A building block approach to multiprocessing

by R. L. DAVIS, S. ZUCKER and C. M. CAMPBELL

Burroughs Corporation
Paoli, Pennsylvania

INTRODUCTION

Today most computing systems have dedicated, self-contained, single processors. When the load on the system exceeds its processing capabilities, it is necessary to replace the original system with one of greater capacity. It would be far better if the original system had been modular in nature, so that additional processing modules could be added in a multiprocessing configuration with the growth in processing requirements, just as memory modules or I/O devices are added with the growth in storage or peripheral requirements. Multiprocessing systems are not new, but they have been previously limited by the processing time consumed in controlling the processor modules. With the advent of microprogrammed computers, however, control functions can now be implemented in microcode, and executed at a much faster rate than has been previously possible. In addition, microprogrammed computers are simpler and therefore more reliable than conventional computers.

This paper describes a structure for connecting and controlling a multiprocessor system using a building block technique. The hardware is modular and includes microprogrammable processors called “Interpreters”, memory modules, and devices. Each Interpreter is interconnected with every memory module and every device via a data exchange network called a “Switch Interlock”.

The current operating system is a simple but comprehensive control program which allows for all the basic capabilities of operating systems emphasizing multiprocessing, multiprogramming and error recovery. Plans are being developed for an extended operating system which would be a set of independent units of microcode selected from a library of such units for each individual system. The building block approach has been used by both the hardware and software implementors of the system.

MULTIPROCESSOR INTERCONNECTION

A major goal in multiprocessor system design is to increase system efficiency by the sharing of available resources in some optimal manner. The primary resource, main memory, may be more effectively shared when split into several memory “modules”. A technique for reducing delays in accessing data in main memory is allowing concurrent access to different memory modules. With this concurrent access capability present, an attempt is made to assign tasks and data to memory modules so as to reduce conflicts between processors attempting to access the same memory module. Nevertheless, since some conflicts are unavoidable, a second technique (reduction of conflict resolution time) is required. These two techniques are largely a function of the multiprocessor interconnection scheme which has been discussed by Curtin and others.

Figure 1 shows three basic functional interconnection schemes. These are described in more detail in Curtin. The disadvantages of the single bus approach (Figure 1) for many processors are:

1. the obvious bottleneck in information transfer between processors and memory modules due to both bus contention and memory contention
2. the catastrophic failure mode due to a single component failure in the bus

A solution to the first problem has been to increase the frequency of operation of the bus.

The multiple bus approach is merely an extension of the single bus approach where all processors contend for use of any available (non-busy) bus. The advantages are redundancy and allowing an appropriate number of buses (less than the number of processors) to handle the traffic between processors and memory modules.

The third approach utilizes a dedicated bus structure (one per processor). Although this approach requires more buses, it requires neither the logic nor, more im-
portantly, the time for resolving priority between processors requesting the use of a bus. Proponents of this approach contend that the time penalty for resolving conflicts for access to a memory module is enough of a price to pay without having to wait for the availability of a bus.

In a Hughes report, the authors distinguish the physical differences between two multiprocessor interconnection schemes. The two approaches (one called multiport and the other called matrix switch) are shown in Figure 2.

The Hughes report characterizes the two connection approaches as follows:

"In the multiport approach, the access control logic for each module is contained within that module, and intercabling is required between each processor and memory pair. Thus, the total number of interconnecting cables is the product of the number of processors and the number of memories. Each module must be designed to accommodate the maximum computer configuration.

"In the matrix switch approach, the same interconnection capability is achieved by placing the access control logic for each module in a separate module. The addition of this module to the system is compensated for by reducing the intercables required to the sum of the processors and memories rather than the product and by not penalizing the other modules with maximum switching logic.

"There generally is no speed differential between multiport and matrix arrangements. The major difference lies in the ability to grow in wiring complexity. Multiprocessors with multiport arrangements are generally wired, at production time, to the maximum purchased configuration. Future subsystem expansion generally requires depot level rewiring. This problem generally does not exist with the matrix arrangement. The maximum capacity is wired in but the switching logic complement reflects the purchased system. Subsystem expansion entails purchase of added processor/memory modules (and necessary cabling if required) plus the required switch matrix logic cards."

An important additional point is that even though all parts of the matrix switch are co-located, they must be designed such that the failure of one node is equivalent
to failure of only one element attached to the matrix switch. For example, failure of a processor-related node must not disable paths from any other processor to any memory module.

Apparent from the arguments in the Hughes report is the desire to reduce the number of wires interconnecting the processors and memory modules. A way to reduce the wiring (in addition to the use of the matrix switch) is by using serial transmission of partial words at a frequency several times that of the processors. This technique has been used by Meng and Curtin. The tradeoff here is between the cost of the transmitting and receiving shift registers and the extra logic necessary for timing and control of the serial transmission versus the cost of the wiring and logic for the extra interconnection nodes for a fully parallel transmission path.

