WESTERN DIGITAL
C O R P O R A T I O N

Advanced Systems Division

TECHNICAL NOTE

NUMBER: M0013

ADVANCED PROGRAMMING PACKAGE

July 1981

SUMMARY: The following Technical Note describes the Advanced Programming Package which provides MicroEngine users with details on accessing various parts of the operating system. The package contains the following segments:

1) An implementation guide on concurrent processing techniques. It describes in detail the "SIGNAL", "WAIT", "SEMINIT" and "ATTACH" command. These items are useful for users who need to know the structure of semaphores and how queing is performed during concurrency operations.
2) A Pascal representation of the Microcode of "SIGNAL" and "WAIT". This data illustrates how "SIGNAL" and "WAIT" are implemented.
3) An example of a U-Minus Program which illustrates how a programmer can access the operating system global variables.
4) A general document that explains to you how to reference absolute memory locations.
5) A document that details the I/O and interrupt addresses.
6) A document on interfacing to the parallel port which illustrates Western Digital's technique.
7) List of operating system globals.
8) Document on accessing the directory on a disk.

Because these techniques allow the user to access the operating system tables, we recommend that only the more knowledgeable users in your organization should be exposed to the details of this package.

TABLE OF CONTENTS

CONCURRENT PROCESSES

RELEASE VERSION 3.0

IMPLEMENTATION GUIDE

Preliminary

OUTLINE

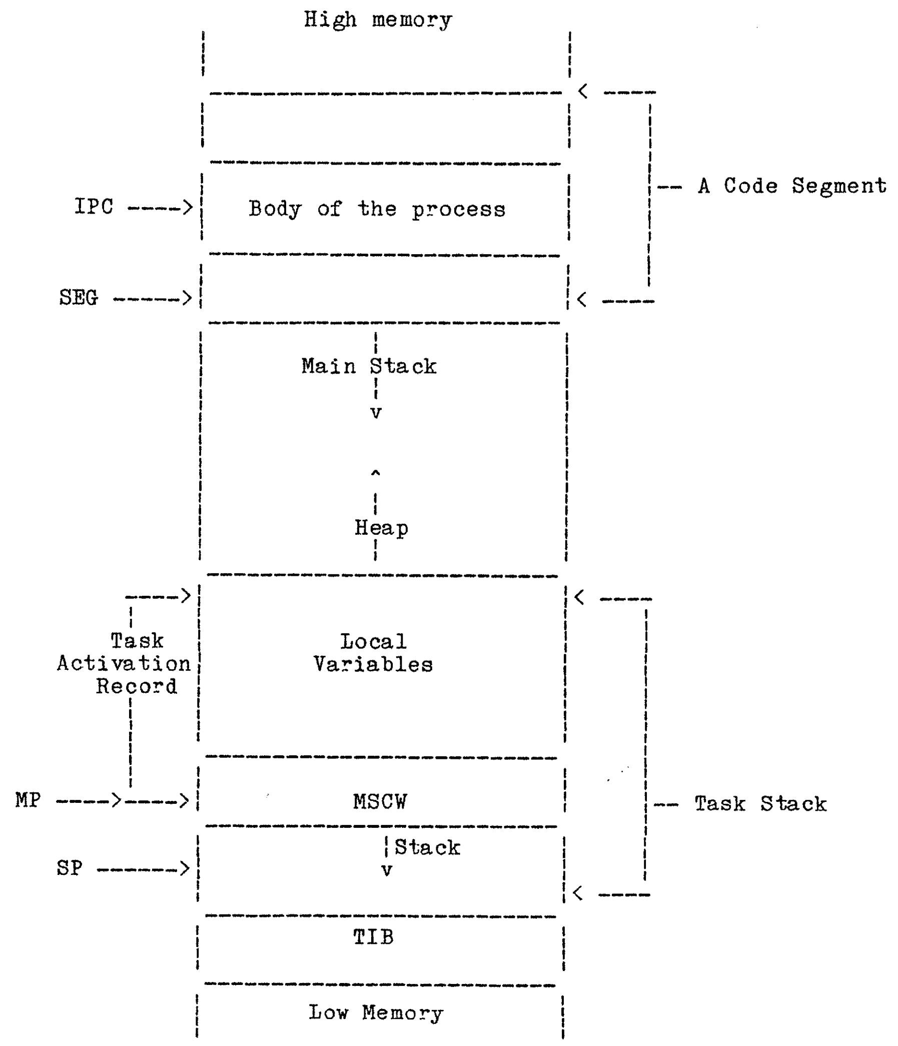

Figure 1 -- Memory during Task Execution

At any given time, the P-machine may have

1 task running
several tasks ready to run
several tasks waiting for each of various semaphores to be signalled.

A P-machine register curtsk always points to the TIB for the currently executing task. Another register readyq points to the first in the list of the tasks ready to run. This is illustrated in Figure 2.

```
                    -----
CURTSK ------>| TIB | -- > NIL                      {register -1}
                    -----

                    -----        -----        -----
READYQ ------>| TIB | -->| TIB | -->| TIB |-->NIL    {register -3}
                    -----        -----        -----

 ---------------
 | count        |
 ---------------    -----        -----        -----
 | waitq        | -->| TIB | -->| TIB | -->| TIB | -->NIL
 ---------------    -----        -----        -----
```

Figure 2 -- P-Machine Task Configuration

1.1 Data Structures

As noted above, there are three main data structures associated with each task, the body, task stack (containing the task's activation record) and Task Information Block (TIB). The body and activation record structures are the same as those for ordinary procedures and the task stack is like the main stack except for its allocation as a fixed space on the heap. Thus, of these data structures, only the TIB remains to be discussed. In addition, there is the newly introduced type semaphore which is also discussed below.

1.1.1 Task Information Block (TIB)

As noted above, the TIB contains information which is used; when a task begins execution, to restore the execution environment (i.e P-machine registers) to its state before the task was interrupted. It also contains other fields which we shall look at.

A Pascal declaration of a TIB is shown in Figure 3. This corresponds to the structure in Figure 4.

```
Type
  byte = 0..255;
  integerp = ^integer;
  mscwp = ^mscw;                           { Mark-Stack Control Word }
  semp = ^semaphore;
  sibp = ^sib;                             { Segment Info Block }
  sibvec = array [0..0] of sibp;
  tibp = ^tib;
```

```
tib  = record { Task Information Block}
       regs:  packed record
              waitq: tibp;                  { Queue link for semaphores }
              prior: byte;                  { Tasks CPU Priority }
              flags: byte                 { State flags, not yet defined }
              splow: integerp;            { Lower stack pointer limit }
              spupr: integerp;            { Upper limit on stack }
              sp: integerp;               { Actual top-of-stack pointer }
              mp: mscwp;                  { Active procedure mscw ptr }
              bp: mscwp;                 { Base addressing environment ptr }
              ipc: integer;              { Byte ptr in current code seg }
              segb: ^codeseg;            { Ptr to seg currently running }
              hangp: semp;               { Which task is waiting on }
              xxx: integer;              { Not used }
              sibs:  ^sibvec;            { Array of sibs for 128..255 }
        end {regs };
      maintask: boolean;                 { True if tib is root task }
      startmscw: mscwp                   { Top mscw in task's stack }
     end { tib }
```

Figure 3 -- Declaration of TIB

```
      MSB            LSB
      ----------------
   0 |  qlink          |
      ----------------
   1 | flags | prior   |
      ----------------
   2 |  splow          |
      ----------------
   3 |  spupr          |
      ----------------
   4 |  sp             |
      ----------------
   5 |  mp             |
      ----------------
   6 |  bp             |
      ----------------
   7 |  ipc            |
      ----------------
   8 |  segb           |
      ----------------
   9 |  hangp          |
      ----------------
  10 |  <reserved>     |
      ----------------
  11 |  sibs           |
      ----------------
  12 |  maintask       |
      ----------------
  13 |  startmscw      |
      ----------------
```

Figure 4 -- Structure of a TIB

The fields sp, mp, bp, ipc and sib contain the values to which the P-machine registers must be restored prior to commencing execution of the task.

