Chris Day

CDF-326

July 13, 1985

Buffer Manager Software Design

Abstract:

The work that the Buffer Manager must do to control the movements of events through the data acquisition pipelines is analyzed. Based on this analysis, a conceptual design for the Buffer Manager is developed step-by-step by gradually adding requirements. As the conceptual design advances, software elements are described which implement the design. The result is a description of the single partition version of the Buffer Manager which is easily extensible to the multiple partition case.

Problem Statement:

Section 3 of CDF-183, "Dataflow within the CDF Data Acquisition System", gives the fundamental definition of the Buffer Manager's task and is quoted below.

> The Buffer Manager is the Master Intelligence controlling the flow of data through this Pipeline. It has responsibility for scheduling the processing elements at each stage (hereafter known as Pipeline Stage Elements or PSEs) and maintaining tables of statistics for each. In order to provide flexibility in configuring the system, PSEs may be dynamically added to or removed from the pipeline such that the availability of another Event Builder, for example, may be quickly and painlessly utilised. A further requirement is that the dataflow should be independent of whether part (or all) of the pipeline is implemented in FASTBUS hardware or in software on a VAX. Thus the functionality of the various PSEs and the protocols by which they communicate with the Buffer Manager should be decoupled as much as possible from their implementation and the transmission medium via which they communicate. Similarly the Buffer Manager itself may initially be implemented as a process (or processes) on a VAX. Other than limiting the number of VAXs that may be associated with a partition, this software or hardware implementation should be transparent.

It is assumed that the reader is familiar with the remainder of that document. The rest of this document describes the portion of the Buffer Manager which manages the dataflow; the statistical section is discussed elsewhere.

The reader should be warned that the names of the messages sent and received by the Buffer Manager have been changed in this document relative to CDF-183 to make their function less ambiguous. However, the design of the dataflow software has very little to do with the content and meaning of the messages sent. Adapting to the new names should not be too difficult.

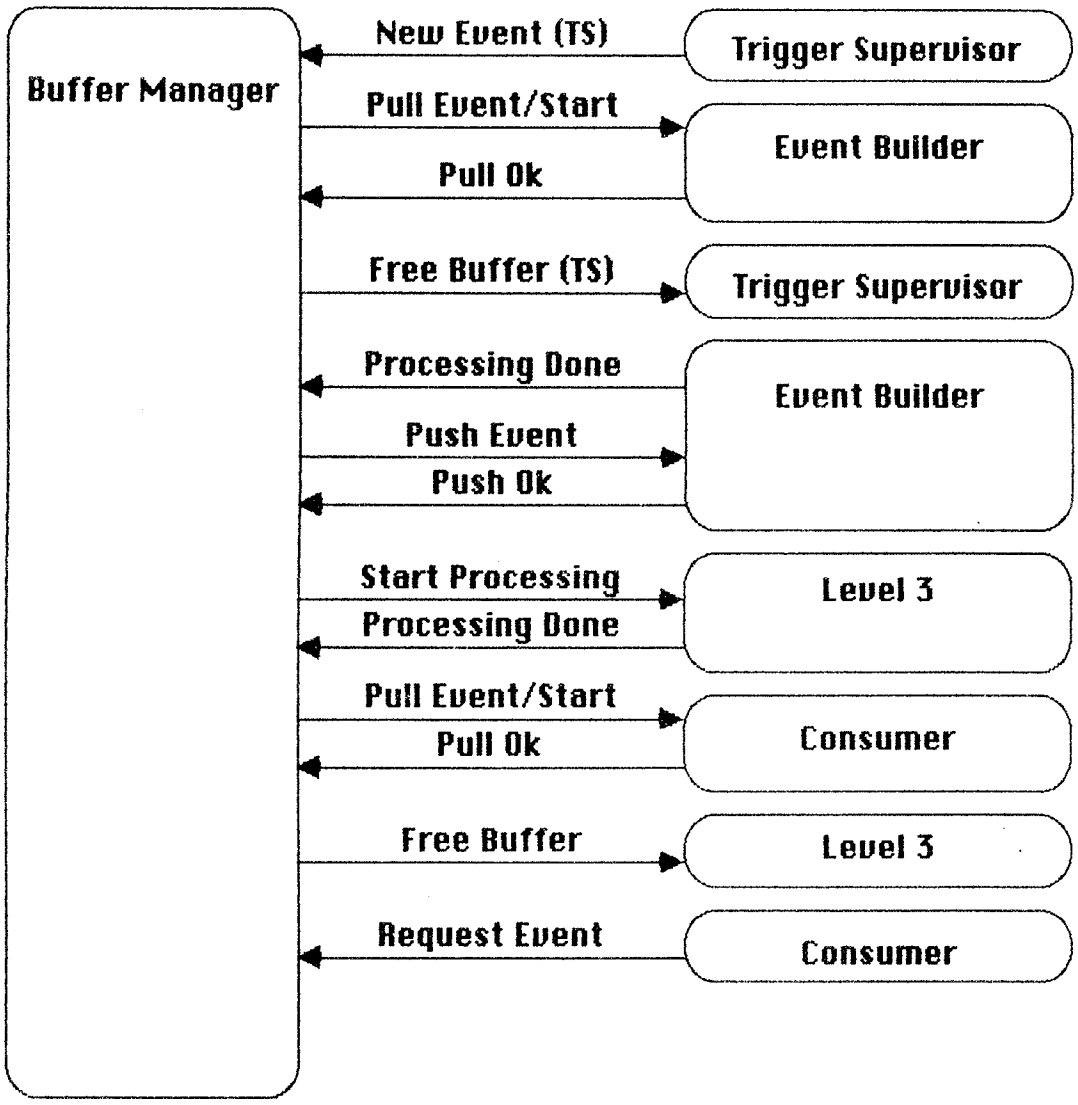
CDF-183 describes the dataflow control messages seen by each PSE. This is useful in the design of an individual PSE but does not make very clear what happens from the Buffer Manager's point of view. Figure 1 shows all of the dataflow messages sent and received by the Buffer Manager to move a single event through a pipeline. The pipeline in this case has four stages, Trigger Supervisor, Event Builder, Level 3 and Consumer Computer, and the event is assumed to pass one of the trigger mask requirements. Time flow is from top to bottom of the figure.

This is a reasonably complex sequence of messages, but what makes the Buffer Manager's life really complicated is that the sequence is not fixed. PSEs can be added or subtracted from the pipeline dynamically. For example, Level 3 might develop a problem and be taken out of the system. In response, the Buffer Manager must stop dealing with Level 3 messages and route the data from the Event Builder to the Consumer Computers directly; none of the other PSEs need be aware that anything has happened. Also, Figure 1 represents the messages required to move only one event. The pipeline will actually have multiple events in it at various PSEs. The sequence of messages for any given event may look like Figure 1, but the sequence as seen by the Buffer Manager will be an interlaced set from all of the events. Furthermore the Buffer Manager may be running several pipelines at once, each with a different structure and each with many events. Finally, multiple pipelines may share physical elements, *e.g.* the Event Builder. In this case the Buffer Manager not only has to determine to which event and to which pipeline messages from the event builder refer, but it may temporarily have to block access to the Event Builder for an event in one pipeline because all of the Event Builder's resources are occupied with events in different pipelines. It must then remember that someone has been waiting when resources are released.

An attempt to design at one stroke a solution which deals with all of these possible conditions, options, and situations is virtually certain to fail.

Figure 1

Buffer Manager/Pipeline Message Traffic
for Single Event



The problem must be broken down into its smallest manageable and meaningful parts and a full solution developed step-by-step from the pieces.

Analysis:

The simplest job that the Buffer Manager has to do is move a single event between two PSEs of a isolated pipeline. Even then there are two cases. Because data transactions on Fastbus are always between a master and a slave, one can either have the Fastbus master pushing data to the following slave (see Figure 2) or have the master pulling data from the preceding slave (see Figure 3). In either case, the Buffer Manager receives a message from the upstream PSE declaring the availability of an event. The Buffer Manager then decides on the destination of the event, sends a message to the master of the transaction requesting that the event be moved and waits for acknowledgement that the move has occurred. Finally, the Buffer Manager sends a message to the slave of the transaction reporting that the event has been moved.

This action of the Buffer Manager can be summarized by the following piece of pseudo-code:

LISTING 1:

```
REPEAT
   Wait_for_event_ready(current_PSE);
   Determine_next_link(pipeline_ID, current_PSE, pipeline_link);
   Send_start_move_message(pipeline_link.Master);
   Await_move_complete_message(pipeline_link.Master);
   Send_move_report_message(pipeline_link.Slave);
   current_PSE := pipeline_link.destination;
UNTIL doomsday;
```

This pseudo-code fragment, like the others in this document, is written is a Pascal-like language. This is because Pascal is much clearer than Fortran would be for some of the later complications and because the Buffer Manager is actually written (mostly) in Pascal.

