel software.

A MASCOT subsystem consists of one or more 'activities' which are connected by 'intercommunication data areas'. Each activity is essentially a 'process' in the conventional sense and can be regarded as a separate thread in a multiprogramming system. The program which determines the function of an activity is called a 'root procedure'. The formal parameters of a root procedure specify the intercommunication data areas and their types which will be required when it is used to support an activity.

Intercommunication data areas fall into two broad categories, 'channels' and 'pools'. Channels are used exclusively for passing message data between activities, i.e. from producer activities on the one side to consumer activities on the other. Pools are generally used as depositories for non-transient data. Conceptually a channel has two uni-directional interfaces and is represented by the symbol I whereas a pool has one bi-directional interface and is represented by the symbol ⎵. Both of these basic types can be further qualified by use of the ALGOL 68 mode and structure facility. Access to pools and channels is generally by means of 'access procedures'.

Root procedures, channels and pools are known collectively as 'system elements' and are built up individually by the construction facilities (see Appendix 1). They are the component elements which are combined together into subsystems by use of a 'FORM' facility. These concepts are best illustrated by an example. **Fig. 1** shows a MASCOT subsystem called triplicated expansion. The subsystem has three activities and uses two root procedures which would have the following procedure headers:

```
ROOT PROC expand = (REF CHARCHAN indata, outdata,
                    REF DICTPOOL lookuptable)
ROOT PROC duplicate = (REF CHARCHAN indata,
                       outdata1, outdata2)
```

The function of expand is to read text as a sequence of characters from a channel of type CHARCHAN, recognise macro definitions and store them in a look up table within a pool of type DICTPOOL, recognise calls of remembered macros, and output the expanded source text as a sequence of characters into another CHARCHAN channel. The function of duplicate is to take a stream of characters from one channel of type CHARCHAN and copy it into two others.

The data areas required by this subsystem are six channels of type CHARCHAN (in, trans1, trans2, out1, out2, out3) and one pool of type DICTPOOL (dictionary). These system elements are actual areas in main store and are conceptually loaded by the construction facilities. Some data areas are internal to a subsystem (trans1, trans2, dictionary); others provide the external interface and are usually shown on the dotted line defining the boundary of the subsystem.

The subsystem is created by a command such as:

```
FORM triplicated expansion =
  (expand (in, trans1, dictionary),
  duplicate (trans1, out1, trans2),
  duplicate (trans2, out2, out3));
```

Such commands will normally be actioned by a command interpreter which carries out checks on parameter types. Once created the subsystem does not run until a further command is issued:

START (triplicated expansion, *p*);

The second parameter in the command specifies the priority to be attached to the start up of this subsystem.

The root procedure duplicate is used by two activities and it follows that MASCOT software is normally re-entrant. In general activities are anonymous although a subsystem which contains a single activity is in effect a named activity.

The FORM command constitutes an additional stage of linking over and above that required to build up individual root procedures, pools and channels. It gives a degree of flexibility which is of considerable advantage during commissioning and modification. It also provides a means of expressing overall software structure clearly and unambiguously.

3. Scheduling

Processing time is allocated by the kernel. **Fig. 2** shows those parts of the kernel which are most directly concerned with the scheduling of activities. The despatcher 'calls' activities to deal with the outstanding work as listed in the current lists. An activity returns control to the scheduler when it has no further useful work to do or has consumed a reasonable share of processor time; this is a cooperative form of scheduling. Alternatively the kernel may force a return of control on a pre-emptive basis.

Either of the following statements can be used to place an activity on the kernel lists:

SUSPEND	The activity is placed on one of the current lists.
DELAY (*n*)	The activity is placed on a delay list and will be transferred to a current list after *n* time units have elapsed.

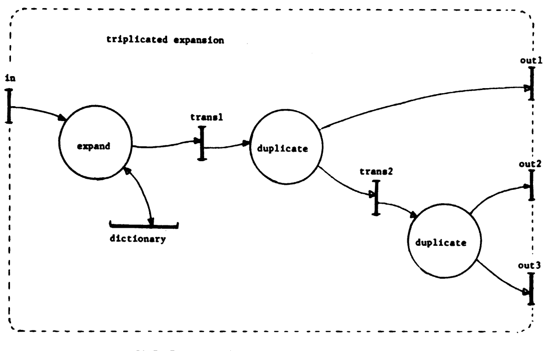

Fig. 1 MASCOT subsystem

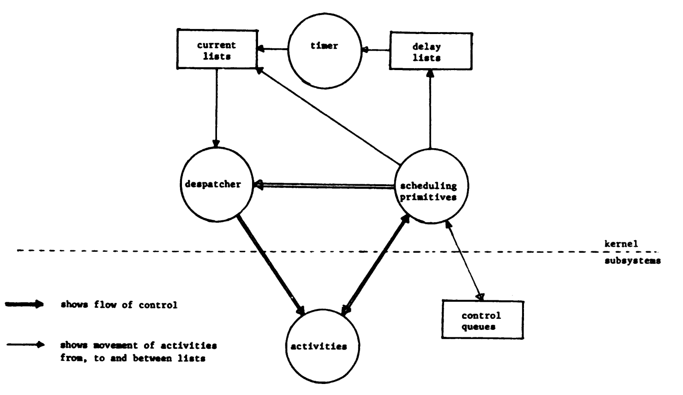

Fig. 2 Scheduling of activities

Statements of this type, which are scheduling primitives, are used by activities to return control unconditionally to the kernel. The despatcher then selects the next activity to be run from one of the current lists and initiates further processing.

In addition to SUSPEND and DELAY there are other primitives which interact with 'control queues' declared within the subsystem software. These control queues contain 'pending lists' which are used to list activities waiting to be restarted by a software stimulus. When an activity is restarted in this way it is transferred from the pending list to one of the current lists; this mechanism is fully described in Section 4. The primitives which interact with control queues may or may not cause control to be returned to the despatcher; this depends on the state of the control queue and the particular primitive used. It should be noted that control queues always appear as elements in a pool or channel data structure and that the primitives which interact with them are normally embedded in access procedures.

The kernel contains several current lists each with its own priority. An activity normally retains the priority allocated at START time except when it is running under the control of a control queue when it assumes the priority of that queue (Section 4). This straightforward treatment of priority is designed to achieve very low scheduling overheads by eliminating searching and reordering of list items when executing scheduling primitives. When items are selected from the current lists the despatcher can use a special technique based on dynamic adjustment of list priority according to a 'scan policy' (chosen to suit the particular application). The effect of this is to restrict the proportion of processor time that can be allocated to activities held on high priority lists. This ensures that all outstanding work is turned over even under heavy overload conditions.

The period of execution of an activity started by the despatcher and terminated by a scheduling primitive is known as a 'slice'. The MASCOT monitor (Appendix 2) includes a facility for measuring and recording slice times and this information can be used to position scheduling primitives so that satisfactory cooperative scheduling is achieved. Also, the despatcher contains a mechanism for detecting excessively long slices and can forcibly terminate the offending activity.

The hardware configuration used to run MASCOT affects the design of the kernel and also influences the detailed programming of the subsystem software. The simplest configuration consists of a single processor with no additional facilities, and the despatcher, together with its associated lists and primitives, provides the executive software which is necessary for parallel processing in such a system. Three additional features are relevant:

More than one processor (with shared storage)
Interrupts
Unilateral slicing.

A multiprocessor configuration is readily catered for by feeding a new slice from a current list to each processor as it becomes free. Interrupts are handled by an interrupt handler (see Section 6). The term 'unilateral slicing' describes a facility which reduces the need for cooperative scheduling; such a mechanism would forcibly suspend activities according to some time rationing system and return them to the current lists in order to achieve a more equitable share out of processor time. In all cases certain straightforward safeguards (e.g. ensuring the indivisibility of critical sections of program in primitive procedures) must be used to avoid the possible corruption of control data; if this is done then MASCOT provides all the facilities necessary for the synchronisation of parallel processing.

The simplicity of the MASCOT scheduling mechanism leads to very efficient operation in practice. Activities created by the FORM facility remain inert until the START command transfers them onto a current list. Following this they are normally either running or held on one of the lists previously described, and only exceptional measures can interfere with this process (see Appendix 2). It should be noted that a common area is used for all kernel and control queue lists. This means that the individual allocations of space for each list can be adjusted dynamically without affecting the overall space required so that, following the FORM command, no activity can ever be held up for lack of list space.