The priority field contains the priority level of this task.

Qlink is only used when the process is part of a wait queue, either of a semaphore or ready-to-run. It is used to construct a linked list of TIBs, as was shown in Figure 2.

When a task is waiting on a semaphore, the field hangp is set to point to that semaphore. If the task is not waiting on a semaphore then hangp is nil. One of its purposes is to allow a process to be removed from a semaphore's wait queue.

The flags field is intended to contain state flags for the task. None of these flags are yet defined.

Sibs is a pointer to a mapping array used for segments with numbers 128..255, the details of which have not yet been implemented.

The boolean field maintask simply asserts whether this TIB is for the "root" or outer task, that is the task which invoked all other tasks. Only the operating system task is in this state.

Startmscw points to the Mark-Stack Control Word of the procedure which started this task. It is identical to the dynamic link of the "topmost" MSCW in the task and is used to identify the outer block of the process.

1.1.2 Semaphores

The function of the semaphore data type has been discussed in the "Microengine Reference Manual in concurrent processes. An object of type semaphore contains two elements* a count field and a queue field. A Pascal equivalent declaration of type semaphore is shown in Figure 5, and the field allocation is shown in Figure 6.

```
Type
  semaphore = record
                count: 0..MAXINT;
                waitq: tibp
              end  semaphore  ;
```

Figure 5 -- Pascal Declaration of Type semaphore

```
      ---------------
    0 |   count     |
      ---------------
    1 |   waitq     |
      ---------------
```

Figure 6 -- Structure of Type semaphore

1.2 Primitive Operations

The following discussion covers those operations which are implemented as P-machine instructions (wait and signal) as well as seminit, which is compiled as an in-line sequence of instructions, and the machine-dependent built-in procedure attach. To support the discussion of wait and signal, some internal operations (enqeue, dequeue, taskswitch and idle) are also described. This desciption is intended to support an implementation of these primitives in assembly language or microcode.

1.2.1 Seminit

The built-in procedure seminit accepts two arguements. (1) a semaphore variable and (2) a positive integer (See Tutorial and Users' Guide). The compiler generates in-line code to set the count part of the semaphore to the integer value and the waitq part of the semaphore to nil.

The code is genereated as follows:

```
GIVEN:     seminit ( sem, count  )


CODE:      Load address     sem
           DUP1
           Load             count
           Store@
           INC              1
           LDCN
           Store@
```

This code loads two copies of the semaphore address onto the stack, loads the count and stores it in that address, then increments the other copy of that address by one, making it point to the waitq field of the semaphore, and stores nil there.

1.2.2 Wait and Signal

1.2.2.1 Supporting Operations

1.2.2.1.1. Enqueue

The enqueue operation is used in the implementation of signal and wait. Its function is to insert a task into a queue ordered by priority. The queue is represented by a linked list of TIBs. A Pascal description of enqueue is in Figure 7.

```
         procedure enqueue ( var qhead:  tibp; qtask:_ tibp )
           label 1;
           var t1 , t2: tibp;
         begin
           {  Find  place in queue to insert the TIB  }
           t1  : = qhead;
           t2  : = nil;
           while t1 <> nil do
             begin
               if t1^.regs.priority <qtask^.regs.priority then
               goto 1);

               {   else    }
                  t2  : =  t1;
                  t1  : =  t1^.regs.qlink
             end;
         1:
            qtask^.regs.qlink  : = t1;
            if t2 = nil then
              qhead  : = qtask
            else
              t2^.regs qlink : = qtask
              end {  enqueue  };
```

Figure 7 -- Pascal Description of enqueue

1.2.2.1.2 Dequeue

Dequeue is the complementary operation to enqueue. It is used to remove a task from the head of a wait queue. In the Pascal description of dequeue shown in Figure 8, a pointer to the TIB of the task is returned in qtask.

```
  procedure dequeue ( var qhead, qtask : tibp);
    begin
      qtask : = qhead;
      qhead : = qhead^.regs.qlink
    end  {  dequeue  };
```

Figure 8 -- Pascal Description of dequeue

1.2.2.1.3 Taskswitch

The Taskswitch routine is executed when the currently running task has been put in a wait queue and there is task at the head of the ready queue to be executed. This situation can occur through either of two different circumstances, (1) when a wait has been executed and the current task must wait on a semaphore or (2) when a signal has been

executed and the task activated has a higher priority than the current task. In the former case, the current task is enqueued on the semaphore before Taskswitch, in the latter case, the current task is enqueued on readyq.

Taskswitch sees that all current P-machine registers are saved in the TIB pointed to by curtsk, thus preserving the task's execution environment. In the example presented here, after Taskswitch is executed, the machine falls into Idle, which will activate the task at the head of the ready queue.

```
TASKSWITCH:
           with curtsk^.regs do
             begin
               t sp : = sp;
               t mp : = mp;
               t bp : = base;
               t ipc : = ipc;
               t sib : = seg;
             and;
```

Figure 9 -- Pascal Description of Taskswitch

1.2.2.1.4 Idle

The Idle loop is executed repeatedly when there are no tasks in the ready queue and the processor is waiting for a hardware-interrupt which has been attached to some Pascal semaphore. When the interrupt is received, the processor jumps out of the Idle loop. The interrupt will cause a signal and, since curtsk=nil, control will return to the Idle loop. If the signal caused a task to be placed in the ready queue, that task will now be activated, otherwise the Idle loop will be repeated again.

When the Idle loop is entered from Taskswitch, it kmust be the case that readyq <> nil. Therefore, the loop will be bypassed and the task at the head of the ready queue will be activated.

```
TASKSWITCH:

IDLE:  while readyq = nil do
         begin
           curtsk : = nil
           if An interrupt is being asserted then
             if Its not masked then
               goto Interrupt
             end;
       dequeue ( readyq, curtsk  );
       with curtsk^. regs do
         begin
           sp   : = t sp;
           mp   : = t mp;
           base : = t bp;
           ipc  : = t ipc;
           seg  : = t sib;
         end;
       goto Ifetch;
```

Figure 10 -- Pascal Description of the Idle Loop

1.2.2.1.5 Ifetch

For completeness, we make mention of the Instruction Fetch (Ifetch) code. The only change from previous interpreters is that we make explicit reference to the need for interrupt recognition.

```
IFETCH:   If An interrupt is being asserted then
            if Its not masked then
              goto Interrupt;
          Fetch opcode byte and execute normally;
```

Figure 11 -- Description of Ifetch

1.2.2.1.6 Interrupts

From an architectural standpoint, the receipt of a hardware interrupt which has been attached to a semaphore must cause a signal operation to be executed on that semaphore. Whatever P-machine instruction is executing when the interrupt is raised must be allowed to complete before the signal is performed.