Note that in principle a given PSE can be either master or slave. For example, some of the proposals discussed for Level 3 have that PSE act as a master to pull data from the preceding stage, but act as a slave to the following Consumer Computer. To provide maximum freedom for the

Figure 2

Master-to-Slave, Single Event Transactions

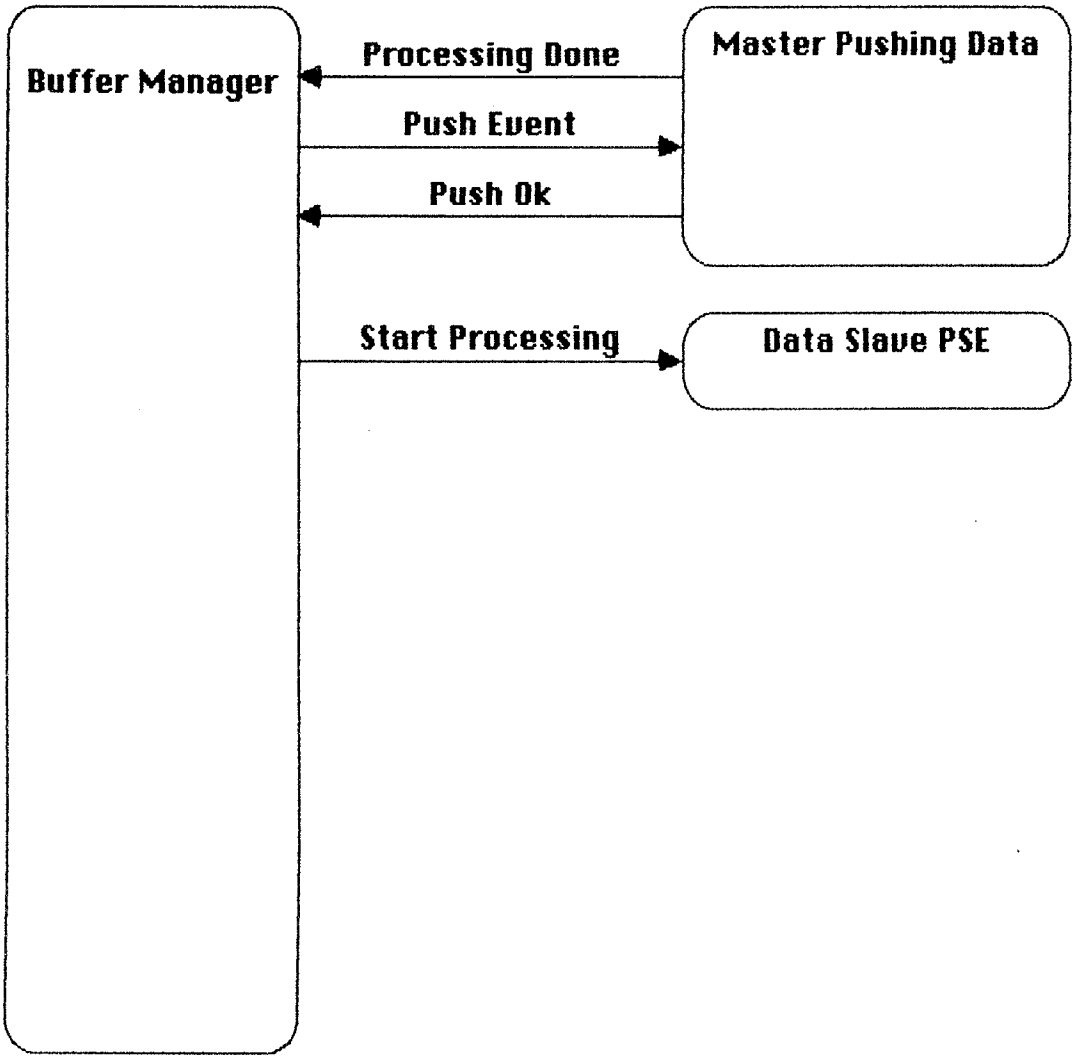
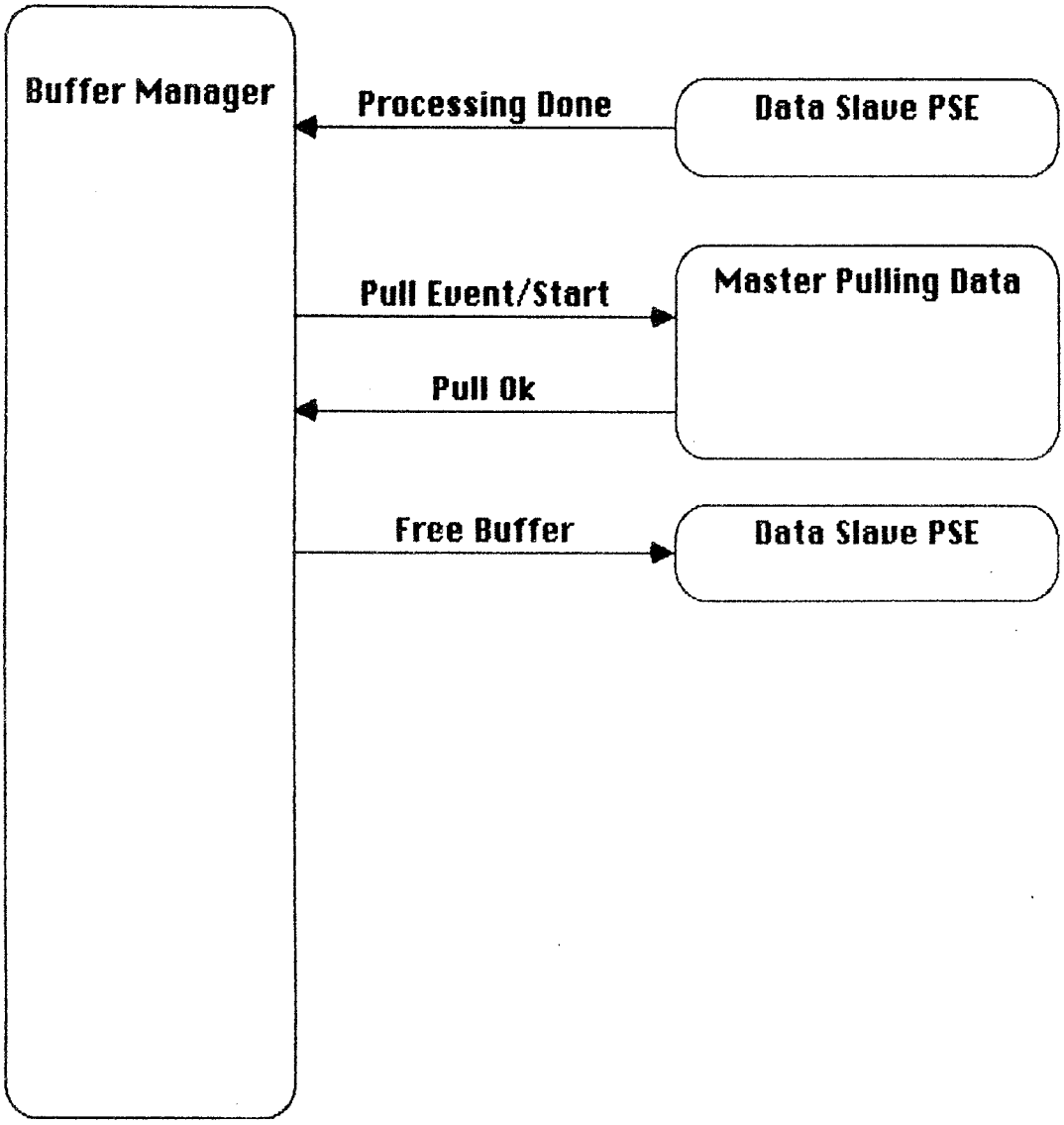


Figure 3

Slave-to-Master, Single Event Transaction



designers of the PSE hardware, it is better to base the Buffer Manager's actions on the role of a PSE in a given transaction rather than on the details of the PSE itself. For this reason, the Buffer Manager deals with the concept of a pipeline link, containing information about the roles of the participating PSEs as well as the PSEs themselves. By dealing with roles, the main loop of the pseudo-code can treat the two cases of Figures 2 & 3 identically.