4. Control queues

Synchronisation embraces the interrelated functions of mutual exclusion, cross stimulation and resource allocation. Mutual exclusion is required to prevent activities running in parallel from interfering with each other's operations on variables which are common to both. A cross stimulation mechanism is required to allow activities to be stopped temporarily and restarted. Resource allocation is concerned with controlling the use of limited physical assets (peripheral devices, memory, etc.) and usually involves a rather complex form of mutual exclusion. MASCOT caters for mutual exclusion and cross stimulation in a direct and simple manner. These basic facilities can be used to build the more complex scheduling algorithms required for resource allocation.

The 'control queue' is the fundamental MASCOT control variable and is a simple data structure having a number of elements. It is necessary to describe three of these in order to explain basic operations on the control queue:

```
MODE CONTROLQ = STRUCT (INT state,
                        LIST pendinglist,
                        INT priority,
                        ........);
```

The integer variable priority is used to denote the priority of the current list to which an activity is transferred when it is removed from the control queue's pending list. The value of priority would normally be preset at load time. The list variable pending list is used to queue up activities which are held on the control queue. Finally we have the integer variable state which can take on five basic values denoting whether the control queue is in use, whether the activity at the head of the list is waiting for an explicit stimulus, and whether the control queue has already been primed by a stimulus. (Note: two further state values may be introduced to support the response facility used in interrupt handling; this is explained in Section 6.)

Before going on to describe the basic primitive procedures which can operate on a control queue we must clearly define the five states of the queue:

State 1: The control queue is not in use and any activity joining the queue will be allowed to proceed immediately.

State 2: The control queue is in use and any activity joining the queue will be held up.

State 3: The control queue is in use but the activity at the head of the queue is held up awaiting a stimulus to restart it.

State 4: As for state 2, except that the queue has already been primed by a stimulus.

State 5: As for state 1, except that the queue has already been primed by a stimulus.

States 4 and 5 allow a stimulus to be remembered and so remove the constraint that a stimulus to restart an activity must be generated after that activity has waited on the control queue.

Four basic primitive procedures are used to interact with control queues:

JOIN (controlq): If the control queue is not in use (states 1 or 5) it is 'secured' and the activity proceeds. Otherwise (states 2, 3, 4) the activity is placed at the back of the pending list to await its turn on a first-in-first-out basis.

LEAVE (controlq): The control queue is 'released' for use by the next (if any) activity on the pending list; this activity will automatically be transferred to a current list with priority as specified by the priority field of the control queue.

WAIT (controlq): This statement can only be used between JOIN-LEAVE brackets and allows an activity to retain its hold on a control queue whilst waiting to be restarted by a stimulus from elsewhere. If the control queue is already primed (state 4) the activity proceeds immediately. Otherwise (state 2) the activity is placed at the front of the pending list.

STIM (controlq): If an activity is waiting on the control queue (state 3) then that activity is restarted by transferring it to a current list with priority as specified by the priority field of the control queue. Otherwise (states 1 or 2) the stimulus primes the control queue by adjusting the state to 5 or 4. A control queue which has already been primed (states 4 or 5) ignores further stimuli.

These MASCOT primitives are easily programmed and are efficient in use. Clearly operations on control queues must be indivisible. **Fig. 3** is a control queue state change matrix summarising the effect of the basic primitive operations. In addition to the illegal operations indicated checks are also carried out to ensure that WAIT and LEAVE are only used by the activity which has been able to secure the queue by means of a JOIN.

The JOIN-LEAVE operations on a control queue give an identical facility to P, V operations on a binary semaphore. Likewise the WAIT-STIM combination is similar to wait, signal operations on a condition variable. In the next section we show how the coordination of JOIN-LEAVE-WAIT-STIM into a unified set of operations on a single type of control variable produces a natural and logical solution to the problem of intercommunication between loosely coupled processes.

5. Activity intercommunication using channels

MASCOT activities communicate with one another by means of channels and pools. Where an activity is the sole user of an interface to a channel or pool it is permissible for that activity to secure the interface for its own exclusive use (see Sections 6 and 7). However it is more common for an interface to be shared between several activities and in this case it is a MASCOT convention that all interaction with the data area must be by means of access procedures.

The pools, channels and associated access procedures can take many forms and are defined by the user to suit his own particular application. However all access procedures follow certain design conventions which cope with the synchronisation aspects. These procedures contain calls of the JOIN-LEAVE-WAIT-STIM primitives which operate on control queues within the pools and channels. By standardising on a restricted number of access procedures operating on a limited set of pool and channel types it is possible for the primitive operations to be hidden completely from the user. This can be a great advantage in a large project employing many programmers.

One of the most important and interesting applications of this technique concerns its use in setting up a message passing system. The principles of this application are best explained by considering two examples. Once these are understood the approach is readily extended to other related problems.

Consider the simplest possible form of channel designed to transmit one word of data at a time. The channel data structure might take the following form:

```
MODE ONEWORDCHAN =
                        STRUCT (CONTROLQ inq, outq,
                                BOOL empty,
                                INT data);
```

Fig. 4 shows the associated put and get access procedures for writing and reading. The elements in a channel data structure invariably fall into three categories.

Control variables

These are control queues for mutual exclusion and cross

Initial state	*State following operation*			
	JOIN	*WAIT*	*LEAVE*	*STIM*
1	2	*	*	5
2	■	3	1 or 2	4
3	■	*	*	2
4	■	2	5 or 4	●
5	4	*	*	●

* illegal operation
● no effect on state
■ no effect on state, but activity joins pending list

Fig. 3 Control queue state change matrix for basic primitives

```
ACCESS PROC put = (INT x, REF ONEWORDCHAN
   chan):
BEGIN
 JOIN(inq OF chan);
 WHILE NOT(empty OF chan)
 DO WAIT(inq OF chan);
 data OF chan := x;
 empty OF chan := FALSE;
 STIM(outq OF chan);
 LEAVE(inq OF chan)
END;

ACCESS PROC get = (REF INT x, REF
   ONEWORDCHAN chan):
BEGIN
 JOIN(outq OF chan);
 WHILE(empty OF chan)
 DO WAIT(outq OF chan);
 x := data OF chan;
 empty OF chan := TRUE;
 STIM(inq OF chan);
 LEAVE(outq OF chan)
END;
```

Fig. 4 Writing and reading procedures for simple channel

stimulation. There are always at least two control queues in a channel, one for controlling input access and one for controlling output access.

State variables
These indicate the state of the data areas used for passing the information. In our example above we have only one state variable but more complex channels will have input and output pointers, etc.

Data area
This is the area set aside for temporary storage of data on its way through the channel.

There is complete symmetry between input and output access procedures. These are always programmed to carry out the following sequence of actions:

1. JOIN the control queue controlling the relevant (own) interface
2. Check the state of the data area and WAIT if necessary
3. Transfer data
4. Amend state
5. STIM the control queue controlling the other interface
6. LEAVE the control queue controlling own interface.

The logic behind this sequence is quite straightforward. If several activities are simultaneously attempting to gain access to an interface by means of the JOIN primitive then all but one are held on the control queue's pending list. The activity which has access then inspects the state of the data area; if it cannot proceed it WAITs on the control queue and will be restarted in due course by a STIM from the other side of the channel. The data is then placed in or taken out of the channel and the state is amended accordingly. Since the action is likely to be of interest to an activity WAITing on the other side of the channel the other interface control queue is STIMmed. Finally the activity uses the LEAVE primitive to release the interface and the next activity (if any) held on the control queue is automatically restarted.

A second more useful illustration of the technique is that of intercommunication using a 'bounded buffer'. Such a buffer for passing characters might take the following form:

```
MODE CHARCHAN = STRUCT (CONTROLQ inq, outq,
                INT inpoint, outpoint, max, size,
                 mask,
                [0 : max] CHAR data);
```

where $\max = 2^n - 1$, $\text{size} = 2^n$, $\text{mask} = 2^{n+1} - 1 \quad n \geqslant 0$

The way in which the structure elements are used can be seen by studying the write and read access procedures in **Fig. 5.** These procedures transfer one character at a time. Where an activity needs to transfer a group of several characters, rather than use repeated call of read and write, it is more efficient to provide additional standard procedures which secure the interface until the message is complete.