Another factor adversely affecting efficiency in a multiprocessing system is a variation in the amount of computation versus I/O processing that must be done. In previous multiprocessing systems I/O functions and data processing functions have been performed in physically different hardware modules with devices being attached only to the I/O controllers (Figure 3). (This technique is typical of Burroughs DS25, B 5500, or B 6700.) In the Burroughs Multi-Interpreter system, however, processing and I/O control functions are all performed by identical Interpreters whose writable microprogram memory can be reloaded to change their function. This technique allows a configuration (Figure 4) in which the devices are attached to the same exchange as the memories and processors.

**Switch Interlock**

The Multi-Interpreter interconnection scheme for forming a multiprocessor is called a "Switch Interlock."
Switch Interlock from fully parallel to fully serial. Functionally, the Switch Interlock consists of parallel-serial conversion registers for each Interpreter, input and output selection gates, parallel-serial conversion registers for each memory module and each device, and associated control logic. Figure 5 outlines the implementation of the Switch Interlock and shows the functional logic units repeated for each Interpreter, memory module, and device. The bit expandability of the Switch Interlock is shown by dashed lines between the input/output switches and the shift registers associated with the memory module, devices, and Interpreters.

The six basic Switch Interlock modules are described below:

1. Memory/Device Controls (MDC)

   The MDC is an interface between the Interpreter and the controls described below (MC and DC), and the control for the high frequency clock used for the serial transmission of data. There is one MDC per Interpreter.

2. Memory Controls (MC)

   The MC resolves conflicts between Interpreters requesting the use of the same memory module and maintains an established connection after completion of the operation until some other module is requested or some other Interpreter requests that memory module. This unit handles up to four Interpreters and up to eight memory modules. System expansion using this module may be in number of Interpreters or in number of memory modules.

3. Device Controls (DC)

   The DC resolves conflicts between Interpreters trying to lock to a device and checks the lock status of any Interpreter attempting a device operation. This unit handles up to four Interpreters and up to eight devices. System expansion using this module may be in number of Interpreters or in number of devices.

4. Output Switch Network (OSN)

   The OSN sends data and control from Interpreters to addressed memory modules (i.e., the OSN is a "demultiplexer"). This unit handles wires for up to four Interpreters and eight devices or memory modules. The number of wires for an OSN (and for an ISN) depends upon both the packaging and the "serialization" selected for the application.

5. Input Switch Network (ISN)

   The ISN returns data from addressed devices or memory modules to the Interpreters (i.e., the ISN is a "multiplexer"). This unit also handles wires for up to four Interpreters and up to eight devices or memory modules.

6. Shift Register (SR)

   These units are optional and are parallel-to-serial shift registers or serial-to-parallel shift registers using a high frequency clock. These are used for serial transmission of data through the ISN's and OSN's. Their size, number and location are determined by system parameters.

Switch Interlock Block Diagram

Figure 6 is a block diagram of a Switch Interlock connecting up to four Interpreters to eight devices and eight memory modules. The shift registers shown are optional and may be eliminated with the resulting increase in the width of the ISN and OSN transfer paths. Although it is not indicated by this figure, the Switch Interlock is expandable in terms of number of Interpreters, devices, memory modules, and path widths.

Overall Switch Interlock Control

In an Interpreter based system utilizing one or more Interpreters, only Interpreters can issue control signals.
to access memories or devices. A memory module or
device cannot initiate a path through the Switch Inter­
lock. It may, however, provide a signal to the Inter­
preter via a display register or other similar external
request device. Transfers between devices and memories
must be via and under the control of an Interpreter.

Controls are routed from the Interpreters through
the MDC to the MC and the DC which, in turn, check
availability, determine priority and perform the other
functions that are characteristic of the Switch Inter­
lock. Data and addresses do not pass through the MDC.

Switch Interlock Timing

Events are initiated by the Interpreter for access to
memories or devices. The Interpreter awaits return sig­
nals from the MDC. Upon receipt of these signals, it
proceeds with its program. Lacking such positive return
signals, it will either wait, or retry continuously, de­
pending upon the Interpreter program (and not on the
Switch Interlock). Any timeout waiting for a response
may be performed by either the program or a device
that will force a STEP in the micropogram after a preset
length of time.

Among the significant signals which are meaningful
responses to an Interpreter and testable as conditions
are the following:

<table>
<thead>
<tr>
<th>Signal Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Switch Interlock has</td>
</tr>
<tr>
<td>Accepted Information</td>
</tr>
<tr>
<td>Read Complete</td>
</tr>
</tbody>
</table>

The address and data output registers of the In­
terpreter may be reloaded and a memory or device has been con­
connected.

Data are available to be gated into the input
data register of the Inter­
preter.