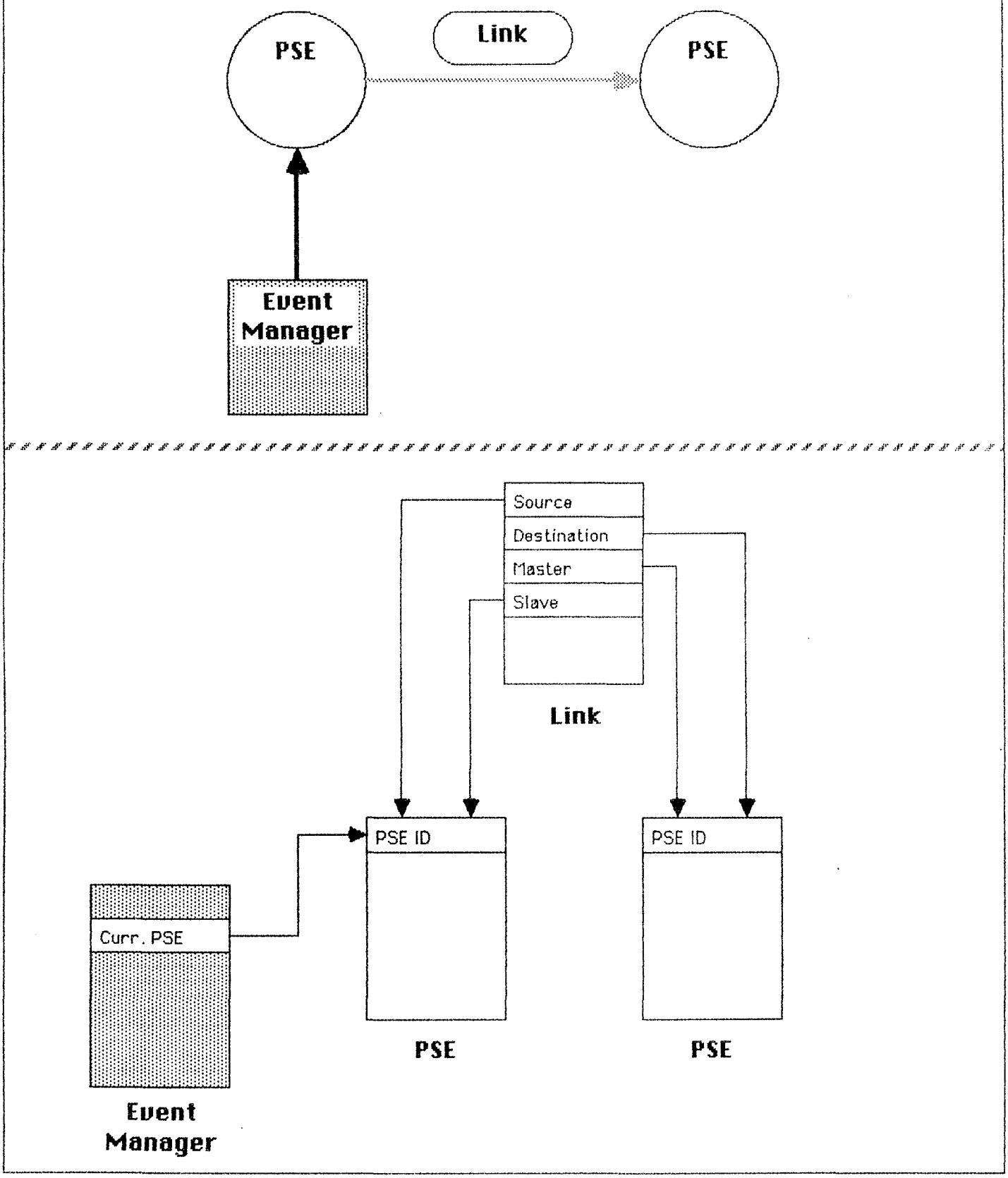
Introducing the concept of a pipeline link also allows a clean separation of the topological functions of the Buffer Manager from the dataflow functions. PSEs can be statically defined objects with characteristics reflecting the particular hardware in isolation from its actions in a pipeline. The requests to the Buffer Manager to build and modify pipelines translate into dynamic modifications of the pipeline_link structures. The only interface to the dataflow module required is the look-up procedure Determine_next_link. This procedure completely hides the implementation of the topological actions. For example, the version of the Buffer Manager for the September 1985 run has the pipeline defined statically via conditional compilation. Later versions will implement the dynamic insertion and deletion of PSEs. This change will require no modification to the dataflow section of the code whatsoever.

Step-wise design:

Single link/One event-

There are now enough concepts in hand to begin designing the system software. Figure 4 presents two views of the situation described above. The upper portion of the figure is an abstract picture of the relationships between software components, while the lower portion sketches actual software structures. In both cases, the figure represents a single pipeline link with a single event at the upstream PSE. The link and the PSEs are realized as passive data records, *i.e.,* they contain no executable code, only data. The PSE record, at this point, only needs to contain a PSE identifier. These records are created when the Buffer Manager is told of the existence of a new PSE. The Link record contains pointers to the source and destination PSEs and master/slave pointers which specify the role each PSE should play in the data transaction. These records are created and filled in when the topological section of the Buffer Manager adds a new

Figure 4


PSE
Link
PSE
Event Manager
Source
Destination
Master
Slave
Link
PSE ID
PSE ID
Curr. PSE
PSE
PSE
Event Manager

stage to a pipeline. The Event Manager is the active portion of the system. It actually executes the code of Listing 1. It keeps in its own local storage a pointer to the Current PSE, that is, the stage where the event data reside at the start of the transaction. A single pass through the code will move the data across the link and point the Event Manager to the new data location.

Multiple link/One event-

Figure 5 represents the case of a single event moving through an extended but finite pipeline. The new feature required is the proper handling of the ends of the pipeline. First, the ends must be detectable, hence the added Boolean fields to the Link records -- Terminal_link and Source_link. Second, the Event Manager must behave appropriately at the ends. If the Event Manager's job is taken to be to shepherd a single event through its pipeline, and if the Event Manager starts with its Current PSE pointer directed to the initial PSE, then the Event Manager can simple repeat the loop of Listing 1 until it reaches the last PSE and then terminate itself. The pseudo-code becomes:

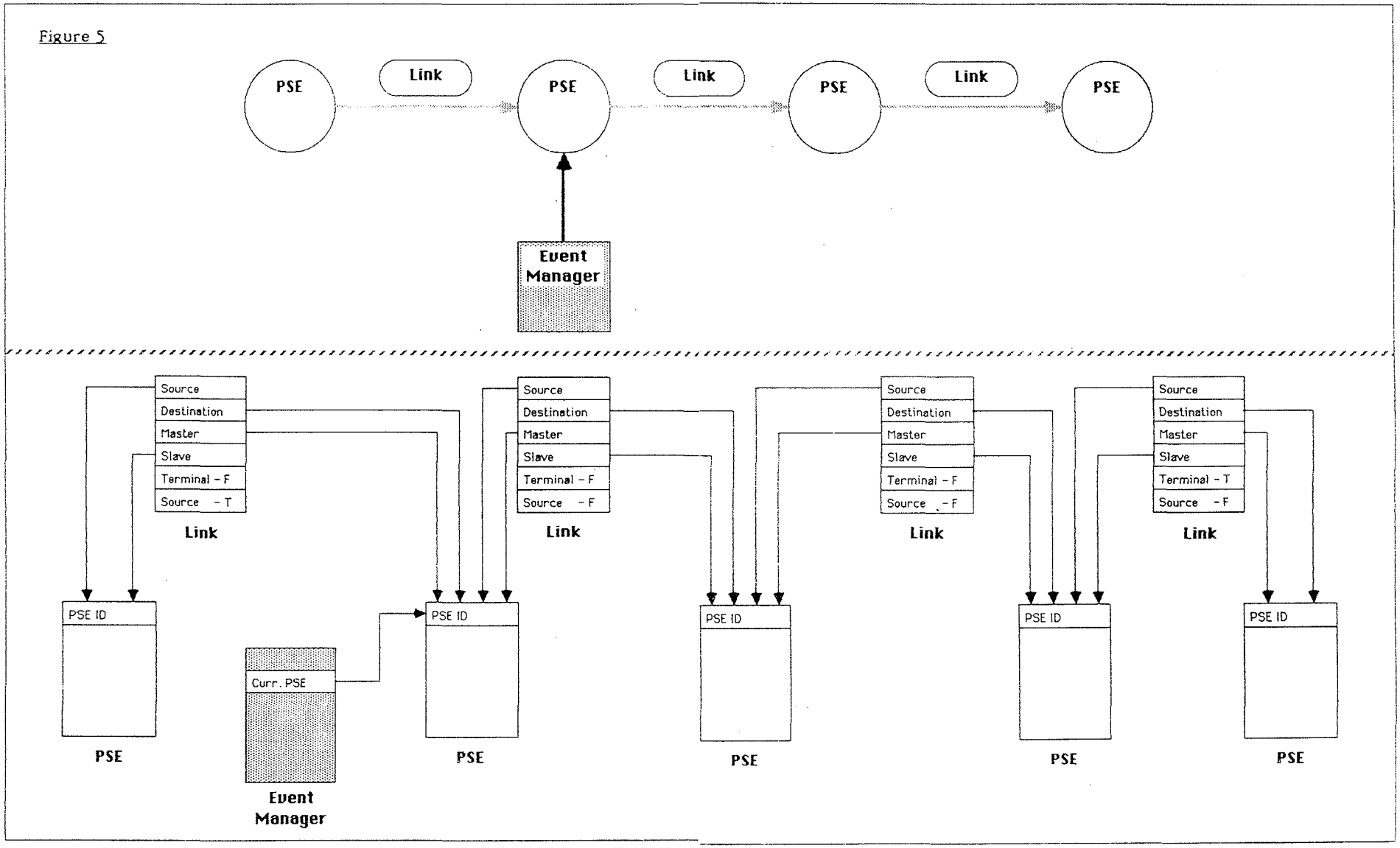
LISTING 2:

```
REPEAT
   Wait_for_event_ready(current_PSE);
   Determine_next_link(pipeline_ID, current_PSE, pipeline_link);
   Send_start_move_message(pipeline_link.Master);
   Await_move_complete_message(pipeline_link.Master);
   Send_move_report_message(pipeline_link.Slave);
   IF pipeline_link.terminal_link THEN
      terminate_Event_manager
   ELSE
      current_PSE := pipeline_link.destination;
UNTIL doomsday;
```