The buffer pointers in CHARCHAN are always incremented modulo 2^{n+1}; this means that the full and empty states can be deduced from the pointer values alone so avoiding the use of any state variable which can be amended from both sides of the channel. Thus there is no need for additional control queues to protect the state variables. This technique could also be applied to the ONEWORDCHAN described previously.

The use of control queues as described above ensures that mutual exclusion is only applied where it is strictly necessary and it is quite possible for the buffer to be accessed from both sides simultaneously. On the other hand a large number of redundant STIMSs are generated. This is quite harmless since STIM is a short procedure taking little time to execute. Once a control queue has been primed with a STIM this condition is only cleared by a WAIT operation on the same control queue; in some circumstances this may cause the 'WHILE condition DO WAIT(q)' statement to be executed twice in quick succession, but this is a small price to pay for the extreme simplicity of the technique.

```
ACCESS PROC write = (CHAR x, REF CHARCHAN
   chan):
BEGIN
 JOIN(inq OF chan);
 WHILE full (chan)
 DO WAIT(inq OF chan);
 (data OF chan) [inpointer(chan)] := x;
 stepinpointer(chan);
 STIM(outq OF chan);
 LEAVE(inq OF chan)
END;

ACCESS PROC read = (REF CHAR x, REF
   CHARCHAN chan):
BEGIN
 JOIN(outq OF chan);
 WHILE empty(chan)
 DO WAIT(outq OF chan);
 x := (data OF chan) [outpointer(chan)];
 stepoutpointer(chan);
 STIM(inq OF chan);
 LEAVE(outq OF chan)
END;
where
PROC full (REF CHARCHAN chan) BOOL:
((inpoint OF chan-outpoint OF chan) MASK mask OF
   chan = size OF chan);
PROC empty = (REF CHARCHAN chan) BOOL:
(inpoint OF chan = outpoint OF chan);
PROC inpointer = (REF CHARCHAN chan) INT:
(inpoint OF chan MASK max OF chan);
PROC outpointer = (REF CHARCHAN chan) INT:
(outpoint OF chan MASK max OF chan);
PROC stepinpointer = (REF CHARCHAN chan):
(inpoint OF chan := (inpoint OF chan + 1) MASK mask
   OF chan);
PROC stepoutpointer = (REF CHARCHAN chan):
(outpoint OF chan := (outpoint of chan + 1) MASK mask
   OF chan);
OP MASK = (INT i, j) INT:
ABS(BIN i AND BIN j);
```

Fig. 5 Writing and reading procedures for bounded buffer

6. Interrupt processing

It is envisaged that the bulk of a computer's workload will be carried out at the base level and as such will be controlled by the despatcher. However where hardware interrupts are present there is a need to respond to stimuli whose time of occurrence is completely uncorrelated with the scheduling of activities by the despatcher. This processing cannot be controlled by the despatcher and an 'interrupt handler' is required.

The interrupt handler is that part of the kernel software which provides the immediate response to hardware interrupts. It must interface with MASCOT subsystems lying outside the kernel and in practice the proportion of the total interrupt handling function that is retained within the kernel is very much dependent on the particular application. At one extreme it is possible for the interrupt handler to cover all aspects of interrupt processing and to interface directly with standard

MASCOT channels. On the other hand MASCOT subsystems can be included to take on a large share of this work thus greatly reducing the size and complexity of the interrupt handling software within the kernel. This section illustrates the latter approach and introduces the necessary additional features required for this style of interrupt programming.

The basic synchronisation facilities described in the previous sections provide a technique for the indirect call of a WAITing activity. The time at which a restarted activity will run is uncertain and there are small but significant overheads associated with the transfer of an activity from list to list. This makes the basic facilities unsuitable for initiating the response to a hardware interrupt and a more direct mechanism is required. Accordingly we define a new type of parameterless procedure called a 'response procedure' which can be associated with a control queue in a manner such that it is called immediately without despatcher intervention whenever that queue is STIMmed.

One way of providing the response procedure facility involves the introduction of two additional control queue state values, three new primitives and a minor extension to the STIM primitive. The new control queue state values are defined as follows:

State 6: A response procedure is associated with the control queue and is waiting to run.

State 7: A response procedure is associated with the control queue and is currently being executed.

The response primitive procedures are defined as follows:

SET RESPONSE (q, response proc): This primitive associates a response procedure with a control queue. The control queue must previously have been JOINed (i.e. be in state 2 or 4). If the control queue has state 2 then the state is adjusted to 6 indicating that a response procedure is associated with the queue. If the control queue has state 4 (i.e. has already been STIMmed) then the state is adjusted to 7 and the response procedure is executed immediately; on completion of the response procedure the queue state is set to 6.

CLEAR RESPONSE (q): This primitive can only be used within a response procedure (i.e. when the control queue is in state 7). It has the effect of cancelling the response and places the activity in which it is embedded on a current list so that this activity will be re-entered from the despatcher at a point following the SET RESPONSE statement. Since in this case the stimulus has not been fully actioned the control queue must be left in state 4. CLEAR RESPONSE causes immediate exit from the response procedure.

CHANGE RESPONSE (q1, q2, response proc): This primitive can only be used within a response procedure (i.e. when q1 is in state 7). It combines the effect of a CLEAR RESPONSE (q1) immediately followed by SET RESPONSE (q2, response proc). It cancels the response on one queue and sets up a new response on a second queue without involving the despatcher. CHANGE RESPONSE causes exit from the response procedure when the associated clearing and setting actions are complete.

Minor extensions to the STIM primitives are required to deal with control queue states 6 and 7. The action of STIM for state 6 is to set the queue to state 7, execute the response procedure and on completion reset the queue to state 6. There is no STIM action for state 7 and the STIM is ignored; state 7 prevents multiple initiation of a response procedure which would cause faulty operation of the response software.

Implementation of the response facilities is quite straightforward. **Fig. 6** is the relevant control queue state change matrix. Special arrangements must be made so that response procedures always return control to the calling primitive (STIM, SET RESPONSE or CHANGE RESPONSE). Also, when the response is set up, the re-entry point following SET RESPONSE must be remembered for use by the response procedure exit mechanism when executing CLEAR RESPONSE. Clearly the only primitives that are allowed within a response procedure are STIM, CLEAR RESPONSE and CHANGE RESPONSE.

Initial state	*State following operation*		
	SET RESPONSE	*CLEAR RESPONSE*	*STIM*
1	*	*	5
2	6	*	4
3	*	*	2
4	7→6	*	●
5	*	*	●
6	*	*	7→6
7	*	4	●

* illegal operation
● no effect on state
7→6 denotes that state is set to 7 whilst the response procedure is running and is set to 6 on completion

Fig. 6 Control queue state change matrix for response primitives

The response facility as outlined above can be used to set up an immediate reaction to STIM from any source. When dealing with interrupts it is merely necessary for each interrupt (index i), in addition to the usual register preservation, etc. to cause the execution of a STIM ((q OF icp) [i]) statement (or equivalent) where icp is an 'interrupt control pool' containing an array q of queues, one queue for each interrupt source. The activities (one per interrupt source) which contain and control the response to interrupts interface with the interrupt control pool and are known as interrupt response activities. It should be noted that those response procedures which contain 'privileged' data transfer instructions will need to be loaded within the appropriate protection regime of the computer.

To illustrate some possible techniques, consider a computer having three hardware interrupts associated with the following functions:

1. A simple clock interrupt with no data transfer
2. An input peripheral for passing single characters. It is assumed that the data is transferred through a hardware register (designated hwreg[2]).
3. An output peripheral for passing single characters. It is assumed that the data is transferred through a hardware register (designated hwreg[3]).