The rationale for this "handshaking" approach is
consistent with the overall Interpreter-based system de­
sign which permits the maximum latitude in the selec­
tion of memory and device speeds. Thus the micro­
programmer has the ability (as well as the responsibil­
ity) to provide the timing constraints for any system
configuration.

Device Operations

The philosophy of device operations is based upon an
Interpreter using a device for a "long" period of time
without interruption. This is accomplished by "lock­
ing" an Interpreter to a device. The ground-rules for
device operations are listed below:

1. An Interpreter must be locked to a device to
   which a read or a write is issued.
2. An Interpreter may be locked to several devices
   at the same time.
3. A device can only be locked to one Interpreter at
   a time.
4. When an Interpreter is finished using a device, it
   should be unlocked so other Interpreters can use
   it. The exception is the case where devices locked
to a failed Interpreter might be unlocked with a
"privileged" instruction by another operative
Interpreter.

Memory Operations

Memory modules normally cannot be locked and are
assumed to require minimum access time and a short
"hold" time by any single Interpreter. Conflicts in ac­
cess to the same module are resolved in favor of the
Interpreter that last accessed the module, or otherwise
in favor of the highest priority requesting Interpreter.
Once access is granted, it continues until that memory
operation is complete. When one access is complete,
the highest priority request is honored from those In­
terpreters then in contention. The Interpreter complet­
ing access is not able to compete again for one clock.
Thus the two highest priority Interpreters are assured
of access. Lower priority Interpreters may have their
access rate significantly curtailed. This problem is re­
solved through careful allocation of data to memory
modules.

Other Multi-Interpreter System Hardware

The executive concept described in the latter part of
this paper requires special hardware features. Although
in a multiple Interpreter system more than one In­
terpreter can execute executive programs at one time,
certain tables used by the executive must be locked for
modification by only one Interpreter at a time. The
Interpreter remains in this "table modifying" state for a
short time, thus minimizing executive conflict delays.
The locking capability is implemented with two "global
condition bits" per Interpreter which are chained to
other Interpreters in a priority scheme. These global
condition bits are 2 of the 16 testable condition bits in
each Interpreter. Each bit can be set in only one In­
terpreter at a time and must be programmatically reset
(the other condition bits in each Interpreter are reset
when they are tested). An Interpreter instruction containing the “Set Global Condition Bit” operation will set the specified global condition bit in that Interpreter only if that bit is not set in any Interpreter and no other higher (wired) priority Interpreter is requesting the same bit to be set in its own Interpreter.

Figure 7 shows the method of resolving priorities for each of the global condition bits. The instruction to set a global condition bit is actually a request to set a global condition bit. This request is latched for one clock time during which time a decision is made to honor this request or not. The global condition bits are programmatically reset independent of other Interpreters.

The top string of horizontally connected gates (Figure 7) OR’s is the corresponding global condition bit from all Interpreters together. The other string of horizontally connected gates in the Interpreters is the wired global condition setting priority. This priority could be wired differently for the two global condition bits. It should be noted that in a system with many Interpreters, these two “carry” chains could have a delay long enough to affect the Interpreter clock rate. In these cases, a carry-lookahead scheme could be used to speed up this path.

It should also be noted that such a scheme could be implemented alternately via programming using Dijkstra’s semaphores, or using memories with a read-modify-write cycle to insure one processor wasn’t reading a memory location in the interval between a read of that location by another processor and a subsequent write of a modified value.

One more of the testable condition bits in each Interpreter is wired to provide an additional inter-Interpreter signal. This bit is called the Interrupt Interpreters bit and is simultaneously set in all Interpreters by an operation originating from any Interpreter. This bit is reset in an Interpreter when tested in that Interpreter.

The global condition bits and the interrupt Interpreters bit are the only multiprocessing hardware features which are part of an Interpreter. Special purpose logic in an Interpreter is thus minimized, making it inexpensive and flexible enough to be used across a wide spectrum of architectures from simple stand-alone device controllers through multiprocessors. This versatility increases the quantity of Interpreters being produced, which in turn lowers the unit cost of an Interpreter.

Special devices (connected through the Switch Interlock) needed for multiprocessing are an interrupt display register and a real-time clock with time-out capability. The interrupt display register is needed due to the limited number of externally settable condition bits in each Interpreter. Such a display register is read as a device by any Interpreter. The response of the display register is a function of the interrupt handling philosophy of the particular application and the design of such a device could be varied without affecting the basic design of the Interpreter or the Switch Interlock.

The same philosophy is true of the real-time clock. Here, the intent of such a device is to provide an external counter for all Interpreters as well as a means of
forcing a program STEP in an Interpreter if that Interpreter didn’t periodically reset its real-time clock. This would prevent an Interpreter from waiting forever for a response from an inoperative memory or device.

MULTI-INTERPRETER CONTROL PROGRAM

The Multi-Interpreter Control Program is a simple, yet comprehensive, operating system characterized by the following capabilities:

(1) Multiprogramming and multiprocessing
(2) Fully automatic operation with manual intervention capability.
(3) Error recovery with no loss of data.