Multiple disjoint pipelines/One event per pipeline-

The pseudo-code presented so far, although simple and comprehensible, can only deal with a universe containing a single event. First, there is only one Current_PSE pointer, so only one event can be tracked at a time. Second, the code expects all inbound messages to come from the PSEs of the current pipeline link and be about the current data transaction; it has no idea what to do with anything else. Third, once the event is delivered to the end of the pipeline, the Event Manager terminates and will not

Figure 5


PSE
Link
PSE
Link
PSE
Link
PSE
Event Manager
Source
Destination
Master
Slave
Terminal - F
Source - T
Link
Source
Destination
Master
Slave
Terminal - F
Source - F
Link
Source
Destination
Master
Slave
Terminal - F
Source - F
Link
Source
Destination
Master
Slave
Terminal - T
Source - F
Link
PSE ID
PSE
Curr. PSE
Event Manager
PSE ID
PSE
PSE ID
PSE
PSE ID
PSE
PSE ID
PSE

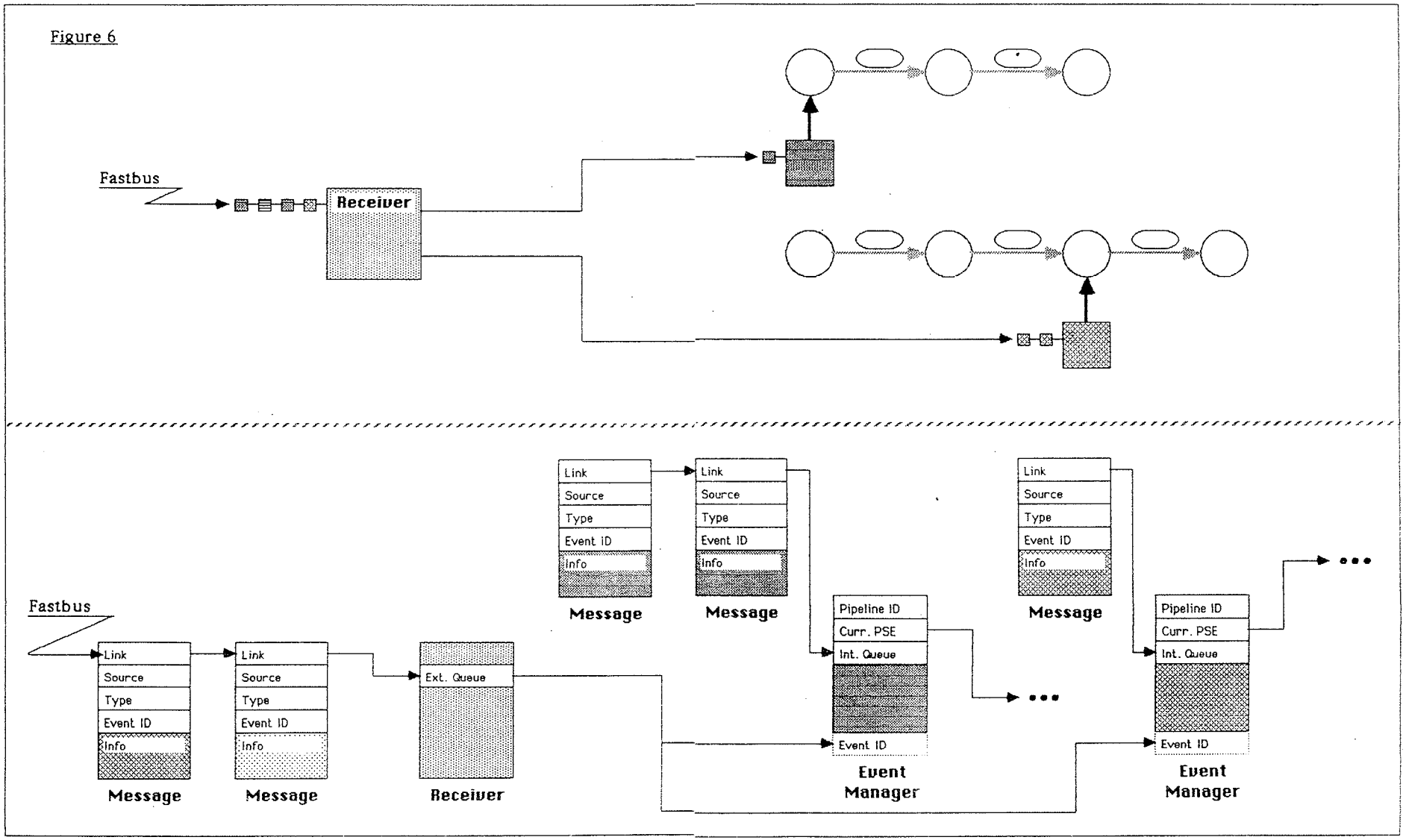
respond to any other messages. Since the Buffer Manager must deal with multiple events, the code of Listing 2 seems seriously inadequate. However, with the addition of proper underpinnings, it will in fact work in the multiple event situation.

Consider the case of two disjoint pipelines each with only one event in it. Then the code in Listing 2 works correctly provided (1) there are two Event Managers, one devoted to each event, and (2) each Event Manager only receives messages about its own event. Both of these conditions can be met in a straightforward manner by embedding Listing 2 into a multiprocess environment. We create an Event Manager process for each event and this process runs its own copy of Listing 2. In addition, we create another process which receives all messages from Fastbus and distributes them to the appropriate Event Managers based on an Event ID. Figure 6 illustrates this situation.

The Receiver process operates as follows. In the idle condition, the Receiver process waits for messages on its input queue. The Fastbus interface places incoming messages onto this queue and wakes up the Receiver process. The Receiver then examines the type field of the message. If the message type is New_event, then the Receiver creates a new Event Manager process, put the message on the Event Manager's message queue, points the new Event Manager's Current PSE at the source PSE specified in the message's Source field and makes the Event Manager ready to run. If the message type is not New_event, then there already is an Event Manager in the system for this event. The Receiver uses the Event_ID field of the message to locate the proper Event Manager, places the message on the Manager's queue and reactivates the process. In either case, the Receiver then relinquishes control of the CPU while any Event Managers are ready to run. When all activity generated as a consequence of the arrival of the Fastbus message stops, the Receiver checks its input queue for another message.

Note that the Event_ID is assigned by the Buffer Manager, not by the PSEs. The bottom section of Figure 6 shows the new data structures needed for message distribution. Messages contain Source, Type and Event_ID fields as well as an Info field. The latter field is of variable length and contains such things as the trigger mask words which can be ignored by the Receiver process. In order to link messages into queues, each message

Figure 6


Fastbus
Receiver
Fastbus
Link
Source
Type
Event ID
Info
Message
Link
Source
Type
Event ID
Info
Message
Ext. Queue
Receiver
Link
Source
Type
Event ID
Info
Message
Pipeline ID
Curr. PSE
Int. Queue
Event ID
Event Manager
Link
Source
Type
Event ID
Info
Message
Pipeline ID
Curr. PSE
Int. Queue
Event ID
Event Manager

contains a Link field which points to the next item in its queue. The Link field is added to the message after its reception; it does not exist on the Fastbus.