Fig. 7 shows a subsystem containing a set of interrupt response activities to deal with these interrupts. Possible forms of the root procedures clockint, incharint, outcharint are given in **Fig. 8.** incharint is programmed using the CLEAR RESPONSE facility to escape from the response procedure whereas outcharint uses CHANGE RESPONSE to switch between two response procedures. It is interesting to compare the two different approaches adopted by these two root procedures to the problem of deciding when to restart the response to interrupts. outcharint produces a direct response with low overhead to every STIM through the channel but lacks the ability to dispense with redundant STIMs—this latter feature occurs naturally in incharint under conditions where the consumer may issue many STIMs before incharint is rescheduled (i.e. in some circumstances this algorithm may be more efficient). A further practical point in these programs is the

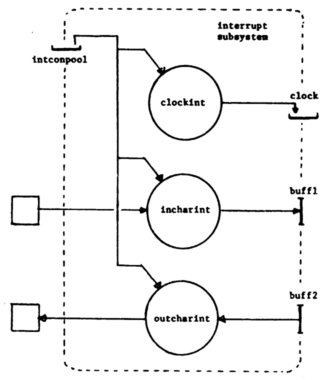

Fig. 7 Interrupt subsystem

presence of the INT i parameter in incharint and outcharint to illustrate a method whereby a single root procedure can be used to support many identical interrupt response activities.

The technique outlined above allows easy and effective control of interrupt handling. The access mechanisms for channels and pools used in conjunction with interrupts must be programmed with particular care since input and output operations on a data area can be interlaced. The form of channel described in Section 5 is quite safe and particularly efficient in use. Consider the operation of incharint (Fig. 8). This is programmed to run for as long as possible until the buffer is full. If data is being removed from the buffer at a sufficiently high rate the input device will run continuously. If the output rate is slower than that of the input device then the input batch size is greater than the capacity of the buffer; for example if data is removed from the buffer at a rate which is $\frac{3}{4}$ the rate of the input peripheral then the batch size is 4 times the size of the buffer. This is an important advantage of the MASCOT approach to channel access synchronisation. However, having said this, it should be noted that the implementation of the clock interrupt in Fig. 8, although efficient as regards time overheads, carries a constant space overhead in the form of the response activity which, once started, is never called again from the despatcher. System designers must weigh the convenience of using this standard interrupt handling technique against the space required to support it.

The program text in Fig. 8 illustrates a practical point which gives efficient operation of MASCOT software. The interrupt response activity secures the input interface of its output channel for its own exclusive use. Thus the JOIN statement is only executed once and there is no need for a matching LEAVE statement (the activity can be stopped by means of the TERMINATE command described in Appendix 2; and of course care must be taken to force a CLEAR RESPONSE where necessary when terminating an activity in this manner).

The MASCOT approach to interrupt handling is unusual in that it allows a considerable degree of direct control over interrupt programming. This may be undesirable in some applications and in such cases interrupt response becomes a system software function and an interface with user programs should be provided by means of standard channels or pools. However where unusual or complex peripheral devices are involved the flexibility of the MASCOT approach can give considerable advantages when developing the interrupt response software.

7. Examples

Two examples are considered. The first deals with the sorting of messages so that they can be handled in an order other than first-come-first-served, and the second is concerned with coordinating reading and writing access to a common data base. These are typical real time problems and it is shown that MASCOT synchronisation facilities provide straightforward solutions.

7.1 *Example* 1. *Priority message sorting*

Consider a subsystem for handling messages in order of priority. It is assumed that such a subsystem (called priority sort) is to have three input channels and one output channel. It is to be designed so that messages appearing in the first input channel are given precedence over those in the second and third input channels; similarly the second channel is given precedence over the third. To further define the problem we specify that the subsystem should be capable of insertion as an optional element between producer and consumer

```
ROOT PROC clockint = (REF ICPOOL icp, REF
   CLOCKPOOL t):
BEGIN
 INT clockintnum = 1;
 RESPONSE PROC tick = VOID; time OF t := time OF
   t + 1;
 JOIN((q OF icp) [clockintnum]);
 SET RESPONSE((q OF icp) [clockintnum], tick)
END;

ROOT PROC incharint = (REF CHARCHAN oc, REF
   ICPOOL icp, INT i):
BEGIN
 RESPONSE PROC transdata = VOID:
 BEGIN IF full(oc) THEN
          CLEAR RESPONSE((q OF icp) [i]) FI;
          (data OF oc) [inpointer(oc)] := hwreg[i];
          stepinpointer(oc);
          STIM(outq OF oc)
 END;
 JOIN((q OF icp) [i]);
 JOIN(inq OF oc);
 DO
 BEGIN WHILE NOT empty(oc) DO WAIT(inq OF oc);
          SET RESPONSE((q OF icp) [i], transdata)
 END
END;

ROOT PROC outcharint = (REF CHARCHAN ic,
   REF ICPOOL icp, INT i):
BEGIN
 RESPONSE PROC transdata = VOID:
 BEGIN IF empty(ic) THEN
          CHANGE RESPONSE((q OF icp) [i], outq OF ic,
             waiting) FI;
          hwreg[i] := (data OF ic) [outpointer(ic)];
          stepoutpointer(ic);
          STIM(inq OF ic)
 END;
 RESPONSE PROC waiting = VOID:
 BEGIN IF full(ic OR lastchar(ic) THEN
          CHANGE RESPONSE(outq OF ic, (q OF icp) [i],
             transdata)FI
 END;
 JOIN((q OF icp) [i]);
 JOIN(outq OF ic);
 SET RESPONSE(outq OF ic, waiting)
END;
```

Note: lastchar delivers value TRUE if the last character inserted in the channel is an end of stream marker.

Fig. 8 Root procedures for interrupt handling

subsystems on either side and can be bypassed if the priority sorting facility is not required. This means that priority sort must not in any way alter the form of the mutually compatible producer output and consumer input interfaces. Priority sort merely arranges the ordering of messages in course of transfer between standard message channels.

This problem is best tackled by developing a technique such that a message appearing in any one of several input channels is able to restart a single activity which can take account of channel precedence whilst transferring the message data. There are several ways of doing this; here we use a method involving a REF CONTROLQ element in the channel data structure to give greater control over the destination of STIMs. Consider the following type of channel for transmitting messages:

```
MODE MESSCHAN = STRUCT (CONTROLQ inq, outq,
                        REF CONTROLQ
                         refoutq,
                        INT inpoint, outpoint,
                        max, size, mask,
                        [0 : max] MESS data);
```

where max = $2^n - 1$, size = 2^n, mask = $2^{n+1} - 1$, $n \geqslant 0$

In normal use refoutq would be preset to point to outq but is also available to route STIMs through to a control queue outside the MESSCHAN data structure.

Two access procedures are required (say readmess and writemess) for reading from and writing to MESSCHAN data structures and these are similar to those shown in Fig. 5. To achieve the desired effect it is necesary to make the following change:

STIM(outq OF chan) in write becomes
STIM(refoutq OF chan) in writemess

The dereferencing involved in executing this statement causes a small additional overhead.

Several MESSCHAN structures can now be combined so that an input to any one of them applies a STIM to a common output control queue. We define a multiple message channel as follows:

```
MODE MULTMESSCHAN = STRUCT(CONTROLQ
                             commonoutq,
                           INT size,
                           [1 : size] MESS-
                            CHAN chan);
```

where refoutq of each MESSCHAN is preset to point to commonoutq.

Using the above channels and associated access procedures the subsystem required for this example is extremely simple and is shown in **Fig. 9**, with the associated root procedure transmit shown in **Fig. 10.** The size of inmess is set to three to give three input message channels (chan[1] OF inmess, chan[2] OF inmess, chan[3] OF inmess). The size of the input channels depends on the desired maximum input queue lengths but the output queue should be kept reasonably short (say 2 or 4 messages); this prevents a flood of low priority input messages from blocking those of higher priority for any length of time but at the same time allows more than one message to be held forward so reducing the chances of holding up the consumer subsystem under heavy loading conditions.

The root procedure transmit uses standard access procedures to read and write messages. This type of situation is one where it might be advantageous to allow transmit permanently to JOIN the output queues of the input channels and the input queue of the output channel and adopt an alternative style of access mechanism which dispenses with calls of JOIN and LEAVE as each message passes through the subsystem (cf Fig. 8). This technique will be more efficient at run time.