Multiprogramming and multiprocessing have characterized Burroughs operating systems for many years. In previous systems, input/output functions and data processing functions have been performed in physically different hardware modules; I/O modules for the former and processor modules for the latter. In the Multi-Interpreter System, however, I/O control and processing functions are all performed by identical Interpreters, and any Interpreter can perform any function simply by a reloading of its microprogram memory. In the Multi-Interpreter Control Program I/O operations simply become tasks which are indistinguishable to the control program from data processing tasks except that they require the possession of one or two I/O devices before they can begin to run. (A task is defined as an independent microprogram and its associated “S” level program and data, which performs explicit functions for the solution of user problems.) Whenever an Interpreter is available it queries the scheduling tables for the highest priority ready-to-run task, which may be an I/O task, a processing task, or a task which combines both processing and I/O functions.

The control program operates automatically, as well as under the control of the operator. The operator may enter new tasks, delete old ones, or change the priority of any task. He may add or delete hardware modules, and call for diagnostic programs to be run. The operator enters his commands through either the card reader or the alphanumeric display console.

The Multi-Interpreter Control Program includes an automatic error detection and recovery capability. All data is stored redundantly to avoid loss of data should a failure occur. The control program maintains this redundancy, in such a way that a restart point is maintained at all times for every task.

Figure 8 defines the flow of tasks through an Interpreter.

Task scheduling and initiation

Every time an Interpreter has no task to perform it scans a Task Table and locates the highest priority ready-to-run task. If there is no ready-to-run task this Interpreter runs the diagnostic program, then (if the diagnostic program indicates no malfunction) again attempts to schedule a task. When the Interpreter finds a task which can be run the appropriate table entries are updated. The Interpreter then loads its microprogram memory with the task’s code and begins to execute this code. The task code must periodically cause the Interpreter to “report in” to a central “timeout” table.

Task termination

The task retains the Interpreter until the task is due to be suspended (assuming that a timeout does not occur). The task first stores its state, then causes the Interpreter to load its microprogram memory with a portion of the control program. This recopies changed memory areas to ensure the redundancy required for error recovery, deallocates resources which the suspended task had used, and updates system tables. Then the Interpreter looks for another task.

Supervisory control

The operator can add or delete tasks or resources as well as enter the data and code which individual tasks
require. Commands and data may be entered via either the card reader or an alphanumeric display. Each of these devices is "owned" by a system task which inputs data for the control program and for the other tasks in the system. Each entry via the card reader begins with a control card which specifies the nature of the entry. When the entry is data or program, the control card specifies the task and the area in that task where subsequent cards are to be loaded.

Commands and data are entered from the alphanumeric display in essentially the same manner as from the card reader, a line on the display corresponding to a card.

Error recovery

The hardware of the Multi-Interpreter System detects failures in memory modules and in I/O devices. When an I/O device fails, it is indicated as "not available" and a message to this effect is sent to the operator (via both printer and display). The device will remain "not available" until the operator enters a message to the contrary. The task whose running caused the malfunction to be detected is immediately aborted, but it will again run when the needed I/O device or an alternate is available. The redundant copies of storage areas serve as the restart point for this rerun. (The primary copies cannot be used as the task was aborted at other than a normal suspension point, hence not at a valid restart point.)

When the core memory module fails while a task is using it, the Interpreter which was running this task marks the failed module "not available", and causes the initiation of an error message to the operator. The task which was being run is aborted. This task will be reinitiated from the redundant core memory areas. Interpreter failures are detected in either of two ways: by the diagnostic program or by a timeout method, in which every Interpreter "checks in" to a central table at periodic intervals, and every Interpreter checks that every other Interpreter is checking in on schedule. When an Interpreter fails it halts (or is halted), and the task (if any) which it had been running is aborted, and will eventually be reinitiated at its restart point by an operative Interpreter.

The system can thus recover from a failure with no loss of data, provided the primary and alternate storage areas of a task are not both lost.

Tables

The operation of the Multi-Interpreter Control Program can be seen in more detail by examining the tables which it utilizes. These tables are divided into two classifications: system (global) tables, and task tables. The former are located in core memory in a segment called the "System Control Segment", and the latter for each task are located in a "Task Control Segment" for that task.

For error-recovery purposes all of these tables are stored redundantly in two different memory modules. Therefore there are actually two "System Control Segments", a primary one and an alternate one. The two copies are identical at all times and whenever one is updated the other is also. In a system that includes bulk storage this redundancy would be maintained in the bulk storage, and only one copy of any segment would be in core memory when the task is running.

The tables for each task are similarly stored redundantly so that there are two copies of each Task Control Segment. Included in each Task Control Segment are pointers to all of the data and program areas which this task may use. These areas may comprise one segment or many contiguous segments, and may be used either solely by this task or shared with other tasks. Each such area is also stored redundantly.