The Receiver process has an External Message Queue header to which the hardware delivers Fastbus messages. The Receiver also has a means of locating an Event Manager from a message's Event_ID. In the first implementation, the Event_ID is actually the virtual address of the associated Event Manager. This gives an extremely fast lookup, but may not be reliable enough, so may change in future versions.

Each Event Manager process acquires two new local variables. One is an Internal Message Queue header to which the Receiver process delivers messages. For use by the Determine_next_link procedure, the Event Managers also have a Pipeline_ID variable. With these adjustments, each Event Manager can run the pseudo-code of Listing 2 with no problems, thereby giving the Buffer Manager as a whole a multiple event capability. Note that there is no need to worry about initialization of the loop in Listing 2; the Receiver process will set up each Event Manager properly. Also, there is no problem with the Event Manager terminating when it has delivered its event to the end of the pipeline; another Event Manager will be created to handle the next event.

The principle concern with the multiprocessing approach is the time spent switching contexts, particularly since Buffer Manager response is vital to the operation of the whole Data Acquisition System. If each Event Manager were a separate VMS process, the system could not possibly be fast enough since each context switch would require about 1 msec. on a µVAX II. Even using the faster kernel of VAXEln would take about 300 µsec. On the other hand, both of these systems were designed to deal with a much more complex multiprocessing environment than is needed for the Event Managers. For example, both systems are event driven, preemptive, priority based systems. Determining which process should run next can be complicated in such systems. For the Buffer Manager, however, all the processes can have the same priority and each process can use the CPU until it must wait for a message or other resource, *i.e.,* preemption due to external events is not required. Under these conditions, the multiprocessing system becomes very simple. In fact the "processes" become coroutines. (See, for example, Knuth, Fundamental Algorithms,

Section 1.4.2, "Coroutines", or Wirth, Programming in Modula-2, Section 30, "Concurrent processes and coroutines". Note that exactly the system needed here is provided as a language feature in Modula-2.) As implemented, the context switch consists of a CALLS instruction with two parameters and all registers saved, a swap of Frame Pointers, and a return. The procedure takes about 20 µsec.

With the context switch reduced to such a short time, the next level of software overhead becomes a concern. A well layered system is highly desirable to assure reliable synchronization, but it can lead to the invocation of many procedures between a high level call, like Wait_for_event_ready, and the actual context switch. If each invocation uses the relatively slow CALLS instruction, the effective context switch time can add up to unacceptable levels. However, one of the features of the EPascal compiler is its ability to implement designated Procedures as inline code rather than as actual subroutines. This not only removes the CALLS instructions, but also allows the optimization modules of the compiler to work through procedure invocations. With only a preliminary effort at optimization using this technique, the Buffer Manager software can receive a message, activate the proper Event Managers, run the proper resource synchronization checks, build and send the appropriate reply message, all in about 1 msec.

One other overhead that could be a problem is the creation and termination of Event Manager processes. This has been reduced by not actually destroying terminated Event Managers but placing them in a pool of unused processes. When a process is to be created, the Process Manager module first tries to reuse a process from the pool. Only if the pool is empty does the Process Manager actually create a new process structure via the memory management services. The pool is initially empty and no processes are ever destroyed; the number of processes in the Buffer Manager rapidly adjusts itself to a level sufficient for the peak demand seen by the system.

Multiple pipelines sharing PSEs/Multiple events per pipeline-

While the situation of Figure 6 is a legitimate multiple event case, it was contrived specifically to avoid two events trying to access the same PSE at the same time. Such collisions can arise if a PSE, *e.g.,* the Event

Builder, is shared between pipelines. A collision can also happen if an event in a pipeline catches up to a previous event in the same pipeline. In either circumstance, the colliding events must coordinate their access to the PSE in dispute, otherwise events can be lost. For example, the Event Builder may be able to handle only one event at a time. If the Event Builder were reformatting one event and was ordered to read out another, the first event could be lost and the pipeline flow seriously confused. To avoid this, the Buffer Manager should delay sending the readout request until the previous event has been moved to the next PSE.

There are two major design problems created by the advent of collisions. One is to decide what resource management discipline to use for the PSEs. The Event Builder might be able to deal with only one event at a time. On the other hand, it might be able simultaneously to read one event from the front end, format a second event, and send a third to Level 3. Then the decision to delay a request for service depends in a complex way on the current state of the Event Builder. Level 3 will be able to service many events at once, but the internal destination of the next event has to be maintained by the Buffer Manager. Since none of the hardware PSEs has yet been fully designed, and each is likely to go through several incarnations, no fixed resource management discipline can be built into the Buffer Manager; anything should be possible.

The second problem is how to implement the synchronization. Sharing resources among multiple processes is the fundamental problem of operating systems. Early operating systems tended to have a massive, monolithic kernel which managed everything. Such kernels are nearly impossible to code correctly and are far too inflexible to handle the dynamic creation of PSEs and their insertion and removal from pipelines. A more modern approach is to control resources in a distributed fashion through the use of monitors. (See, for example, Brinch Hansen, The Architecture of Concurrent Programs, Section 4.3 "Monitors", or Bowen and Buhr, The Logical Design of Multiple-Microprocessor Systems, Chapter 3, "Monitors".) A monitor is a collection of procedures for manipulating a resource along with a mechanism for assuring the orderly execution of those procedures by multiple processes.

In the following discussion, it is assumed that the reader is familiar with semaphores as software synchronization constructs. (See, for example,

Bowen and Buhr, The Logical Design of Multiple-Microprocessor Systems, Section 1.2.3, "Critical Regions -- Semaphores", or VAXELN Programming, DEC Document Order Number: AA-Z451A-TE, Version 1.0, pp 1-29 ff. VAXELN semaphores are not used for speed reasons. They have been reimplemented in the coroutine context, but their abstract behavior remains the same.)

There are, in fact, two synchronization problems when trying to share resources. One is synchronizing access to the resource and the other is synchronizing access to the controlling software. A generic monitor provides both types of control by providing two data types and four procedures which act on variables of those types. (The Buffer Manager uses monitors of type gladiator. See Bowen and Buhr, The Logical Design of Multiple-Microprocessor Systems, Section 3.4.2, "Monitor of Type Gladiator" and Section 3.3.4, "Gladiator".) The two data types are the condition_variable, which is basically a process queue, and the this_monitor, which is a record containing a semaphore called the monitor gate and a condition_variable called the eligible queue. A given monitor may have any number of condition_variables, but must have one and only one this_monitor. The four procedures are Enter and Leave, which take a variable of type this_monitor as argument, and Sleep and Awaken, which take a variable of type condition_variable and a variable of type this_monitor as arguments.

Each procedure of a specific monitor begins with the statement Enter(Here) and ends with the statement Leave(Here) where Here is the this_monitor variable which defines the monitor is question. The Enter routine causes the calling process to wait on the gate semaphore. If no other process is actively executing in one of the procedures of the monitor, the gate will be open and the calling process will proceed. The Leave routine checks to see if there are any processes waiting in the eligible queue. (As we shall see in a moment, processes in the eligible queue are ones which have been blocked in the monitor waiting on a condition_variable.) If so, it makes the first one ready to run. Otherwise, it signals the gate semaphore and proceeds to its asynchronous duties. The actions of Enter and Leave on the semaphore of Here make the procedures of the monitor collectively into a conventional critical region. This is the mechanism by which a monitor solves the problem of controlling access to the resource synchronization code; no more than one

process can be actively executing in the monitor at one time.

Sleep and Awaken, along with the action of Leave on the eligible queue, provide the means by which resources can be controlled. Their behavior is best explained by a simple example.

LISTING 3:

```
MODULE producer_consumer;
   EXPORT put_work, get_work;
   INCLUDE gladiator, buffer_definition;

   VAR
      here: this_monitor;
      waiting_for_work, waiting_for_empties: condition_variable;
      work, empties: buffer_queue;

   PROCEDURE put_work(VAR buff: buffer_pointer);
   BEGIN
      enter(here);
      insert_buffer(buff, work);
      awaken(waiting_for_work, here);
      IF empties.empty THEN
         sleep(waiting_for_empties, here);
      buff:=remove_buffer(empties);
      leave(here);
   END {put_work};

   PROCEDURE get_work(VAR buff: buffer_pointer);
   BEGIN
      enter(here);
      insert_buffer(buff, empties);
      awaken(waiting_for_empties, here);
      IF work.empty THEN
         sleep(waiting_for_work, here);
      buff:=remove_buffer(work);
      leave(here);
   END {get_work};
END {producer_consumer}.
```

Listing 3 gives the solution to the classic producer-consumer problem in terms of calls to a monitor. The monitor consists of the two routines put_work, used by producers, and get_work, used by consumers. The routines are symmetrical, so the discussion will concentrate on put_work. Once the calling producer returns from Enter, it has exclusive access to the monitor. It deposits its full buffer on the work buffer queue and calls Awaken with the waiting_for_work condition variable. If there are no consumers waiting in the condition variable's process queue, the routine does nothing and returns. If there are processes waiting, Awaken takes

the first waiting process, moves it to the eligible queue of Here and then returns. In either case, the producer continues on in sole control of the monitor. The producer then checks the empty buffer queue. If a buffer is available, the producer removes it and leaves the monitor. If a buffer is not available, the producer cannot continue its functions; it calls Sleep with the waiting_for_empties condition variable. Sleep checks the eligible queue and makes the first process, if any, ready to run. If there are no eligible processes, Sleep makes the producer signal the monitor gate, thereby releasing its exclusive access to the monitor. In either case, Sleep then causes the producer to place itself on the waiting_for_empties process queue and generates a context switch to the first process on the ready to run queue. Note that if there had been a consumer waiting on the waiting_for_work condition variable, it was moved to the monitor-wide eligible queue and would be made ready to run by the producer either Sleeping or leaving the monitor.