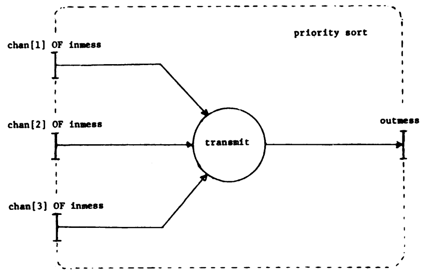

Fig. 9 Message sorting subsystem

```
ROOT PROC transmit = (REF MULTMESSCHAN ic,
   REF MESSCHAN oc):
BEGIN
 REF CONTROLQ sync = commonoutq OF ic;
 MESS x;
 INT m = size OF ic;
 JOIN(sync);
 DO BEGIN
 scan: FOR n TO m
   DO IF NOT empty ((chan OF ic) [n])
     THEN readmess(x, (chan OF ic) [n]);
             writemess(x, oc);
             GOTO scan
   FI;
  WAIT(sync)
 END
END;
```

Fig. 10 Root procedure 'transmit'

7.2. *Example* 2. *Readers and writers*

As our second example we consider the problem of 'readers and writers' and program it in the style of monitor local procedures proposed by Hoare (1974). The problem concerns a record (regarded as a POOL) which is kept up to date by a number of 'writer' activities and is accessed by a number of 'reader' activities. The design constraints are as follows:

1. Any number of reader activities may simultaneously access the record.
2. Writer activities must have exclusive access.
3. A new reader should not be started if a writer is waiting.
4. To avoid the danger of indefinite exclusion of readers, those readers waiting at the end of a write should have priority over writers which arrive after them.

Four procedures are required:

(*a*) startread	entered by reader who wishes to read.
(*b*) endread	entered by reader who has finished reading.
(*c*) startwrite	entered by writer who wishes to write.
(*d*) endwrite	entered by writer who has finished writing.

The record to be accessed could take the following form:

```
MODE RECORDPOOL = STRUCT(CONTROLQ iq, rq,
                         INT rcount,
                         - - - - - - -);
```

where iq	Ensures exclusive access by writing activities and where necessary queues readers and writers in order of arrival.
rq	Protects operations on rcount.
rcount	Counts number of readers with current access.

The startread, endread, startwrite, endwrite procedures are given in **Fig. 11** and are self explanatory.

This solution would be somewhat safer in use if each pair of start and end procedures were combined into a single access

```
PROC startread = (REF RECORDPOOL pool):
BEGIN
 JOIN(iq OF pool);
 JOIN(rq OF pool);
 (rcount OF pool) PLUS 1;
 LEAVE(rq OF pool);
 LEAVE(iq OF pool)
END;

PROC endread = (REF RECORDPOOL pool):
BEGIN
 JOIN(rq OF pool);
 IF (rcount OF pool) MINUS 1 = 0
 THEN STIM(iq OF pool)
 FI;
 LEAVE(rq OF pool)
END;

PROC startwrite = (REF RECORDPOOL pool):
BEGIN
 JOIN(iq OF pool);
 WHILE(rcount OF pool) > 0
 DO WAIT(iq OF pool)
END;

PROC endwrite = (REF RECORDPOOL pool):
BEGIN
 LEAVE(iq OF pool)
END;
```

Fig. 11 Control procedures for readers and writers problem

procedure. Such access procedures would have to allow users to specify their particular form of data transfer as a PROC parameter, but would avoid the danger of unmatched start or end statements.

8. Semaphores and monitors

In this section we will compare the MASCOT approach to synchronisation with techniques using conventional semaphores and monitors. Each approach to synchronisation has its own particular advantages and to make a comparison it is necessary to define a common problem. The problem considered here is that of the bounded buffer (channel) which can be accessed simultaneously by many producer and consumer processes, access conflicts being resolved on a first-in-first-out basis. The MASCOT solution to this problem is described in Section 5 and Fig. 5. As a further aid to the establishment of a common framework for discussion the action of semaphores and monitors is described in terms of MASCOT-like operations.

8.1 *Semaphores*

Fig. 12 shows the construction of a semaphore control variable and associated P, V operations using MASCOT control queues and primitives. The operations on 'i OF s' must be indivisible to guarantee the integrity of the counting variable.

To obtain the required control over access to a buffer of type CHARCHAN (Section 5) it is necessary to replace the two control queues by four semaphores: inseml, insem2, outseml, outsem2. The access procedures then take the form shown in Fig. 12.

inseml is initialised to the size of the buffer and ensures that input processes are blocked when there is no space in the buffer. outseml is initialised to zero and ensures that output processes are blocked when there is no data to be cleared. insem2 is initialised to 1 and ensures that no two input processes attempt to write simultaneously. outsem2 performs a similar function for output processes. By using two semaphores insem2, outsem2 we allow input and output processes to run concurrently when it is safe to do so.

It is seen that synchronisation using semaphores is not as direct as can be achieved with the MASCOT approach. Four control variables are required instead of two and the counting of full and empty spaces is effectively duplicated by the semaphore operations and the stepping of the pointers controlling data access. Furthermore the synchronisation protocol is not so readily understood. The advantage of the MASCOT approach mainly derives from the ability to program an explicit conditional wait (in fact the conditional wait can be achieved by using binary semaphores in place of general semaphores but in this case it is still necessary to have four control variables to achieve synchronisation). The explicit conditional wait is of course also a feature of monitors which will now be considered in more detail.

8.2 *Monitors*

The monitor approach to this synchronisation problem is based on the rule that only one process should be allowed access to the buffer at any one time. If a process finds that it has to wait it must release its access to the buffer to allow use by other processes.

MASCOT contains no facilities for LEAVEing one control (q1 say) whilst simultaneously JOINing or WAITing on

```
MODE SEMAPHORE = STRUCT(CONTROLQ q, INT i):

PROC P = (SEMAPHORE s):
BEGIN
 JOIN(q OF s);
 WHILE i OF s = 0 DO WAIT(q OF s);
 i OF s := i OF s - 1;
 LEAVE(q OF s)
END;

PROC V = (SEMAPHORE s):
BEGIN
 i OF s := i OF s + 1;
 STIM(q OF s)
END;

ACCESS PROC write = (CHAR x, REF CHARCHAN
   chan):
BEGIN
 P(inseml OF chan);
 P(insem2 OF chan);
 (data OF chan) [inpointer(chan)] := x;
 stepinpointer(chan);
 V(insem2 OF chan);
 V(outseml OF chan)
END;

ACCESS PROC read = (REF CHAR x, REF
   CHARCHAN chan):
BEGIN
 P(outseml OF chan);
 P(outsem2 OF chan);
 x := (data OF chan) [outpointer(chan)];
 stepoutpointer(chan);
 V(outsem2 OF chan);
 V(inseml OF chan)
END;
```

Note: inseml, insem2, outseml, outsem2 are SEMAPHOREs within CHARCHAN

Fig. 12 Channel access control using semaphores

another (q2 say). This is the sort of action necessary in order to implement Hoare's WAIT primitives on his condition variable. MASCOT is also deficient of the means of implementing the SIGNAL primitive. Thus to implement monitors we define Hoare's primitives in MASCOT terms (changing the name of WAIT to WAIT FOR SIGNAL to avoid confusion with the MASCOT WAIT). However since it is our aim to expose the underlying scheduling implications of these primitives we have given them each two control queue parameters. The first, q1, is the control queue used to guarantee that only one monitor procedure is active at a time. It is a rule in both cases that this control queue must have been JOINed by the caller. The second control queue corresponds to a condition variable. The primitives are defined:

WAIT FOR SIGNAL (q1, q2)—releases the mutual exclusion on q1 by performing LEAVE (q1) and places the calling activity in a first-in-first-out queue of waiting activities on q2.

SIGNAL (q1, q2)—if q2 contains any waiting activities then the front activity is taken off and is placed at the front of q1 whence it will be restarted by the next LEAVE(q1).

These primitives are not formally part of MASCOT but are introduced to facilitate the comparison with monitors. They constitute the minimum additions to MASCOT to allow monitor style programming.