Figure 9 illustrates how the core memory is utilized, and shows the System Control Segment and its relationship to the Task Control Segments. (The term "segment" refers to a 256 word area of core memory.) The System Control Segment includes a pointer to each Task Control Segment, which in turn includes the pointers to each area used by the task.

Memory modules are presently paired, where both modules of the pair are allocated identically, so that the primary Task Control Segment and the redundant Task Control Segment for the same task have the same segment number. The same applies to data and program areas.

System control segment

The System Control Segment is illustrated in more detail in Figure 10. This segment is always located in segment zero (the first segment) of some memory module. Its redundant copy is always located in segment zero of some other module. Segment zero in all modules is reserved for possible use as the System Control Segment or its alternate copy. If the module containing the prime or redundant copy of the control segment should fail, any remaining module is selected and its segment zero is overwritten with the contents of the System Control Segment so that redundancy is immediately restored. Word zero in segment zero of every memory module contains pointers to both the prime and al-
Figure 9—Memory utilization

*ONE PER PROGRAM OR DATA AREA
<table>
<thead>
<tr>
<th>Task Table</th>
<th>Resource Availability Table</th>
<th>Memory Module Availability</th>
<th>Memory Map</th>
<th>Interpreter Table</th>
<th>Error Flags: Printer</th>
<th>Error Flags: Display</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entry No. 1</td>
<td>1 = Resource Available</td>
<td>1 = Module Available</td>
<td>1 = Available</td>
<td>Time Next Report Due</td>
<td>1 = Error Associated With This Unit</td>
<td>Error Reports:</td>
</tr>
<tr>
<td>Entry No. 2</td>
<td>Identity of Interpreter Using This Resource</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Interpreter Failure, Task Timeout, I/O Device Failure, Memory Module Failure, Diagnostics Complete, Illegal Card</td>
</tr>
<tr>
<td>Entry No. 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 10—System control segment
ternate System Control Segments, so that any Interpreter can “find its way back” to the System Control Segment should it become “lost” because of a memory module failure. Word zero of the System Control Segment itself is no exception, and includes these pointers.

Resource Availability Table

This table contains one entry for every hardware resource. The entry includes a flag-bit, which indicates whether this resource is available or in use, and a field which, when the resource is in use, gives the identity of the Interpreter which is using it.

Memory Module Availability Table

This one-word table consists of one bit for every memory module in the system. This bit indicates whether or not the module is “up” (available for use).

Memory Map Table

This table contains one bit for every usable segment in core memory. A “1” in the bit position corresponding to a particular segment means that the segment is unassigned. Since modules are paired, a single bit in this table actually represents two segments, the prime segment and the segment in the paired module which holds the redundant copy.

Interpreter Table

This table has one entry for every Interpreter, and is used primarily to implement the “reporting” scheme used to detect Interpreter malfunctions. Every task run on the system is written so that, if it must run for “N” or more seconds, the Interpreter will “report in” to its entry in the Interpreter Table at intervals of less than “N” seconds. This reporting consists of replacing the “Time Next Report Due” field with the present time plus “N” seconds. Every Interpreter, when it completes its present task and is ready to find a new one, scans the Interpreter Table (after first updating its own entry in this table), looking for overdue entries. If any are found, they are reported to the supervisor. In addition, the most significant bit in the entry which was found overdue is set, so that other Interpreters will not make this same report.

Each entry in the Interpreter Table also includes two other fields. The first of these is a 1-bit field indicating that diagnostics are to be run on this Interpreter. This bit becomes set from a command entered by the supervisor. When set, the Interpreter will run diagnostics at the completion of the present task. Once diagnostics have been run the Interpreter causes a message to be sent to the supervisor indicating the test results, and then resets the “Run Diagnostics” bit.

The final field in an Interpreter Table entry gives the identity of the task which that Interpreter is currently running.

Error Flags Table

This table indicates the error messages which are to be sent to the operator. Such messages are sent both via the line printer and via an alphanumeric CRT display which is only used by the operator. Each portion of the table consists of one word for every type of error. Within each such word, the individual bits form an indexed vector which indicates more specific error information, normally the unit number of the failed unit.

Changing the System Control Segment

Whenever any change to a table in the System Control Segment must be made, the segment is locked. (This locking is implemented using the “global condition” bits found in every Multi-Interpreter System.) The required writing then takes place, to both the prime and the redundant copy of the table, and the segment is then unlocked.