Assume that the producer-consumer system contains only one producer, one consumer, and one buffer and that the producer originally has the buffer. Then the consumer will be waiting in the waiting_for_work queue, by virtue of the call to Sleep in get_work, and will be made eligible by the producer's call to Awaken. When the producer Sleeps in turn, the consumer will be reactivated and return from its call to Sleep *in get_work!* A work buffer is then guaranteed to be available (the producer just put it there and the monitor gate has not been opened, so no other process could have taken it away); the consumer removes it and leaves the monitor. Since there are no processes eligible (the producer is waiting on the waiting_for_empties queue) the exit of the consumer opens the monitor gate. Therefore, once the consumer has transformed the buffer into an empty one, it is able immediately to pass through the Enter call of get_work and Awaken the producer. The producer and consumer then continue to trade the buffer back and forth in a controlled fashion.

The above discussion uses single buffers, consumers, and producers for ease of discourse; the system works equally securely with any positive numbers of processes and buffers.

One point in particular is worth noting about monitors and the code of Listing 3. It is the fact that specific condition variables and their associated this_monitor variable together make up the heart of monitor.

It would be a catastrophe if one were to call Awake or Sleep with a condition variable from one monitor and a this_monitor variable from another monitor. The EXPORT controls of EPascal prevent this from happening. Since only the monitor routines, put_work and get_work, are exported and not the crucial variables, the compiler will ensure that no other portion of the system can access and misuse them, while still allowing the variables to be shared among the procedures within the scope of the monitor code itself.

Condition variables allow the implementation of a wide variety of resource management schemes. However, in the absence of any particulars about the resources contained in the PSEs, some skeletal form of resource needs to be invented so that the monitor interface can be defined and the software design proceed. An abstraction which appears adequate to the task is the idea of PSE Ports. Each PSE is assumed to have an unspecified number of input data Ports and an unspecified number of output data Ports. Before a data transaction can take place across a pipeline link, the Event Manager must acquire an output Port from the upstream PSE and an input Port from the downstream PSE. If a Port is not available, the Event Manager is blocked and the data transaction is delayed. Once the data are moved, the Ports must be released by the Event Manager. Each PSE monitor then needs two routines, one for the Event Manager to acquire a port and one to release it; the details of when and why Ports are available can be hidden within the specific monitor for a given type of PSE.

There remains a problem. Although the use of abstract Ports as resources to be controlled allows the definition of a fixed argument list for the Event Manager to use when accessing a PSE's monitor, the particular routine, and therefore the particular monitor, to call depends on the PSE. One has to have a release_TS routine, a release_EB routine, &c. If these routines are presented directly to the Event Manager, then it has to actively decide which routine to call. This makes the Event Manager much more cluttered and complex, and virtually assures that one day the Event Manager will call release_EB when it should have called release_L3 with disasterous results.

Fortunately, EPascal provides a way to deal with this difficulty. Each PSE record contains two additional fields. One is a pointer to that PSE's acquire_port routine and the other points to the proper release_port

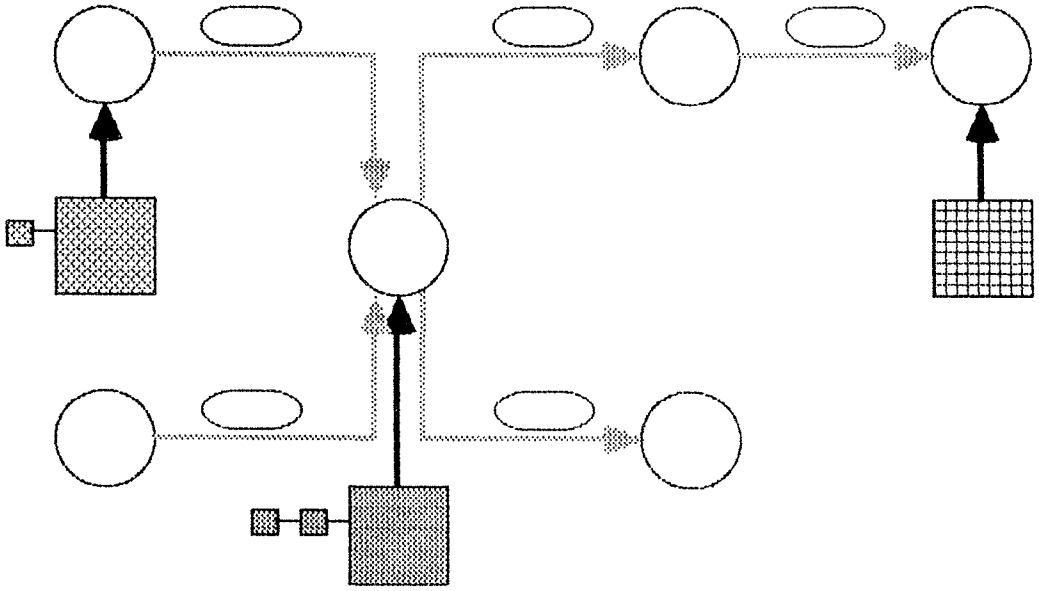
routine. The Event Manager code then can indirectly invoke the routine pointed to by the standard PSE field without knowing what that routine actually is. By making the creation of the PSE data block a function of the monitor Module, one can easily assure that the right routine is associated with the given PSE. Futhermore, by exporting only the PSE creation function, no other component of the system can locate the port routines and call them inappropriately.

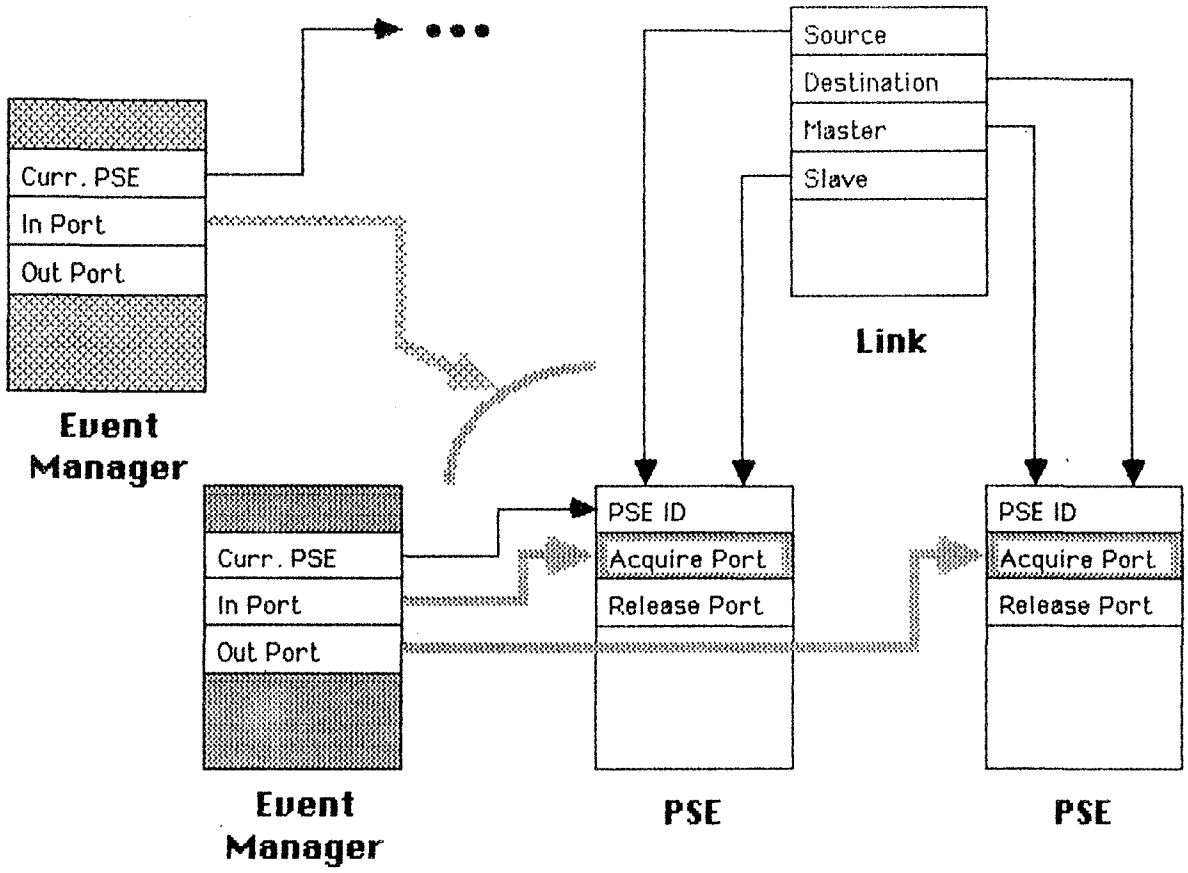
At this point, collisions are firmly under control. Figure 7 illustrates the type of situations that have been under discussion and the additions to the data structures required. Note that each Event Manager has acquired fields to hold Port identifiers so that when the Ports are released, the monitors known which resource is being returned. For the reader interested in how this whole discussion of monitors translates into code, Listing 4 gives the pipeline_definition Module, which defines the generic aspects if a PSE, and the EB_Monitor Module, which is a specific implementation of a PSE monitor.