In the Pascal MicroEngine implementation by Western Digital, interrupts are only acknowledged between P-instructions, thus it is possible to immediately perform the signal. It is this environment which has been reflected in the discussions of the Ifetch and Idle code.

For related material, see attach.

```
INTERRUPT:  Obtain vector address from device;
            Push contents of mem word designated by vector onto stack
            goto signal;
```

Figure 12 -- Western Digital Interrupt Handling

1.2.2.2. Wait

The use of the Pascal statement

```
wait (  s  );
```

(where s is a variable of type semaphore) causes code of the following form to be generated by the compiler:

```
LDA     s          ; Push address of s on stack.
WAIT
```

The wait operation pops the address off the stack and uses it to test the count part of the semaphore. If the count is zero then the current task is enqueued on the semaphore and a Taskswitch is made. Otherwise the count is decremented. A Pascal description of wait is shown in Figure 13.

```
WAIT:  { var s:  semp }
       pop(  s  );
       if s^.count = 0 then
         begin
           enqueue ( s^.waitq, curtsk  );
           curtsk^.hangp : = s;
           goto Taskswitch
         end;
       { else }
       s^.count : = s^.count-1;
       goto Ifetch
```

Figure 13 -- Pascal Description of wait

1.2.2.3 Signal

The P-code generated by a signal operation is analogous to that generated by the wait. The Pascal statement

```
signal (  s  )
```

causes the code

```
LDA      s                ;Push semaphore address
SIGNAL
```

to be generated.

During the signal operation, the address of the semaphore is popped off the stack and the queue part is tested. If found to be nil, the count is incremented. If the queue part is non-nil, there are tasks waiting on that semaphore; the task at the head of the semaphore queue is removed and added to the ready queue.

If the priority of the currently running task is less than the priority of task newly readied (which must therefore be at the head of the re ready queue, having a priority greater than all other ready tasks). then a Taskswitch must be made. This preserves the rerquirement that the currently executing task be the highest priority task in the ready queue.

Otherwise the processor jumps directly to the Ifetch code and the cur- rently executing task continues to run.

One special case that can arise is when the CPU has been idling (no current task), waiting for an interrupt. The interrupt is treated as a signal. The only change for handling this case is that, instead of jumping to Taskswitch or Ifetch, when complete, signal jumps back into the Idle loop. This will cause the newly readied task to be started since readyq is no longer nil.

A Pascal description of the algorithm for signal is shown in Figure 14.

```
SIGNAL:  { var s:  semaphore; qtask:  tibp }
         pop (  s   );
         if s^.waitq<>, qtask   );
           begin
             dequeue (  s^.waitq, qtask   );
             qtask^.hangp   : = nil;
             enqueue (  readyq, qtask   );
             if curtsk = nil then
               goto Idle;
             { else }
               if curtsk^.regs.priority < qtask^.regs.priority then
                 begin
                   enqueue (curtsk,  readyq);
                   goto Taskswitch
                 end
               { else }
                 goto Ifetch
         end;
         { else waitq = nil }
         s^.count := s^.count+1)
         if curtsk = nil then
           goto Idle
         { else }
           goto Ifetch
```

Figure 14 -- Pascal Description of signal

2 Attach

The attach intrinsic procedure is intended to establish a logical correspondence between a semaphore and an "interrupt identifier" such that when the specified interrupt is raised by the hardware, the specified semaphore will be signalled. A Pascal declaration of attach would be

```
type                       samp :  semaphore;

procedure attach ( s   :   semp;  interrupt id    :  integer  );
```

The variable s contains a pointer to the semaphore involved, while interrupt id contains whatever the machine requires to identify a given interrupt. When attach is called, the stack is as shown in Figure 15. Both parameters are popped off the stack by the time attach returns.

```
             ////////////////////////
             ------------------------
            |           s            |
             ------------------------
  sp ----->|     interrupt id       |
             ------------------------
```

Figure 15 -- Stack state when attach is called

Attach is called as one of the "standard procedures". For version 3.0, the standard procedures are all in segment 3; attach is procedure number 23 within that segment, thus the call to attach is

```
                    CXG      3,23
```

The details of how attach works are strongly processor-specific due to the varying nature of hardware interrupts. The approach used on Western Digital Corporation's Pascal Microengine illustrates how simple attach can be.

To understand the Western Digital attach, we must first understand the Microengine's interrupt structure. This machine designed to execute P-code directly has as number of possible interrupt "vectors" or, to be more pre- cise, "interrupt codes". A table is maintained in main memory which, indexed by interrupt code, contains the address of the semaphore associated with each code. When the processor acknowledges an interrupt, the interrupting device returns its device-specific interrupt code. The processor then picks up the semaphore address associated with that interrupt code and signals it. To simplify matters further, the semaphore table is based at address 0, so that the interrupt code is, itself the address of the table entry.

```
                                                 -------------
                               |----------->|             |
                                                 -------------
 |                  |          |            |             |
  ------------------           |             -------------
 |                  |          |
  ------------------
 |                  |-----------|
  ------------------                             -------------
 |                  |          |    |------>|             |
  ------------------                             -------------
 |                  |          |            |             |
  ------------------                             -------------
 |                  |------|
  ------------------
Interrupt                                         Semaphores
Codes
(Addresses)
```

Figure 16 --, Microengine Interrupt Structure

This is the attach mechanism for the Microengine consits simply of a Pascal procedure:

```
procedure attach  (s= : semp;  Interrupt code  : ^semp  );
  begin
    Interrupt code  :=  s
    end  { attach },
```

Figure 17 -- Microengine Implementation of Attach

3. PROCESS START

Under the III.0 UCSD concurrency specification, processes are declared in a manner similar to procedures. A process is invoked by the start statement. The purpose of the start statement is to create a Task Information Block (TIB) for the process to be started and to place this TIB representing the process on the ready queue, which holds all processes that are ready to run when processing resources are allocated to them.

Since a process is declared similarly to a procedure, a process may have parameters. In order to insure that the parameters being passed to the process being started are those values at the time of the start statement, there is sychronization code implemented using semaphores to assure that the main program cannot proceed until the sub-process being started has received the parameter values at the time of the start statement. This is implemented by semaphore synchronization performed by compiler generated code in conjunction with an operating system intrinsic. The mechanism basically causes the program starting a process to wait on a semaphore that is signalled when the subtask has received all parameter values.

A process declaration takes the form:

  Process 〈identifier〉 〈formal parameter part〉

A process is started by the procedure Start described by the following format.

```
  Start  ( <process statement>  <processid var>],
          [stacking expression>,
          <priority expression>]]]
```

There are three optional parameters for the Start procedure.