The task table

The Task Table contains information about every task in the system. Figure 11 indicates the contents of an entry in this table, which consists of two words. The first word contains the information which an Interpreter in the process of finding a new task must examine to determine whether this task is the one which should be
selected. The most significant bit in this first word is the "not-running" bit. This bit is reset whenever the task is running and set at all other times. Next are the "ready-to-run" bits. These bits must all be ones for the task to be scheduled. If one or more of its bits are zero, the task lacks something, other than an I/O device, which it requires for running. For example, if the task requires data to be supplied by some other task, one of its ready-to-run bits would remain reset until the completion of the other task. Similarly, if the task in ques-
tion is awaiting data from the card reader, a bit will be left reset and will be set by the Card Reader Task after the data has been loaded. These bits have no system-wide assignments, but are assigned for each application. All unassigned bits remain set at all times. Any task can reset any of its own bits, and can, upon its suspension, set bits for other tasks.

The next field in the Task Table Entry is the “present-priority” field. Each task has a priority, and that ready-to-run task with the highest priority is the one which is selected.

In order for a task to be “ready-to-run”, not only must all of its ready-to-run bits be set, but also the core memory which it requires, and any I/O devices which it may need must be available. The remaining fields of the first Task Table Entry word deal with these items.

There is a field which specifies the memory which this task requires, and two fields which can specify up to two I/O devices which it also requires. Before the task can be considered “ready-to-run”, the required areas of memory, and all required I/O devices must be available.

The second word of the Task Table Entry contains additional information about the task. The first such field specifies the priority to which the task reverts after it has completed running. (A priority-at-completion of zero indicates that the task is to be dropped from the system at completion.) Next is the location of the Task Control Segment for this task, as discussed in connection with Figure 9. Another field contains the identity of the Interpreter which is running this task.

The final field in the Task Table Entry gives the status of the core memory areas used by this task. The

<table>
<thead>
<tr>
<th>Pointer to Data Area</th>
<th>Address of This Module’s Seg’m’t</th>
<th>TTE Address (2 X Task Num.)</th>
</tr>
</thead>
</table>

Figure 12—Task control segment
Task Control Segment and all program and data areas associated with task exist in two different memory modules. When the task runs, only one module is actually used, and the other holds the state of this task as it existed after it was last processed. This then serves as a "restart point" should the present processing be prematurely terminated by a failure. This alternate area is not changed until this present processing has been successfully terminated and the primary copy of the task's Task Control Segment updated to form a consistent whole which can itself be used as a restart point. Only then is the Task Control Segment and all data areas copied from the primary to the alternate module.

**Task control segment**

Figure 12 indicates the contents of the Task Control Segment. The first field points to the end of the Reference Area, since the size of the Reference Area may differ from task to task. (The otherwise unused end of the segment may be used as a work area by the task.) The second field gives the address of this module, so that, when the task is completed, word zero of segment zero can be found. (This word must be accessed in order to locate the System Control Segment.) The third field in this first word gives the identity of the present task, and enables the Task Table Entry for this task to be located once the System Control Segment has been found.

The next section of the Task Control Segment holds the state of the task, and also task parameters. First is the address of the task code at which the task is to be restarted. This task code then interprets the remainder of the state information, if any.

The next section of the Task Control Segment contains a copy of the Task Table Entry (from the System Control Segment), put there by the control program when the task is initiated. The task may then conveniently modify its Task Table Entry. For example, the task may reset one or more of its "ready-to-run" bits, may indicate the identity of an I/O device which is required for further processing, or may even change its priority. When the task is suspended, and it runs until it suspends itself (unless it is suspended because of a time-out), the Task Table Entry in the System Control Segment will be updated by the control program to correspond to the Task Table Entry as found in the Task Control Segment.

In addition to modifying its own Task Table Entry, the task may, upon its suspension, cause Task Table Entries for other tasks to have one or more ready-to-run bits set. This is accomplished through the next word in the Task Control Area. This word allows the task to specify up to four other tasks, and to supply a bit pattern to be "ored" with the ready-to-run bits for each of these tasks. In this way the completion of one task can be used to initiate another.

The final section in the Task Control Segment is the reference area, which points to data and program segments which this task may use. Each reference area, in addition to specifying the address of the segment(s) to which it refers, also contains a "read-only" bit, and the control program does not update the alternate copy of any entry so tagged.

In summary, the Multi-Interpreter Control Program allows any number of Interpreters to operate concurrently in a multiprocessing and multiprogramming mode, allows tasks and hardware modules to be added and deleted by the operator, and provides error recovery capability.

**A BUILDING BLOCK APPROACH TO SOFTWARE**

There are two approaches that may be followed when developing an operating system for a multiprocessor with microprogrammable Interpreters. One is an approach taken in the paper by Davis and Zucker as well as B. J. Huberman where a machine language is developed. This language includes instructions which aid in the development of an operating system. Since operating systems are organized about tables, lists, queues and stacks the machine language developed allows for easy handling of tables and other data structures.

We have presently chosen to take a building block approach. The Multi-Interpreter Control Program which is our current system, presents the operating system as a set of modules which may be obtained and run by an Interpreter when needed. A new technique currently under development generates an operating system which is basically a set of independent "S" instructions which may be used by any task on a defined configuration of firmware and hardware.