In order to exercise monitor style control over access to a buffer of type CHARCHAN it is necessary to replace the two control queues by three which we will name qmutex, qcin, qcout. qmutex is the main queue controlling access to the buffer. qcin holds input processes which have to wait and qcout holds output processes which have to wait, i.e. they correspond to condition variables. The access procedures would be programmed as shown in **Fig. 13.**

The difference between MASCOT and monitors in the context of this problem is immediately apparent. The input and output monitor processes are subjected to control by a single control queue and this unfortunately increases the interaction between input and output particularly when the buffer is busy. Read and write procedures will normally be embodied in program loops within consumer and producer processes on either side of the buffer and there will be a high probability that these processes are blocked and restarted as they compete for access. This problem can be alleviated by introducing facilities for the dynamic adjustment of process priority but this complicates both the scheduling mechanism and the monitor procedure programs. In contrast the MASCOT approach reduces the interaction between input and output processes to the absolute minimum.

The importance of achieving a highly efficient method of interprocess communication using buffers of this type depends very much on the amount of message passing which takes place. MASCOT is aimed at supporting a system design method which is based on the extensive use of loosely coupled processes to limit and define interactions within the system. In this context it is most important to provide efficient message passing facilities.

Of course significant advantages of the monitor approach arise from rigorous run time protection of a data area and the possibility of using formal proof rules for program correctness. MASCOT relaxes the protection criteria and a pragmatic approach is adopted to the verification of access procedure algorithms. However the writing of access procedures can be confined to skilled programmers who can make use of approved standard techniques, and well proven test procedures can be used to test and authenticate the resulting software. The correct use of access procedures is ensured by compile and FORM time parameter checks. Thus a highly satisfactory set of practical tools is provided to cope with the problems arising from software production for concurrent processes.

```
ACCESS PROC write = (CHAR x, REF CHARCHAN
   chan):
BEGIN
 JOIN(qmutex OF chan);
 IF full(chan) THEN
 WAIT FOR SIGNAL(qmutex OF chan, qcin OF chan) FI;
 (data OF chan) [inpointer(chan)] := x;
 stepinpointer(chan);
 SIGNAL(qmutex OF chan, qcout OF chan);
 LEAVE(qmutex OF chan)
END;

ACCESS PROC read = (REF CHAR x, REF
   CHARCHAN chan):
BEGIN
 JOIN(qmutex OF chan);
 IF empty(chan) THEN
 WAIT FOR SIGNAL(qmutex OF chan, qcout OF chan)
   FI;
 x := (data OF chan) [outpointer(chan)];
 stepoutpointer(chan);
 SIGNAL(qmutex OF chan, qcin OF chan);
 LEAVE(qmutex OF chan)
END;

Note: qmutex, qcin, qcout are CONTROLQs in CHARCHAN
```

Fig. 13 Channel access control in the style of a monitor

9. Discussion

The MASCOT approach to programming, if applied in its entirety, impinges on every aspect of software design and implementation. It is however possible to make use of a subset of the MASCOT facilities and embed these within existing arrangements for compilation, loading, etc. In its most limited form a MASCOT system consists of the kernel (possibly excluding the interrupt handler) and the synchronisation primitives. Even then there is the option of implementing the kernel either on the bare machine so that it provides the fundamental level of software, or of building it on top of an existing operating system. Both approaches have proved successful in a number of applications. The MASCOT kernel has been implemented at the fundamental level on Marconi Myriad, CTL Modular 1, Ferranti FM1600B, Ferranti Argus 700, Intel 8080 and Texas Instrument 990 series computers and it has been constructed on top of host operating systems for Marconi Myriad, ICL 1907, CTL Modular 1 and DEC PDP11 machines.

The kernel is particularly compact and efficient. In the Marconi Myriad implementation on a bare machine the following approximate main store sizes were required for the basic software:

Despatcher and list handling (Fig. 2)	520 words
Scheduling primitives (Fig. 2)	500 words
Activity table (Appendix 1) for 50 activities	550 words
Subsystem table (Appendix 1) for 20 subsystems	80 words
Interrupt handler (Fig. 2) for 24 channels multiplexed on one level	30 words

The smallest viable system will usually require a command interpreter whose size will depend on the facilities provided; 500 words are sufficient for a basic interpreter, including the necessary typewriter control software. Thus it should be possible to construct a reasonably useful MASCOT kernel together with a basic interpreter in about 2,200 words of main store. Further facilities, e.g. the monitor (Appendix 2) and various loading and linking options (Appendix 1) will clearly require additional space. Time overheads are heavily dependent

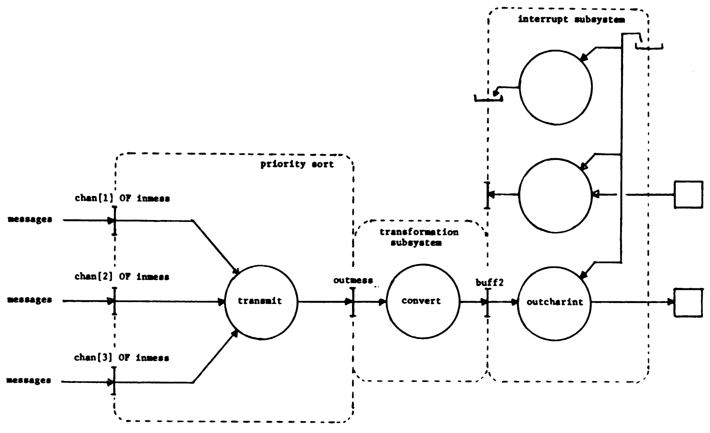

Fig. 14 Interconnection of three subsystems

on the machine architecture. Machines designed for re-entrant programming and which have good microprogramming facilities are ideal for minimising scheduling overheads.

The overall structure of MASCOT subsystems interconnected by channel and pool data areas can be used to produce a style of programming with a high degree of functional modularity. For example, **Fig. 14** shows how messages passing through three channels can be multiplexed so that they are transmitted in character form to a single output peripheral. The facility is achieved by the insertion of a message to character transformation subsystem between the priority sort subsystem of Fig. 9 and the interrupt subsystem of Fig. 7. This approach results in data being copied several times over from one data area to another, and contrasts with the more conventional method of using procedure calls to carry out particular programming functions. However it should be noted that modularity of this type is precisely that which is enforced by distributed processing architectures. The use of an individual subsystem to perform each function gives considerable flexibility and reduces the area of influence of component elements of software. If desired two or more subsystems can be amalgamated to increase efficiency. Thus MASCOT software structure gives the system designer the option of 'direct call' or 'indirect call through the operating system' (i.e. through a MASCOT channel) in a straightforward manner.

Although MASCOT can be used in a wide range of applications it is most suited to systems where most of the software is main store resident and is dedicated to a particular online requirement. The MASCOT design has its origins in such systems, examples of which are found in command and control applications having a large number of operators requiring rapid response from online terminals. In these systems the loading and linking flexibility (Appendix 1) is used during commissioning and subsequent development, and the generality of the approach allows offline analysis and support programs to be readily mixed in with the online software.

Returning now to the synchronisation aspect it is prudent to enquire as to the in-use safety of MASCOT programming. It is immediately apparent that JOIN statements can be combined in a manner which gives a potential deadly embrace situation. For example, the two sequences JOIN(x)... JOIN(y) ... and JOIN(y) ... JOIN(x) ... called on by two different activities run this risk. The answer to this problem is to adopt a nested hierarchical approach to the use of JOIN ... LEAVE statements so that at least one segment of program governed by any given control queue can always be active at any one time (i.e. we are safe from deadly embrace if JOIN(x) always precedes JOIN(y) in time sequence). Even then, from the practical point of view, it is essential to take note of potential bottle necks where many activities can be held on a single control queue whilst awaiting the use of some particular pool or channel interface.

Although the pitfalls associated with the synchronisation primitives are avoidable this can be difficult to achieve in a large project using many programmers with varying skill levels. In this case the recommended approach is to provide a comprehensive set of standard access procedures, channels and pools which are carefully inspected and tested before issue for general use; if this is done there should be no need for widespread and dangerous use of primitive procedures. Further safeguards can be applied by ensuring that access procedures are only used with compatible pools and channels. The ALGOL 68 mode and structure facilities can achieve this or, alternatively, a translator such as that used for the MASCOT Oriented Reliable Applications Language, MORAL (Harte and Jackson, 1976) can be used to check for compatibility prior to compilation.