LISTING 4:

```
MODULE pipeline_definition;
TYPE
   PSE_ID = (DAQ_Buffer_Manager, DAQ_Trigger_Supervisor,
             DAQ_Event_Builder, DAQ_Level_3,
             DAQ_Consumer_Computer);

   direction = (Ingress, Egress);
   pipeline_link = RECORD
      source:        ^PSE_control_block;
      destination:   ^PSE_control_block;
      master:        PSE_ID;
      slave:         PSE_ID;
      terminal_link: BOOLEAN;
      source_link:   BOOLEAN;
      END;

   PSE_control_block = RECORD
         PSE:          PSE_ID;
         acquire_port: ^ANYTYPE;
         release_port: ^ANYTYPE;
         END;

   FUNCTION create_PSE: ^PSE_control_block;
   FUNCTION_TYPE;

   PROCEDURE acquire(port:INTEGER; IO:direction; this:^PSE_control_block);
   PROCEDURE_TYPE;

   PROCEDURE release(port:INTEGER; this:^PSE_control_block);
```

Figure 7




Curr. PSE
In Port
Out Port
Event Manager
Source
Destination
Master
Slave
Link
Curr. PSE
In Port
Out Port
Event Manager
PSE ID
Acquire Port
Release Port
PSE
PSE ID
Acquire Port
Release Port
PSE

```
    PROCEDURE_TYPE;

END {pipeline_definition}.


MODULE EB_Monitor;
   EXPORT create_EB;
   INCLUDE gladiator;
   INCLUDE pipeline_definition;

   VAR
      here: this_monitor;
      event_builder_busy: BOOLEAN;
      waiting_for_event_builder: condition_variable;

   FUNCTION create_EB OF TYPE create_PSE;
   VAR new_EB: ^PSE_control_block;
   BEGIN
      NEW(new_PSE);
      WITH new_PSE@ DO
      BEGIN
         PSE:=DAQ_Event_Builder;
         acquire_port:=Address(acquire_EB);
         release_port:=Address(release_EB);
      END;
      create_EB:=new_PSE;
   END {create_PSE};

   PROCEDURE acquire_EB OF TYPE acquire;
   BEGIN
      enter(here);
         IF event_builder_busy THEN
            sleep(waiting_for_event_builder, here);
         event_builder_busy:=TRUE;
         port:=1;
      leave(here);
   END {acquire_EB};

   PROCEDURE release_EB OF TYPE release;
   BEGIN
      enter(here);
         event_builder_busy:=FALSE;
         port:=0;
         awaken(waiting_for_event_builder, here);
      leave(here);
   END {release_EB};

END {EB_Monitor}.
```

Meanwhile, what has become of the Event Manager code? Very little, in fact. Listing 5 gives the new Event Manager loop. Save for the addition of the four INVOKE statements, the code is effectively unchaged from the no-collision case. Because of the indirect access to the Port routines and the power of the monitor concept, this code will work regardless of how

the hardware PSEs are ultimately implemented.

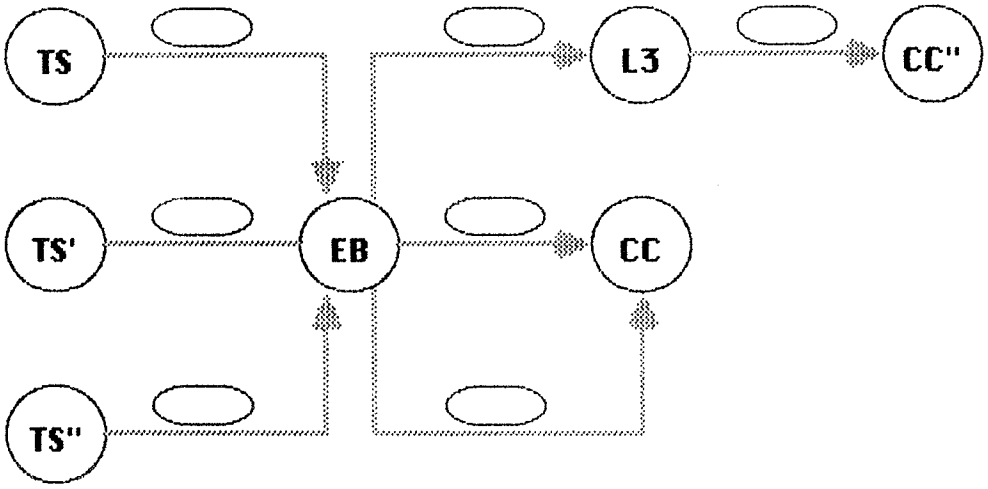
LISTING 5:

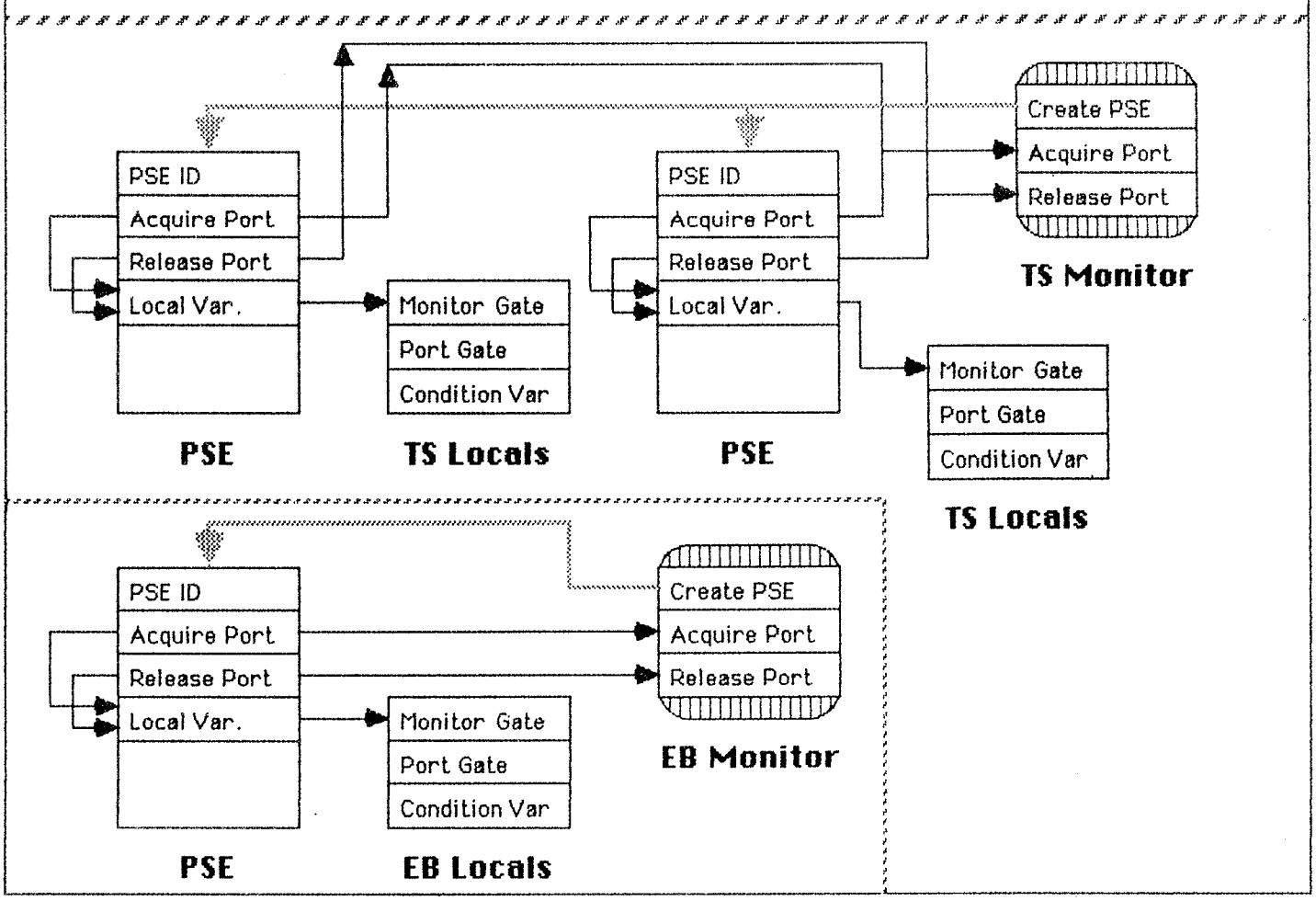
```
REPEAT
   Wait_for_event_ready(current_PSE);
   Determine_next_link(pipeline_ID, current_PSE, pipeline_link);
   WITH pipeline_link@ DO
   BEGIN
      INVOKE(source@.acquire_port, acquire, Out_port, EGRESS, source);
      INVOKE(destination@.acquire_port, acquire, In_port, INGRESS,
            destination);
      Send_start_move_message(Master);
      Await_move_complete_message(Master);
      INVOKE(source@.release_port, release, Out_port, source);
      INVOKE(destination@.release_port, release, In_port, destination);
      Send_move_report_message(Slave);
      IF terminal_link THEN
         terminate_Event_manager
      ELSE
         current_PSE := destination@.PSE;
   END {WITH};
UNTIL doomsday;
```