1) Processid - a pre-declared variable type in UCSD Pascal. When present, assigns a value to the variable which is unique to the process that has been started. This points at the TIB created for the process.

2) Stacksize expression - determines how much stack space will be allocated for this process. If no value is given, the compiler allocates a default value of 200 words.

3) Priority expression - determines what processes are handled first by the CPU. The higher priority processes are executed before lower priority processes. If no priority is given, the new process will inherit the priority of its caller.

Start commands can only be called from a main task, such as the outer block of a user's program. If called from a subtask, a run-time error is generated.

```
{PASCAL REPRESENTATION OF SIGNAL & WAIT}

{ This program is a pascal representation of the signal and
  wait commands. The program illustrates how to manipulate the
  ready and wait queues as well giving an insight into the
  workings of the multi-tasking operating system.}
{$g+}
Program pseudointerp;
Label 1, 2, 3, 4, 5, 6;
Type
    tibp = ^tib;
    tib = record {task information block}
            regs: record
                  {working registers ....}
                  prior: integer;
                  qlink: tibp
                end { regs }
           { non-hardware specfic stuff follows }
          end { tib };

    semaphore = record
                  count: Ø..MAXINT;  { number of times signalled }
                  waitq: tibp
                end { semaphore } ;

 Var  { processor registers }
   curtsk,                      { task info block currently in execution }
   readyq,                      { list of tasks waiting for cpu time }
   qhead,                       { global list head operated on by enque and}
                                  { deque }
   qtask: tibp;                 { input and output var used in enque and deque}
   s: ^ semaphore;              { temp storage for wait and signal }

procedure enque;
  label 1;
  var tl, t2: tibp;
begin
  tl := qhead;
  t2 := NIL;
  while tl <> NIL do
    begin
      if tl^.regs.prior < qtask^.regs.prior then
       goto 1;
      t2 :=tl;
      tl :=tl^.regs.qlink
    end;
1:
  qtask^ .regs.qlink := tl;
  if t2 = NIL then
    qhead := qtask
  else
    t2^.regs.qlink := qtask
end {enque};
```

```
procedure deque;
begin
  qtask := qhead;
  qhead := qhead^.regs.qlink
end { deque } ;

function AnInterruptIsBeingAsserted : boolean;
  begin end;
function ItsNotMasked: boolean;
  begin end;

procedure FetchOpcodeByteAndDispatchNormally;
  begin end;

procedure InteractWithDeviceOnBusAndObtainVectorAddress;
  begin end;

procedure PushContentsOfMemWordDesignatedByVectorOnStack;
  begin end;

procedure PopSemaphoreAddressIntoS;
  begin end;

procedure SaveInternalCopiesOfRegistersInCurtskRegs;
  begin end;

procedure RestoreInternalCopiesOfRegistersFromCurtskRegs;
  begin end;

begin { pseudo-interpreter }

  1: { Ifetch }
      If AnInterruptIsBeingAsserted then
        if ItsNotMasked then
          goto 2 { Interrupt };
      FetchOpcodeByteAndDispatchNormally;

  2: { Interrupt }
      InteractWithDeviceOnBusAndObtainVectorAddress;
      PushContentsOfMemWordDesignatedByVectorOnStack;
      goto 4 { signal };

  3: { Wait }
      PopSemaphoreAddressIntoS;
      if s^.count = Ø then
        begin
          qhead := s^.waitq;
          qtask := curtsk;
          enque;
          s^.waitq := qhead;
          goto 5 { Taskswitch }
        end;
      s^.count := s^.count-1;
      goto 1 { Ifetch };
```

```
  4: { Signal }
      PopSemaphoreAddressIntoS;
      if s^.count = Ø then
        if s^.waitq <> NIL then
          begin
            qhead := s^.waitq;
            deque;
            s^.waitq := qhead;
            qhead := readyq;
            enque;
            readyq := qhead;
            if curtsk  = NIL then
              goto 6;
            if curtsk^.regs.prior < qtask^.regs.prior then
              begin
                qtask := curtsk;
                qhead := readyq; enque;
                readyq := qhead;
                goto 5 { Taskswitch } ;
              end;
            goto 1 { Ifetch }
          end;
      s^.count := s^.count +1;
      if curtsk  = NIL then
        goto 6;
      goto 1 { Ifetch };

  5: { Taskswitch }
      SaveInternalCopiesOfRegistersInCurtskRegs;
  6: while readyq = NIL do
      if AnInterruptIsBeingAsserted then
        begin
          curtsk  := NIL;
          if ItsNotMasked then
            goto 2 { interrupt };
        end;
      qhead := readyq ;
      deque;
      curtsk := qtask;
      readyq := qhead;
      RestoreInternalCopiesOfRegistersFromCurtskRegs;
      goto 1 { Ifetch }

end { pseudo-interpreter } .
```

```
{$U-}
{This program demonstrates $U-. This compiler option allows
a programmer to access operating system globals. Be careful
about altering operating system globals as this can have
a deleterious effect. This option also allows dynamic allocation of files
in the heap.}

program pascalsystemexample;

type
  phyle = file;
  inforec = record
             worksym,workcode: ^phyle;
             errsym,errblk,errnum: integer;
             slowterm,stupid: boolean;
             altmode: char;
           end;
var filler: array[0..6] of integer; { space holder for unused OS globals}
    userinfo: inforec;

segment procedure theprogram;
{This segment procedure is the actual user program.}
{The  program's global variables should be declared here.}

type filep = ^phyle;

var cp: filep;
    arr: packed array[0..511] of char;
    c: char;

{Declare 8 segment procedures with no code to make subsequent
 segment procedures fall in the user segments. This is necessary
 as the operating system uses segments 0 and 2-7, while a user
 program has segments 1 and 8 - 15. These 'forward' declarations
 are only needed if the program contains other segment procedures.
 Note that $U- allows forward procedures to remain unresolved,
 since they are needed only as space holders.}

 segment procedure num2; forward;
 segment procedure num3; forward;
 segment procedure num4; forward;
 segment procedure num5; forward;
 segment procedure num6; forward;
 segment procedure num7; forward;

 {The program's segment procedures, if any, go here,}
 segment procedure firstuserses;
 var i: integer;
 begin
  writeln (' in segment  8 ');
  i := i + 1;
end;
```

```
begin

{This code is invoked when this program is executed.}
{In other words, this will be the outerblock of the program.}
{for example, get the altmode character defined by SETUP }
 c := userinfo.altmode;

 {Dynamically allocate a file }
  new(cp);
  reset (cp^,'dum.text');
  if blockread(cp^,arr,2) <> 2 then writeln ('read error ');

   { call the first user segment procedure }
  firstusers;
end;
begin end. {This code will never be executed.}
```