The main body of this paper has concentrated on data intercommunication between activities and its associated synchronisation mechanisms. For this purpose the control queue primitive operations are ideally suited. These primitives can also be used within access procedures to a pool data structure to provide a convenient method for the management of resources. Each resource is represented by a boolean variable and access to the pool is synchronised by the access procedure(s) using a single control queue. Processes wishing to acquire resources must call a procedure giving a list of the requested resources as a parameter. The access procedure then ensures that each process in turn is allowed exclusive access to the pool and, if successful, the states of the resource variables are adjusted to indicate which resources are now secured. If a process is unsuccessful a 'not available' answer is returned and it must try again when resources have been released by other processes. The allocation algorithm can be made as sophisticated as necessary for the problem in hand, and deadly embrace can be avoided by imposing sensible rules for batching and ordering of resource requests which can be policed by the access procedures.

MASCOT, like other synchronisation techniques, works well provided that there are no 'hang-ups' caused by software or hardware faults. These faults can occur in a manner which prevents certain control queues from being released for use by other activities and this could progressively disrupt a major part of a computer load. It is important that a well designed system guards against such faults by ensuring that all subsystems are either running or ready to run. Special supervisory subsystems can be incorporated to perform a continuous programme of checks to establish that all is well throughout the system. Subsystems which are found to be faulty can be TERMINATED (Appendix 2). The checking and recovery strategies are of course heavily application dependent.

10. Conclusion

In this paper we have described the MASCOT approach to synchronisation and have shown how this can be used to form the basis of a method for process intercommunication. The principal features and advantages of the MASCOT approach are summarised below.

1. *Terminology*

The use of the CONTROLQ and the JOIN, LEAVE, WAIT, STIM primitives have the advantage of giving a clear indication of the underlying scheduling implications of process synchronisation. At a higher level the concepts of CHANNEL, POOL, ACCESS PROC, ROOT PROC, activity, subsystem have been found useful in describing and designing the components and functions of real time systems.

2. *Synchronisation*
The use of a matched set of four synchronisation primitives allows a logical approach to interprocess communication problems, particularly where a conditional wait facility is required. The resulting solution to the bounded buffer problem is neater and more efficient than can be achieved using conventional techniques for mutual exclusion and cross stimulation.

3. *Relevance*
MASCOT is designed to deal with the type of software required to support on line command and control systems in military, civil and industrial applications. In such systems the distinction between the operating system and user programs becomes extremely blurred. MASCOT provides a unified set of facilities for dealing with both the application programs and the conventional operating system functions and this results in a homogeneous and flexible approach to the software design and production problem.

4. *Overheads*
The basic kernel is compact and efficient. It is difficult to make direct comparison with other systems but it is believed that the size of the MASCOT kernel in relation to the facilities offered is highly competitive.

5. *Interrupts*
MASCOT provides an option for dealing explicitly with interrupts using the RESPONSE PROC, SET RESPONSE, CLEAR RESPONSE, CHANGE RESPONSE facilities. There is no requirement for interrupts to be buried away in the depths of an operating system and this is certainly an advantage in the class of real time systems mentioned in 3 above.

6. *Machine and language independence*
MASCOT style programming is independent of any particular programming language or hardware. As such it is possible to use it as a standard approach when dealing with a wide range of applications using a variety of different machines. Standardisation of this nature can be a great advantage when faced with the management problems associated with software procurement and maintenance for a large number and variety of systems. Machine and language independence is also of considerable technical importance when a system is being developed by construction of a software prototype on one machine for subsequent transfer to another.

7. *The MASCOT approach*
Although we have restricted the paper to the synchronisation aspects of MASCOT a brief description has also been given (Appendices 1 and 2) of the construction and test facilities. The coordinated approach which MASCOT adopts to construction, operation and test is an important feature of the method and of great practical significance in tying together the various stages of the software procurement process.

Of course many of the above features are present in other approaches to real time systems programming. However the comprehensive nature of MASCOT coupled with the straightforward approach to the principal aspects of real time software production is considered to give significant advantages. MASCOT is attracting considerable interest at the present time and is being used in a number of practical projects. An active research programme is aimed at extending its use and exploring its full range of applicability.

11. Acknowledgement

The authors wish to thank the members of the MASCOT Suppliers Association (MSA) for several suggestions and comments during the design and development of the MASCOT facilities. The MSA represents selected government establishments and system houses concerned with the transfer of MASCOT technology from a research environment to a fully engineered form suitable for widespread use.

Appendix 1 System construction facilities

MASCOT system construction is formulated in terms of the fundamental stages of the construction process. Thus we have the conventional facilities of COMPILE, LOAD and LINK, and to these is added FORM which provides the final stage of linking and work space allocation.

The basic unit of construction is the module. Each module has a name and it is a MASCOT convention that this name should be the same as the name of the procedure or data structure which the module represents. For compiling and loading purposes it is essential for the module name to be further qualified by a version number so that different editions of the same module can be identified.

Four types of file are used to support the construction process:

1. Source Text File (STF)—This file contains the source texts for all modules.
2. Compiled Code File (CCF)—The contents of this file are generated as a result of the compilation process.
3. Load Map File (LMF)—This file indicates which modules are currently loaded and gives their location in the machine.
4. System Element File (SEF)—This file lists those modules which can be used to form MASCOT subsystems, i.e. it contains load details of root procedures, channels and pools which have been 'linked' (see below).

The STF and CCF are essentially part of the user's data base and there may well be several sets of these files. There is only one LMF and SEF and these lie within the kernel data base.

In addition to these files there are three other important components of the kernel data base which are used in the construction process.

Activity table
This contains an element for each activity and records its state (running, held on current list, held on pending list, etc.) together with certain other key information (for example, program re-entry point, store index for work space) vital to the running of the activity. The current, delay and pending lists lie within the activity table which chains together activities on each list; thus the start of a list is represented by an index number denoting an entry in the activity table.

Subsystem table
This tabulates the current set of subsystems. It is indexed by subsystem name and contains a pointer to the first activity in the subsystem, remaining activities being chained together in the activity table.

Activity stack
This is an area set aside to provide the work space requirements for each activity as it is FORMed. The portion of work space allocated to an activity within the activity stack is called the activity environment.

We are now in a position to outline the various construction operations.

COMPILE
This takes the source text for a module (specified by name and version number) from the STF and places the result of compilation as an element in the CCF. In addition to generating the compiled code, this operation must also list details of work

space requirements and all external calls of and references to other modules so that linking can take place at a later stage.

LOAD
This takes an element (specified by name and version number) from the CCF and loads the compiled code into core. It records the details of the loaded module in the LMF. If the module is a system element and it requires no further linking (i.e. it is ready for use) then an entry is made in the SEF at this stage.

LINK
This acts on a loaded module with a view to establishing the links for all external calls and references. It also computes work space requirements for code modules taking into account its own requirements and those of any called modules. A module can only be linked to other modules which have themselves been fully linked and successful linking of a system element results in an entry in the SEF.

FORM
This is used to build up subsystems in the manner described in Section 2. The FORM operation checks the mode of root procedure parameters, sets up an entry in the activity table for each activity, secures an activity environment in the activity stack for each activity and loads the root procedure parameters into each activity environment.

It should be noted that the above method is based on static work space allocation where work space requirements are calculated at COMPILE time, consolidated at LINK time and allocated at FORM time. This rules out certain programming facilities such as dynamic setting of array sizes and unrestricted recursion, but produces the important advantage of avoiding the possibility of activities being held up by shortage of work space. If it is essential for work space to be allocated dynamically then this can still be arranged by means of a POOL with suitable ACCESS PROCEDURES for securing and releasing space.

Various techniques are possible for giving some flexibility in the use of space in the subsystem table, activity table, activity stack, and the remaining area outside the kernel available for system elements and other modules. Perhaps the simplest approach is to divide the computer into 'standard' and 'development' portions. In such a scheme the standard portion can only be changed when the computer is completely reloaded whereas the development portions can be rewritten as required. Alternatively facilities can be provided for removing any subsystems or module without stopping the machine. To give tight control and safe operation two additional files would be required:
Forward Cross Reference File (FCRF) listing system elements used by subsystems and modules required to support other modules.
Reverse Cross Reference File (RCRF) listing subsystems supported by each system element and modules supported by other modules.