The building block approach used in the hardware implementation of the multiprocessor is also being followed for the software development. A simple flexible method of system configuration is defined. This system has the ability to automatically select the appropriate units, either system microcode or system tables, from a prestored library of such units. This technique permits using only those units necessary to perform all the desired system functions for a particular set of tasks. System functions are requested as needed from a Parts List. The appropriate units are selected from the list to
form a customized system from a set of standard building blocks.

A primary design criteria of the operating system (Manager) are flexibility, simplicity and reliability. The operating system must be flexible enough for all classes of problems to fit within its structure. All tasks must be able to access all facilities provided by the system with a given configuration of the hardware. Simplicity is necessary to allow the system to be maintained, changed, tested and initially coded with relative ease. The system architecture must be sufficiently straightforward so that the users can easily comprehend its structure and operation. The manager must also be reliable and the reCoding of a single unit should not affect the rest of the system.

To achieve the above criteria a modular, distributed system manager has been defined. Each unit is a separate group of microinstructions designed to run in a multiprogrammed mix and must be validated as a standalone program. Extreme attention must be paid to modularity, allowing new units to be plugged in without causing bugs in the rest of the system. Just as the multiprocessor's hardware flexibility lies in its easily changeable hardware configuration so should the multiprocessor's software flexibility lie in its easily changeable software configuration.

The system is composed of a single structure, and a set of conventions for using this structure. Although systems will differ, both in type of units and hardware, the set of conventions and the general structure will always be used in the same way. The specific units needed for the manager functions will be plugged into the structure as needed.

The three segments defining the operating system are the Locator, the Parts List and the units. The units are the function modules of the system which do the desired operations for the users. This collection of modules is both programs and data (tables) normally thought of as

![Figure 13—System manager](https://www.computerhistory.org)
the operating system. The tasks may select a unit through the Locator. The Locator is a part of all tasks and can access all units with a parameter called the unit number which locates the required unit using the Parts List. The Parts List is a table containing the location of all units associated with a system. Each time a new system is required, a new Parts List and a new set of units are developed. However, the Locator remains the same for all systems independent of the kind of hardware or software in use (Figure 13).

The traffic flow within a single processor is shown in Figure 14. Each task will require the use of the system units to perform some known utilities for them. This includes functions like memory allocation, I/O formatting, conversions, editing, debugging, resource allocation, channel checking, etc. The utility units perform a function and then return to the caller.

Interrupts are soft in a Multi-Interpreter System. Testing for interrupts is processed by a standard interrupt testing unit called by a task. The detection of an interrupt may suspend the caller task or it may call another unit to process the interrupt before returning to the caller. User tasks may suspend themselves or be completed by calling the Suspend Self or End units. When a program has been suspended or ended the next highest priority program ready to run is selected by the scheduler unit.

When an Interpreter task is assembled, a Source Table is developed for all of the manager hardware dependent subroutines it may use, as well as all of the functions needed by its Interpreter. This table becomes input information to the operating system Librarian (Figure 15). Each microprogrammed task has such a table developed before it can run. When the task must access the library units, it indexes into the Source Table. The Source Table defines the library unit needed. Source Tables for all tasks are used as inputs to the Operating System Librarian. This information is used with a catalog of units to develop the Parts List for the system.

Each Source Table is modified to become a list of pointers to the Parts List (Figure 16) and to the allocation portion of the Locator in microprogram memory. As each unit is referenced during a task run, the Source Table addresses the allocation part of the Locator. Space is allocated in microprogram memory. The address pointing to the allocation routine is changed to point to the desired unit now transferred into microprogram memory. The unit can now be executed. Unless this unit is deallocated from microprogram memory, the Source Table now can point directly to the unit address in this memory.

To locate a unit for allocation in microprogram memory, the allocator uses the pointer in the Source Table to locate the pointer in the Parts List which points to the unit. When the unit is present in main memory it need only be copied into the required location in microprogram memory. When the unit is not present it must
be read into memory from a peripheral device as defined in the Parts List. This device will be defined by the file directory (Figure 17) so that units may be accessed through a hierarchy of storage in a recursive manner.

Since all units of code in the Parts List are reentrant, all processors may have the same code at the same time without interfering with one another (provided they use different work areas). The work area is unique for each task and within each work area is a unique Source Table. Figure 18 shows the distributed local operating system with many Interpreters. Each Interpreter is: executing a different task, working in a unique work area, using a unique Source Table. All Interpreters share the same Parts List and have identical Locator sections. Any one unit can be unique to a task or shared by many tasks.

Since all system tasks are independent of each other, the parts that bind the system together are the system tables. These contain the important interfaces of the system and the means of communication between Interpreters and/or processes. The system tables define the three main attributes of the system: the tasks to run, the resources available, and the Interpreters running.