Referencing Absolute Memory Locations

Absolute memory locations can be addressed on the MicroEngine. This use is discouraged as it is easy to corrupt operating system tables or code due to the power of this technique. Because the MicroEngine has memory mapped I/O, even I/O control registers may be accessed, and in fact, the I/O drivers use this technique. Absolute addressing is performed by means of Pascal variant records. A variant record specifies that two different variables with possibly different types may occupy the same memory location. The Pascal program below allows a user to access an absolute memory address interactively.

```
      program examine;

      type memrec = record
                      memcell : integer
                    end;


      var memvariant : record case boolean of
                         true : (memadd : integer);
                         false : (memconts : ^memrec);
                       end;


        i : integer;


      begin
        write (' enter absolute address ');
        readln (i);
        memvariant.memadd :=i;
        writeln (' contents of ',i,' = ',memvariant.memconts^.memcell);
      end.
```

If an address of a MicroEngine I/O port were entered, the program would return the contents of the I/O port register.

```
statcmdrec' = record case boolean of
                true : (command : integer);
                false : (status : packed array [Ø..7] of boolean);
              end; {for devices that use same reg for stat and cmd}

whole = Ø..maxint;
paralrec = record
             porta : statcmdrec;
             portb : integer;
             portc : statcmdrec;
             pcontrol : integer;
           end;
```

```
floppyrec = record
              fstatcom : statcmdrec;
              track : integer;
              sector : integer;
              data : integer;
              filler : array [Ø..3] of whole;
                {dma fields}
              dcontrol : integer;
              dstatus : statcmdrec;
              trcountl : integer;
              trcounth : integer;
              bufaddl : integer;
              bufaddh : integer;
              memex : integer;
              intid : integer;
            end;

serialrec = record
              data : integer;
              statsyndle : statcmdrec;
              control2 : integer;
              control1 : integer;
              filler : array [Ø..3] of integer;
              switch : statcmdrec;
            end;

VAR

  serialtrix : record case integer of
                 Ø: (sdevadd : integer);
                 1: (serial : ^serialrec);
               end;

  paraltrix : record case boolean of
                true : (pdevadd : integer);
                false : (parallel :  paralrec);
              end;

  floppytrix : record case boolean of
                 true : (fdevadd : integer);
                 false : (floppy :  floppyrec);
               end;

Program Serialtest;

{This program illustrates the concepts of referencing absolute
 memory locations by using varriant records. It writes to the
 serial port using unitwrite.}

 Type
   statcmdec = record case boolean of
     true : (command : integer);
     false : (status : packed array[Ø..7] of boolean);
   end; (* for devices that use same reg for stat and cmd*)
   serialrec = record
                 serdata : integer;
                 statsyndle : statcmdrec;
                 control2 : integer;
                 controll : integer;
                 filler : array[Ø..3] of integer;
                 switch : statcmdrec;
               end;
Var
   serialtrix : record case integer of
             Ø : (devadd : integer) ;
             1 : (serial : ^serialrec);
           end;

   Procedure sunitwrite (ch: char);

   Begin
     with serialtrix do
       begin
         devadd := -1ØØ8; (* FC1Ø *)
         with serial^ do
           begin
             controll := 135; (*87 hex *)
             control2 := 1; (* Ø1 *)
             repeat
               until statsyndle.status[Ø];
             serdata := ord(ch);
           end;
       end;
    end;

    begin
      sunitwrite ('h' ); sunitwrite ( 'i' );
    end.
```

Interrupts and Special Addresses

The MicroEngine supports not only I/O devices in the I/O address space, but other functions such as interrupt handling and bootstrap tests. The I/O address space on the MicroEngine ranges from FC00 to FC7F. On the single board MicroEngine the I/O address space contains the system value for NIL, a bootstrap test, and interrupt latches. Addresses in this space on the MicroEngine are implemented using external logic. The table below summarizes the addresses currently utilized in the I/O address space.

FC00	System value for NIL (used for software pointers)
FC10-FC13	Serial port A register addresses
FC18	Switch used for DMA EOB and DINTR signals
FC20-FC23	Serial port B register addresses
FC30-FC37	Floppy disk-DMA register addresses
FC40	Microcode uses this address during interrupt handling (see below)
FC48	Software writes to the latch at this address to to enable all interrupts in the system. This is done as interrupts are disabled after a hardware interrupt.
FC50	Autoload address for DMA (currently unused)
FC60	Dummy address used during interrupt handling (see below)
FC68	Microcode examines this address during system bootstrap to determine whether to boot from floppy or ROM. See section 3.7.2 of the MicroEngine Software Manual.
FC70-FC73	Parallel port register addresses

The addresses FC40 and FC60 are required by the firmware for interrupt handling. The address FC68 is used by the firmware to test whether to boot from floppy disk or ROM during bootstrap. The address FC00 is the microcode recognized value for a NIL pointer. The address FC30 which contains the floppy disk-DMA devices is required by the firmware if boot from floppy disk is desired. In addition, the DMA EOB (end of block signal) and the floppy controller chip signal, DINTR, must be interfaced to address FC18 at bit positions 5 and 4 respectively. The floppy - DMA interface at this address FC30 and the signals at FC18 are not required if boot from ROM is implemented. Chip set users must implement these addresses using external logic.

Interrupts generated by external devices are handled by the Pascal firmware using the addresses FC40 and FC60. A hardware interrupt signal generated by a peripheral device causes the firmware to access address FC40. The write to FC40 is used to latch the interrupt encoder so the interrupt address doesn't change while it is being read. Refer to Table 5-3 in the MicroEngine Hardware Manual for the mapping of devices to interrupt vector addresses. Also refer to the MicroEngine schematics for details of an example hardware interface to the Western Digital Pascal processor. As a part of the interrupt sequence, the firmware executes an instruction that raises IACK, the interrupt acknowledge signal in the firmware. This instruction that raises IACK also must present an address on the bus. The firmware uses FC60 as this address. FC60 is basically a dummy address that must be reserved. The IACK signal causes external logic to gate the interrupt vector address generated by the encoder to present this interrupt vector address on the bus.
The interrupt vector address contains a pointer to a software semaphore attached to the vector address. The firmware then executes the P-code operator, SIGNAL, using the interrupt vector address as a parameter and signals this semaphore. A software I/O driver would WAIT for this semaphore to be signaled in order to proceed.

Interfacing to the MicroEngine Parallel Port

The MicroEngine includes a standard 8255 programmable peripheral interface chip which is memory-mapped into consecutive word addresses beginning at FC70. An 8255 has four 8-bit registers which appear on the data bus as the low-order 8 bits of the data at the four addresses starting at FC70: Port A is FC70, Port B is FC71, Port C is FC72 and the control register (write-only) is FC73. These four consecutive addresses are declared in a Pascal record and bit patterns are deposited in (or read from) the low-order 8-bits of each address by means of two Pascal variant records.

In order to refer to the absolute memory address of the 8255 registers, a variant record is declared to contain either an INTEGER or a pointer to a data-type which maps the register names to the 8255 registers, depending on the field referred to:

```
VAR
    paraltrix: RECORD CASE BOOLEAN OF
            true: (pdevadd: INTEGER);
            false: (parallel: paralrec);
          END;
```

The pointer is made to point to the correct area of memory by assigning the starting address, -912 (-912 signed decimal = FC70 hex), to the field "paraltrix.pdevadd".