Cloned PSEs-

There remains one topic to be discussed. In the final multiple partition case, there will be a substantial number of PSEs but a smaller number of PSE types. (See the upper portion of Figure 8.) The resource management characteristics of a given PSE are determined by its type, so it would be useful to be able to reuse the code for a given monitor type for each instance of the corresponding PSE. Unfortunately, a monitor by its very nature must include local variables, the this_monitor variable and the condition variables at least, which survive across invocations of the monitor's procedures. In the examples of Listings 3 & 4, these variables have been declared as static variables permanently within the scope of the monitor Module itself. Since there is therefore a one-to-one match between monitor Modules and local variables, there must be a separate Module for each PSE rather than each PSE type. However, a little prestidigitation will get us around this limitation.
First, instead of declaring the condition variables, &c. needed by a monitor Module as static variables, they are collected together to form a TYPE declaration of a local_variables record. Any variable of this type now represents a distinct instance of a monitor of the type specified by the monitor Module. Note that the content of the local_variables records will

Figure 8


TS
L3
CC"
TS'
EB
CC
TS"


Create PSE
Acquire Port
Release Port
TS Monitor
PSE ID
Acquire Port
Release Port
Local Var.
Monitor Gate
Port Gate
Condition Var
PSE
TS Locals
PSE ID
Acquire Port
Release Port
Local Var.
Monitor Gate
Port Gate
Condition Var
PSE
TS Locals
Create PSE
Acquire Port
Release Port
EB Monitor
PSE ID
Acquire Port
Release Port
Local Var.
Monitor Gate
Port Gate
Condition Var
PSE
EB Locals

vary from monitor type to monitor type. Second, a field is added to the PSE record which is a pointer to a local_variables record of arbitrary structure. (Putting the local variables directly into the PSE record would lead to a very complicated, confusing, and therefore probably wrong, variant record structure for the PSE record.) Third, the create_PSE routine in each monitor Module generates a new local_variables block each time it is called and installs the proper pointer into the PSE record locals field. Finally, the monitor routines are modified to act on the "local" variables found by following the locals pointer in the PSE record. The lower section of Figure 8 illustrates the situation and Listing 6 shows the modified code for the example monitor. The Event Manager is completely uneffected by the change.

LISTING 6:

```
MODULE pipeline_definition;
TYPE
   PSE_ID = (DAQ_Buffer_Manager, DAQ_Trigger_Supervisor,
             DAQ_Event_Builder, DAQ_Level_3,
             DAQ_Consumer_Computer);

   direction = (Ingress, Egress);
   pipeline_link = RECORD
      source:        ^PSE_control_block;
      destination:   ^PSE_control_block;
      master:        PSE_ID;
      slave:         PSE_ID;
      terminal_link: BOOLEAN;
      source_link:   BOOLEAN;
      END;

   PSE_control_block = RECORD
         PSE:          PSE_ID;
         acquire_port: ^ANYTYPE;
         release_port: ^ANYTYPE;
      locals:          ^ANYTYPE;
         END;

   FUNCTION create_PSE: ^PSE_control_block;
   FUNCTION_TYPE;

   PROCEDURE acquire(port:INTEGER; IO:direction; this:^PSE_control_block);
   PROCEDURE_TYPE;

   PROCEDURE release(port: INTEGER; this:^PSE_control_block);
   PROCEDURE_TYPE;

END (pipeline_definition).


MODULE EB_Monitor;
```

```
EXPORT create_EB;
INCLUDE gladiator;
INCLUDE pipeline_definition;

TYPE
   local_variables = RECORD
      here: this_monitor;
      event_builder_busy: BOOLEAN;
      waiting_for_event_builder: condition_variable;
      END;

FUNCTION create_EB OF TYPE create_PSE;
VAR
   new_EB: ^PSE_control_block;
   new_instance: ^local_variables;
BEGIN
   NEW(new_PSE);
   WITH new_PSE@ DO
   BEGIN
      PSE:=DAQ_Event_Builder;
      acquire_port:=Address(acquire_EB);
      release_port:=Address(release_EB);
      NEW(new_instance);
      locals:=new_instance;
   END;
   create_EB:=new_PSE;
END {create_PSE};

PROCEDURE acquire_EB OF TYPE acquire;
BEGIN
   WITH this@.locals@::local_variables DO
   BEGIN
      enter(here);
         IF event_builder_busy THEN
            sleep(waiting_for_event_builder, here);
         event_builder_busy:=TRUE;
         port:=1;
      leave(here);
   END {WITH};
END {acquire_EB};

PROCEDURE release_EB OF TYPE release;
BEGIN
   WITH this@.locals@::local_variables DO
   BEGIN
      enter(here);
         event_builder_busy:=FALSE;
         port:=0;
         awaken(waiting_for_event_builder, here);
      leave(here);
   END {WITH};
END {release_EB};

END {EB_Monitor}.
```

With these changes, the task of adding a new class of PSE is strictly limited to writing a well-defined prototype monitor Module. The topological section of the Buffer Manager only needs to know about the existence of the PSE type; it can clone off as many copies as it needs and connect them together as it sees fit. The topological module only deals with the invariant PSE record structure. The Event Managers are oblivious to the whole issue; they aren't even aware that there are different PSE types.

Extensions:

Essentially everything described above has been implemented in the prototype Buffer Manager for the September 1985 run. The principle missing component is the dynamic topological module. For now, a choice of pipeline configuration is made at compile time; the arrangement of PSEs into pipeline links is then a fixed feature of the Buffer Manager image.

The creation of pipeline links is the basic job of the topological section. This task only depends on the generic PSE record. Appropriate linkage fields to build the desired PSE network will have to be added along with routines to manage that network. The ability to clone new PSEs is already provided in the create_PSE routines. They are driven by the PSE record structure defined elsewhere and will not care if fields are added. The Determine_next_link procedure will have to be modified to access the new PSE network. Because of the strict partitioning of function in the Buffer Manager system, extant code will not be effected by these changes.

One extension which will effect the Event Manager is the possibility of multiple Consumer Computers. This means that for terminal links only, the destination may in fact be a list of destinations, which list depends on the trigger mask requirements. In that case, the Event Manager must be trained to initiate multiple data transactions simultaneously and not terminate until all the transfers are complete.

References:

Quarrie, David, CDF-183, "Dataflow within the CDF Data Acquisition System", 18 June 1985.

Knuth, Donald E., *The Art of Computer Programming*, Vol. 1, Fundamental Algorithms, Addison-Wesley, 1973.

Wirth, Niklaus, Programming in Modula-2, Springer-Verlag, 1982.

Brinch Hansen, Per, The Architecture of Concurrent Programs, Prentice-Hall, 1977.

Bowen, B. A. and R. J. A. Buhr, The Logical Design of Multiple-Microprocessor Systems, Prentice-Hall, 1980.