It is then possible to provide two further facilities:
DELETE (subsystem) to recover the space in the subsystem table, activity table and activity stack.
REMOVE (module) to recover space in the area outside the kernel available for system elements and other modules.

LOAD, LINK, FORM, DELETE and REMOVE would all need to interact with the FCRF, RCRF to give a comprehensive system of cross checking on usage; this is not too difficult to provide although it does complicate the construction process to a degree.

The system construction facilities would normally be provided through a command interpreter although there is no reason why more complex systems should not allow these operations to be called from within programs. Inevitably any implementation will need to take account of existing compilers, loaders, file handling, etc.; this constraint has been found to cause little difficulty in practice.

Appendix 2 Test facilities

The basic structure and organisation of MASCOT software produces an excellent test environment. The construction facilities are extremely flexible and allow user facilities to be dummied, isolated or duplicated as required. This is of considerable advantage when testing software throughout all stages of implementation and development. There are also some additional MASCOT features which can be used for test purposes. Firstly a range of commands provides overall control at run time and secondly a monitor facility allows recording of user-kernel interactions arising from primitive procedure calls. These two features are briefly described below.

1. *Commands*
The following commands are available for control at run time:
START(subsystem, p). This starts a subsystem by placing its activities on the current list having priority p.
TERMINATE(subsystem). This stops a subsystem and places it in a state where it can only be restarted by a START command.
HALT(subsystem). This temporarily stops all activities in a subsystem either at the end of the current slice or on attempted re-entry from the despatcher. This is done in a manner which allows them to be continued from the same point at a later stage. It can be used to give a controlled pause for all activities in a designated subsystem.
RESUME(subsystem). This restarts a subsystem which has previously been stopped by the HALT command.
The commands listed above would normally be available through an online command interpreter (itself a subsystem) and would be used for normal control as well as for test purposes.

2. *MASCOT monitor facility*
The monitor facility in MASCOT is designed to provide a means of observing and recording the interaction between MASCOT subsystems and the kernel software. The facility is capable of providing a record, in strict sequence, of slices initiated by the executive programs (i.e. the despatcher and the interrupt handler) and primitives called by active subsystems. The events recorded by the monitor take place at the fundamental level of operation and occur at a high rate under normal conditions. Thus it is necessary to provide comprehensive selection facilities to limit recording to specific areas, together with some means of buffering information to produce complete and continuous short term records.

The recording element of the monitor software is an integral part of the executive programs and primitive procedures lying within the kernel. It has been possible to provide a facility which carries negligible overheads when not in use, and which minimises any disturbance to the natural timing of events caused by the recording mechanism. Output of the record is implemented in the form of a privileged subsystem having access to the kernel data base. Control of the facility is exercised by means of a command interpreter.

The MASCOT monitor records the occurrence of selected events where selection is exercised in terms of primitives, activities and queues:

Primitives
All primitives previously mentioned can be selected to be the subject of monitoring. In addition the start of a slice by an executive program is also included as a pseudo-primitive called ENTER.

Activities
These may be selected in terms of complete subsystems or individual activities within a subsystem.

Queues
A queue can be selected in terms of its main store address or other suitable reference number or identifier. Since this may not be readily available a special facility (see ALLQS option below) is included to obtain this information.

The three selector fields above ensure that a selected primitive must be called from within a selected activity and, if it has a control queue parameter, must operate on a selected queue before recording can take place.

Monitor data is gathered in a special circular buffer and various commands are provided for overall control. Input to the buffer can be switched on and off as required. Output from the buffer can be arranged to give a snapshot of immediate past events or to provide as complete a continuous record of events as is possible within the limitations of buffer and output capacity.

Certain optional extra items of data can be recorded if desired:
TIME—The time of occurrence of each event.
SLICE—The duration of each slice.
LOCATION—The address associated with each event (i.e. the point of call of a primitive or the location to which an executive program transfers control).
ALLQS—This allows a user to override the queue selection mechanism and this produces a record containing queue addresses, reference numbers or identifiers which can be used subsequently to select the control queues of particular interest.
The monitor facility is an important aid during the initial stages of implementation of a MASCOT system when it can verify the correct operation of the executive programs and primitive procedures. In an established system it can be used during the implementation and test of access procedures and interrupt control subsystems. It can also provide valuable timing information at all stages of a test program. The MASCOT monitor can be enhanced by arranging for other forms of online test aid to write their information in the monitor output buffer, so allowing the production of a single time sequence record for a wide range of preselected events.

References
Dijkstra, E. W. (1968). Co-operating Sequential Processes, *Programming Languages*, Academic Press.
Ford, W. S., and Hamacher, V. C. (1977). Low Level Architecture Features for Supporting Process Communication, *The Computer Journal*, Vol. 20, No. 2, pp. 156-162.
Harte, H., and Jackson, K. (1976). Achieving Well Structured Software in Real Time Applications, *Proceedings of IFAC/IFIP International Workshop on Real Time Programming*, IRIA, France.
Hoare, C. A. R. (1974). Monitors: An Operating System Structuring Concept, *CACM*, Vol. 7, No. 10, pp. 549-557.
Jackson, K., and Simpson, H. R. (1975). MASCOT—A Modular Approach to Software Construction Operation and Test, *RRE Technical Note No.* 778, October 1975.
Kammerer, P. (1977). Excluding Regions, *The Computer Journal*, Vol. 20, No. 2, pp. 128-131.
Shrivastava, S. K. (1975). A View of Concurrent Process Synchronisation, *The Computer Journal*, Vol. 18, No. 4, pp. 375-379.
Woodward, P. M., and Bond, S. G. (1974). *ALGOL 68-R Users Guide*, HMSO.

Book reviews

Microcomputer Architectures, edited by J. D. Nicoud, J. Wilmink and R. Zaks, 1978; 283 pages. Proceedings of Euromicro, 1977. (*North-Holland*, $40·00)

The proceedings of a 1977 conference on 'Microcomputer architecture' might sound a little dated, and this would indeed be the case for those seeking details of the most recent products. Most of the systems discussed make use of the established eight bit processors. The chosen papers are, however, respectable contributions concerned with the solution of specific problems in original and improved ways. Many of the problems are not new, or specific to microprocessor systems, but will remain topical for some years yet.

The selection panel appear to have struck a good balance between the highly theoretical studies and the systems and applications areas. The most interesting central themes are multimicroprocessor configurations, shared bus structures, synchronisation and speed limitations; shared memories; fault tolerant systems and reliability. Applications areas include data communications systems, graphics, automatic control and signal processing.

Some papers are not strictly concerned with microcomputers, viz 'A modular microprogrammable pipeline signal processor in ECL-technology', but these infiltrators are of high quality. The hardware designers get good coverage, with bus standards, programmable logic arrays, languages for logic design and simulation of central processors. With only two papers devoted to programming languages, and many of those on cross assemblers in the reject pile, the selection panel clearly felt that there was little progress to report in this area. They did accept a very good paper describing a universal cross assembler, which must be of commercial value.

In all, 31 papers were presented at this truly European affair, with only one from Britain! If this reflects our true position, then I urge you all to buy this, and many other books on microcomputers. At $40.00 you may prefer to buy another 8080 just in case they ever come back into fasion.

I would echo the panel chairman's comments, that insufficient research and development is reported on programming tools and languages. The large number of systems being written in BASIC and Assembler gives little satisfaction to those who struggled to provide current high level languages in the face of the same scepticism in their day as that voiced by the current wave.

This book is for the departmental libraries of those actively involved with microcomputer research and development.

I. P. Page (Uxbridge)

Information Representation and Manipulation in a Computer 2nd edition by E. S. Page and L. B. Wilson, 1979; 271 pages. (*CUP*, £10·00; £3·95 paper)

This is the welcome second edition of a now well established introductory text on data structures, brought up to date and somewhat expanded. Starting with a section on the various data representations —including data compression and error detection and correction— the authors continue with chapters on the basic data structure of arrays, linear lists and trees and finish with a short chapter each on searching and sorting. Lots of exercises with solutions or suggestions given, and each chapter has its own bibliography. At £3·95 in soft cover, it would be difficult to find better value for hard-up students.

Arthur S. Radford (Leicester)