The data type which is pointed to is declared as four consecutive words of memory:

```
  TYPE
     paralrec = RECORD
                  porta : statcmdrec;
                  portb : statcmdrec;
                  portc : statcmdrec;
                  pcontrol : INTEGER;
                END;
```

Each of the first three of these words (pcontrol is the control register which is write-only) is in turn declared in the second variant record in such a way as to be either an INTEGER or a series of eight individual bits depending on the field referred to in a Pascal statement:

```
TYPE
   statcmdrec: RECORD CASE BOOLEAN OF
                  true: (command : INTEGER);
                  false : (status : PACKAGED ARRAY[0..7] OF BOOLEAN);
               END;
```

With these three declarations and the assignment of the absolute memory address to paraltrix.pdevadd,the individual 8255 registers can be accessed by the names of the nested records. For example, the refer to port A as an integer, the name paraltrix.parallel.porta.command is used; to refer to the least significant bit of port A, the name paraltrix.parallel.porta .status [0] is used. Note that a packed array of booleans has been im- plemented in exactly the way in which a systems programmer thinks: .status [0] corresponds to the least significant bit of the integer .command or, from a hardware point-of-view, 'bit 0.' The names of the first two of the nested records can be put in a WITH statement and thereafter will be implicit: viz., "WITH paraltrix, parallel DO BEGIN ...... END."

The 8255 is configured on the MicroEngine board with Port A as a buffered input port, with Port B as a buffered output port and with Port C as two input lines and six output lines; only four of the output lines of Port C are brought out to the J3 37-pin connector. Only a few of the possible ways of programming the 8255 will be consistent with this hardware configuration. In particular, mode 2 which configures Port A as a bi-directional 8-bit data bus cannot be used on the MicroEngine board.

The simplest use of the parallel interface is referred to as mode 0. When programmed to mode 0, Port A is an unlatched input port whose data will continuously follow the logic level presented to the assigned pins on the J3 connector and Port B is a latched output port whose assigned pins on the J3 connector will reflect the data last transferred to Port B.

The attached program uses these declarations to program the 8255 and to exercise Port B as an output port. Because the MicroEngine data bus is inverted as seen by the 8255, the bit pattern to be deposited in the control register (pcontrol) must be inverted before being converted to the signed decimal value which is assigned to paraltrix.parallel.pcontrol. Conversion routines are provided for going between binary and signed decimal in either direction but the inversion is not done within the program.

[The MicroEngine Manual specifies that the operating system programs the parallel port control register using the pattern 0110 011x and states that the result is that which would be obtained, according to the specifications given in the manual, with the pattern 1001 100x. The discrepancy is due to the inversion of the data bus in relation to the 8255. In fact, the operating system programs the parallel port control register with the pattern 0110 1111 which sets Port A to unlatched input and Port B and C to latched output. The difference is in bit 3 which controls the direction of the 4 high order bits of Port C.]

The alternative to the mode 0 operation programmed by the operating system is called mode 1. In mode 1:

Port A is a strobed, latched input port with control signals of:

STBA- (STroBe A): an input signal to strobe data into the port;

IBFA (Input Buffer Full A): an output signal to acknowledge to the peripheral that the data has been latched; and

INTRA (INTerrupt Request A): an output to interrupt the CPU when data has been latched: becomes active (high) when the STBA- has gone inactive (high), IBFA is active and the interrupt enable flip-flop (controlled by bit set/reset of PC4) is set (high).

Port B is a strobed, latched output port with control signals of:

OBFB - (Output Buffer Full B): an output signal informing the peripheral that data is available; reset when ACKB - becomes active (low).

ACKB - ACKnowledge B): an input signal from the peripheral sent when the data has been accepted.

INTRB (INTerrupt Request B): an output signal to interrupt the CPU when data has been accepted by the peripheral: goes active (high) when ACKB - is no longer active (high), OBFB - is no longer active (high) and the interrupt enable flip-flop (controlled by bit set/reset of PC2) is set (high).

The two remaining bits of Port C (PC6 and PC7) must be programmed as output to match the hardware buffers. These two bits can be used for output only by means of the bit set/reset function (a write into the control register) and not by writing directly to Port C.

The control register patterns for mode 1 operation are:

For mode 1 on Port A (input) but mode 0 on Port B (output): 1011 0000 (inverted = 0100 1111 = -177) in which case lower Port C will be defined as output and OBFB - (PC1) will be available as an output signal in addition to PC6 and PC7. PC3, PC4 and PC5 are control signals for Port A in mode 1; PC2 is disabled because it is buffered as an input signal and PC0 is not available at the J3 connector.

For mode 0 on Port A (input) but mode 1 on Port B (output): 1001 0100 (inverted = 0110 1011 = -149) in which case upper Port C will be defined as output and IBFA (PC5) will be available as an output signal in addition to PC6 and PC7 (each usable only by means of the bit set/reset function). PC0, PC1 and PC2 are control signals for Port B in mode 1 and PC3 is not available at the J3 connector.

For mode 1 on both Ports A and B: 1011 0100 (inverted = 0100 1011=-181) in which case PC6 and PC7 will be available for output using the bit set/reset function. The other bits of Port C are control signals for Ports A and B in mode 1.