A Task Table is used for scheduling and termination of tasks at the global level. All information pertaining to the state, selection criteria, needs, and scheduling of a task will be found in this table. Intertask communication and task Interpreter communication is performed in the Task Table.

The Resource Table provides the information for allocating system resources among the different tasks by the various units.

An Interpreter Table contains information pertaining to each individual Interpreter. The Interpreters check on each other for the detection of malfunctioning hardware, as well as inter-Interpreter communication via this Table. An Interpreter may also use its table entry as temporary storage or for storing information about future tasks (e.g., an Interpreter may have to run routine diagnostics at the completion of its present task).
I/O system units would become descriptor building units or communication units instead of direct execution I/O units. In some systems, tasks would do their own I/O by having direct execution Parts List units while other tasks in the same system might assume an I/O module and pass descriptors to an I/O Task using other units.

Those tasks of an operating system which stand alone (e.g., I/O processor, system loaders, external communicators, etc.) are called and entered in the task table as if they were an application task with a high priority. Their ready to run bits can be set and reset by the system units.

The structure of the building block type operating system is a systematic way of constructing a manager so that its functions can be applied with identical units at all levels in a hierarchy of operating systems (e.g., course scheduling can be done at a global level while finer scheduling can be used at the next level). An Interpreter may use a unit to select a system for emulation, the task being that system's Operating System. Using the resources assigned to it at the global level, this Interpreter may choose to run its own set of tasks (which look like data to the global system). The Operating System of an emulation may be totally in the "S" language being interpreted or it may be running global units from the Parts List on its own resources assigned to it at the global level.

Figure 19 shows the control levels an Interpreter goes through to execute a user task. Interpreter 1 has been
given resources and is scheduled at level 1 to run a B 3500 emulation. The initial task to be run is the operating system for the B 3500 called the lVICP. It in turn receives data about the tasks to be run on a B 3500. The MCP sets up the tasks and executes them in a multiprogramming environment. Figure 19 shows task 2 as running. Interpreter 1 is now running a B 3500 program and is a B 3500 system which is totally independent of the rest of the system except for reporting status or requesting more resources.

Interpreters 2 and 3 have both been scheduled to become a B 5500 system. These Interpreters now have become a multiprocessing B 5500 system running the MCP and the B 5500 tasks. Interpreter 3 is running task 1 and Interpreter 2 is running task 3 of the B 5500 system’s tasks in the B 5500 schedule.

Interpreter 4 has selected a task oriented emulation. This is a microprogram specifically oriented to doing a specific job. It needs no operating system but uses the units available to it in the Parts List for executing common functions or manager type functions.

A task must decide how critical its need is for quick access to a system unit. It may decide a unit is important and once it is located and stored in microprogram memory it is left there for the duration of the task. A task which is required to process real time interrupts may want the interrupt testing unit to be a part of its code and not have to indirectly address the unit via its Source Table. Such a task may directly assemble the interrupt unit from the library into its microprogram memory. Other units which are less critical will be brought from main memory each time they are accessed.

CONCLUSIONS

Microprogrammed systems in the past have been relatively simple and primarily suitable for small, dedicated tasks. A technique has been needed to interconnect many small microprogrammed processors into one system, and to control this array of processors so that it can dynamically and efficiently share a large load. This paper has presented such a technique. The Switch Interlock allows an array of Interpreters to be integrated into a unified system with many memories and peripheral devices. The type of software described here provides a means for controlling this unified system, and allows control programs to be "custom-tailored" to each application, providing maximum efficiency. Thus the flexibility of microprogramming can now be applied to large scale systems, and virtually any size system may be constructed with a smooth evolution from a small system to a large one. A degree of reliability and of simplicity in logistics and maintenance not previously possible in medium or large scale systems is provided due to repeated use of a small number of relatively simple module types.

Microprogrammed multiprocessing systems thus appear well suited for a wide variety of data processing applications.

REFERENCES

1 R L DAVIS S ZUCKER
Structure of a multiprocessor using microgrammable building blocks
Proc of National Aerospace Electronics Conference Dayton Ohio pp 186-200 May 1971

2 E W REIGEL D A FISHER U FABER
The interpreter—A microprogrammable processor
AFIPS Conf Proc (SJCC) vol 40 Montvale NJ AFIP Press 1972

3 W A CURTIN
Multiple computer systems

4 R C LARKIN
A minicomputer multiprocessing system

5 Seek flex preliminary design study, volume 1: System design and rationale
Hughes Aircraft Co Ground System Group Report FR71-10-430 July 23 1971

6 J D MENG
A serial input/output scheme for small computers
Computer Design vol 9 no 3 pp 71-75 March 1970

7 E W Dijkstra
The structure of 'THE' multiprocessing system
Communication of ACM Vol 11 N 5 pp 341-346 May 1968

8 B J HUBERMAN
Principles of operation of the venus microprogram
Mitre Technical Report 1843 May 1 1970