The output signals of Port B and PC1, PC5, PC6 and PC7 are buffered using a 74LS136 exclusive-OR gate which is an open collector device requiring a pull-up resistor of 1K ohms at the device end of the connecting cable.

```
{$U-}

    { ***************************************************************** }
    {                                                                   }
    {   Copyright (c) 1979 Regents of the University of California.     }
    {   Permission to copy or distribute this software or documen-      }
    {   tation in hard or soft copy granted only by written license     }
    {   obtained from the Institute for Information Systems.            }
    {                                                                   }
    { ***************************************************************** }

program pascalsystem;

{ ********************************************** }
{                                                }
{   WESTERN DIGITAL CORPORATION                  }
{                                                }
{    UCSD PASCAL OPERATING SYSTEM GLOBALS        }
{                                                }
{    RELEASE LEVEL:  III.Ø                       }
{                                                }
{ ********************************************** }

const
```

```
     mmaxint   = 32767; { maximum integer value }
     maxdir    =    77; { max number of entries in a directory }
     vidleng   =     7; { number of chars in a volume id }
     tidleng   =    15; { number of chars in title id }
     maxseg    =    15; { max code segment number }
     fblksize  =   512; { standard disk block length }
     dirblk    =     2; { disk addr of directory }
     agelimit  =   300; { max age for gdirp...in ticks (5 seconds) }
     eol       =    13; { end-of-line ...ASCII cr }
     dle       =    16; { blank compression code }
     maxretry  =    10; { retry count for disk drivers }

     maxq          =  79; { type-ahead queue index limit }
     maxqp1        =  80; { type-ahead queue length }
     (*
     minremqavail  =  30; { Send Xoff when q down to this avail }
     remumeqavail  =  80; { Send Xon when q back to this avail   }
     *)

     hiiopriority  = 250; { kbddriver (serial in) processes   }
     midiopriority = 245; { disk in/out, parallel out, serial out }
     lowiopriority = 240; { enabler process for kbddrivers }

TYPE

     iorsltwd = (inoerror,ibadblock,ibadunit,ibadmode,itimeout,
                 ilostunit,ilostfile,ibadtitle,inoroom,inounit,
                 inofile,idupfile,inotclosed,inotopen,ibadformat,
                 istrgovfl);

                                           { COMMAND STATES...SEE GETCMD }

     cmdstate = (haltinit,debugcall,
                 uprognou,uproguok,sysprog,
                 componly,compandgo,compdebug,
                 linkandgo,linkdebug);

                                           { CODE FILES USED IN GETCMD }

     sysfile = (assmbler,compiler,editor,filer,linker);

                                           { ARCHIVAL INFO...THE DATE }

     daterec = packed record
                 month: 0..12;     { 0 IMPLIES DATE NOT MEANINGFUL }
                 day:   0..31;     { DAY OF MONTH }
                 year:  0..100     { 100 IS TEMP DISK FLAG }
               end { DATEREC } ;

                                           { VOLUME TABLES }
     unitnum = 0..maxunit;
     vid = string[vidleng];

                                           { DISK DIRECTORIES }
     dirrange = 0..maxdir;
     tid = string[tidleng];
```

```
filekind = (untypedfile, xdskfile, codefile, textfile, infofile,
           datafile, graffile, fotofile, securedir);

direntry = record
             dfirstblk: integer; { FIRST PHYSICAL DISK ADDR }
             dlastblk: integer;  { POINTS AT BLOCK FOLLOWING }
             case dfkind: filekind of
               securedir,
               untypedfile: { ONLY IN DIR[0]...VOLUME INFO }
                  (dvid: vid;             { NAME OF DISK VOLUME }
                   deovblk: integer;      { LASTBLK OF VOLUME }
                   dnumfiles: dirrange;   { NUM FILES IN DIR }
                   dloadtime: integer;    { TIME OF LAST ACCESS }
                   dlastboot: daterec);   { MOST RECENT DATE SETTING }
               xdskfile,codefile,textfile,infofile,
               datafile,graffile,fotofile:
                  (dtid: tid;               { TITLE OF FILE }
                   dlastbyte: 1..fblksize;  { NUM BYTES IN LAST BLOCK }
                   daccess: daterec)        { LAST MODIFICATION DATE }
           end { DIRENTRY } ;

dirp = ^directory;

directory = array [dirrange] of direntry;

                                    { FILE INFORMATION }

closetype = (cnormal, clock, cpurge, ccrunch);
windowp = ^window;
window = packed array [0..0] of char;
fibp = ^fib;

fib = record
         fwindow: windowp;    { USER WINDOW...F^, USED BY GET-PUT }
         feof,feoln: boolean;
         fstate: (fjandw,fneedchar,fgotchar);
         frecsize: integer;    { IN BYTES...0=>BLOCKFILE, 1=>CHARFILE }
         case fisopen: boolean of
           true: (fisblkd: boolean;     { FILE IS ON BLOCK DEVICE }
                  funit: unitnum;       { PHYSICAL UNIT # }
                  fvid: vid;            { VOLUME NAME }
                  freptcnt,             { # TIMES F^ VALID W/O GET }
                  fnxtblk,              { NEXT REL BLOCK TO IO }
                  fmaxblk: integer;     { MAX REL BLOCK ACCESSED }
                  fmodified:boolean;    { SET NEW DATE IN CLOSE }
                  fheader: direntry;    { COPY OF DISK DIR ENTRY }
                  flock : semaphore;    { File access lock. }
                  case fsoftbuf: boolean of { DISK GET-PUT STUFF }
                    true: (fnxtbyte,fmaxbyte: integer;
                           fbufchngd: boolean;
                           fbuffer: packed array [0..fblksize] of char))
       end { FIB } ;

                                    { USER WORKFILE STUFF }

inforec = record
```

```
                symfibp,codefibp: fibp;           { WORKFILES FOR SCRATCH }
                errsym,errblk,errnum: integer;    { ERROR STUFF IN EDIT }
                slowterm,stupid: boolean;         { STUDENT PROGRAMMER ID!! }
                altmode: char;                    { WASHOUT CHAR FOR COMPILER }
                gotsym,gotcode: boolean;          { TITLES ARE MEANINGFUL }
                workvid,symvid,codevid: vid;      { PERM&CUR WORKFILE VOLUMES }
                worktid,symtid,codetid: tid;      { PERM&CUR WORKFILES TITLE }
              end { INFOREC } ;

{ declarations supporting idsearch / treesearch intrinsics --          }
{ compiler using idsearch will have set up rw table with correct       }
{ len for rwinfo, and have set syscom^.rwtable to point to it.         }
 alpha    = packed array [0..7] of char;
 trsnodep = ^trsnode;                    { symbol table node declaration }
 trsnode  = record                       { -- used by treesearch }
              key   : alpha;
              rlink : trsnodep;
              llink : trsnodep;
            end;
 idsinfo  = record                       { idsearch returns results via this }
              symcursor : 0..1023;       { "pseudo record". compiler must     }
              sy        : integer;       { declare vars in this order and     }
              op        : integer;       { pass its symcursor to idsearch.    }
              id        : alpha;
            end;
 rwtblrec = record
              rwindex : array ['A'..'['] of integer;
              rwinfo  : array [0..0] of
                          record
                            id : alpha;
                            sy : integer;
                            op : integer;
                          end;
            end  {rwtblrec};

                                          { SYSTEM COMMUNICATION AREA }
                                          { SEE INTERPRETERS...NOTE   }
                                          { THAT WE ASSUME BACKWARD   }
                                          { FIELD ALLOCATION IS DONE  }
 syscomrec = record
               unused : array [0..1] of integer; { 2 spare words. }
               sysunit: unitnum;    { PHYSICAL UNIT OF BOOTLOAD }
               rwtable: ^rwtblrec;  { reserved word table for treesearch }
               gdirp: dirp;         { GLOBAL DIR POINTER,SEE VOLSEARCH }
               diskinfo: packed record
                           dseekrate: integer;  {STEP RATE FOR DISK DRIVE}
                           dreadrate: integer;  {DISK READ COMMAND}
                           dwriterate: integer; {DISK WRITE COMMAND}
                         end;
               auxinfo:  packed record    { 5 words total }
                           baudrates:   packed array [0..7] of 0..15;
                                        { 2 words, indices [0,4] not used }
                           xonoff: packed array [0..7] of boolean;
                           clockvalue: integer;   { tick clock rate }
                           menudriven: boolean;   { using *system.menu }
                         end;
               expanstwo:  array [0..12] of integer;  {spare}
```

```
          auxcrtinfo: packed record
                         verdlaychar: char
                      end;
          hightime,lowtime: integer;
          miscinfo: packed record
                      nobreak,stupid,slowterm,
                      hasxycrt,haslccrt,has8510a,hasclock: boolean;
                      userkind:(normal, aquiz, booker, pquiz)
                    end;
          crttype: integer;
          crtctrl: packed record
                     rlf,ndfs,eraseeol,eraseeos,home,escape: char;
                     backspace: char;
                     fillcount: 0..255;
                     clearscreen, clearline: char;
                     prefixed: packed array [0..8] of boolean
                   end;
          crtinfo: packed record
                     width,height: integer;
                     right,left,down,up: char;
                     badch,chardel,stop,break,flush,eof: char;
                     altmode,linedel: char;
                     backspace,etx,prefix: char;
                     prefixed: packed array [0..13] of boolean
                   end
        end { SYSCOM };

miscinforec = record
                msyscom: syscomrec
              end;

memlinkp = ^memlink;
memlink = record
            nextavail: memlinkp;
            nwords: integer
          end { memlink } ;

markp = ^marknode;
marknode = record
             prevmark: markp;
             availlist: memlinkp
           end { marknode } ;

byte       = 0..255;
integerp   = ^integer;
bytearray  = packed array [0..0] of byte;
codeseg    = record case boolean of
               true:  (int: packed array [0..0] of integer);
               false: (byt: bytearray);
             end;

sibp = ^sib;
sibvec = array [0..0] of sibp;
sib = record { segment info block }
      segbase: ^codeseg;{ memory address of seg }
      segleng: integer; { # words in segment }
      segrefs: integer; { active calls - microcode maintained }
```

```
        segaddr: integer; { absolute disk address }
        segunit: unitnum; { physical disk unit }
        prevsp:  integerp;{ SP saved by getseg for relseg cut back }
      end { sib } ;

mscwp = ^mscw;
mscw = packed record  { mark stack control word }
         msstat: mscwp;    { lexical parent pointer }
         msdynl: mscwp;    { ptr to caller's mscw }
         msipc:  integer;  { byte index in return code seg }
         msseg:  byte;     { seg # of caller code }
         msflag: byte
       end { mscw } ;

semp = ^semtrix;
tibp = ^tib;
tib = record { Task Information Block }
        regs: packed record
                waitq: tibp;           { QUEUE LINK FOR SEMAPHORES }
                prior: byte;           { TASK'S CPU PRIORITY }
                flags: byte;           { STATE FLAGS...NOT DEFINED YET }
                splow: integerp;       { LOWER STACK POINTER LIMIT }
                spupr: integerp;       { UPPER LIMIT ON STACK }
                sp: integerp;          { ACTUAL TOP-OF-STACK POINTER }
                mp: mscwp;             { ACTIVE PROCEDURE MSCW PTR }
                bp: mscwp;             { BASE ADDRESSING ENVIRONMENT PTR }
                ipc: integer;          { BYTE PTR IN CURRENT CODE SEG }
                segb: ^codeseg;        { PTR TO SEG CURRENTLY RUNNING }
                hangp: semp;           { WHICH TASK IS WAITING ON }
                iorslt : iorsltwd;     { Result of last I/O call. }
                sibs: ^sibvec          { ARRAY OF SIBS FOR 128..255 }
              end { REGS } ;
        maintask: boolean;
        startmscw: mscwp
      end { TIB } ;

semtrix = record case integer of
            0: (sem: semaphore);
            1: (fakesem: record
                           count: integer; { outstanding signals }
                           waitq: tibp     { task queue }
                         end);
          end { sem } ;

ports = 1..maxport;
cards = 0..maxcard;

statcmdrec = record case boolean of
               true : (command : integer);
               false : (status : packed array[0..7] of boolean);
             end;  { for devices that use same reg for stat and cmd }
whole = 0..maxint;
paralrec = record
             porta : statcmdrec;
             portb : integer;
             portc : statcmdrec;
             pcontrol : integer;
        end;

floppyrec = record
              fstatcom : statcmdrec;
              track : integer;
              sector : integer;
              data : integer;
              fswitch : statcmdrec;
              intprior : integer;
              intbase : integer;
              filler : integer;
              {  dma fields   }
              dcontrol : integer;
              dstatus : statcmdrec;
              trcountl : integer;
              trcounth : integer;
              bufaddl : integer;
              bufaddh : integer;
              memex : integer;
              intid : integer
            end;

serialrec = record
              data : integer;
              statsyndle : statcmdrec;
              control2 : integer;
              control1 : integer;
              filler : integer;
              switch : statcmdrec;
              { special for single board system }
              { ** do NOT touch these fields in the modular ** }
              filler2 : array [0..1] of integer;
              switch2 : statcmdrec;
            end;

sercontrol = record
               readsem, writebell,
               writesem, havch, qlock : semaphore;
               front, rear : integer;
               chq : packed array [0..maxq] of byte;
               serialtrix: record case integer of
                             0: (sdevadd: integer);
                             1: (serial: ^serialrec);
                           end;
               stst: semaphore;
               stwaitno: integer;
               sflag,fflag : boolean; { start/stop, flush }
               wrlock : semtrix;
            end;


iorequest = record                     { Communication link between }
              ioready,                 { unitread/unitwrite and the }
              iohavework,              { I/O driver processes.      }
              iodone : semaphore;
              iounit : unitnum;
              iowindowp : windowp;
```

```
              ioinx,
              iobytes,
              ioflags : integer
            end;

    floppyio = record
                 floppylock : semtrix;
                 floppysem : semaphore;
                 fselect : integer; { reflects unit number }
                 fa : windowp;
                 fblock, finx, fbytes, fflags, fmode : integer;
                 la : windowp; { ptr to 'local' buffer }
                 floppytrix : record case boolean of
                                true : (fdevadd : integer);
                                false: (floppy : ^floppyrec );
                              end;
                 flready, flhaswork : semaphore;
                 fstartit : boolean; { a trix flag }
               end;

    decmax = integer[36];
    longtrix = record case integer of
                 0: (intar: array [0..0] of integer);
                 1: (BCDar: packed array [0..0] of 0..15);
               end {longtrix};

    memtrix = record case boolean of
                true:  (addr: integer);
                false: (loc:  integerp);
              end;

    devtype = (invalid, blocked, parallel, serial);

VAR
    syscom: ^syscomrec;                { MAGIC PARAM...SET UP IN BOOT }
    gfiles: array [0..5] of fibp;      { GLOBAL FILES, 0=INPUT, 1=OUTPUT }
    userinfo: inforec;                 { WORK STUFF FOR COMPILER ETC }
    ostibp: tibp;                      { taskinfo block of op sys prog }
    emptyheap: ^integer;               { HEAP MARK FOR MEM MANAGING }
    inputfib,outputfib,                { CONSOLE FILES...GFILES ARE COPIES }
    systerm,swapfib: fibp;             { CONTROL AND SWAPSPACE FILES }
    syvid,dkvid: vid;                  { SYSUNIT VOLID & DEFAULT VOLID }
    thedate: daterec;                  { TODAY...SET IF FILER OR SIGN ON }
    state: cmdstate;                   { FOR GETCOMMAND }
    heapinfo: record     { heap management }
                lock: semaphore;
                topmark,
                heaptop: markp
              end { heapinfo } ;
    taskinfo: record     { stuff for task management }
                lock: semaphore;
                taskdone: semaphore;
                ntasks: integer
              end { taskinfo } ;
    ipot: array [0..4] of integer;    { INTEGER POWERS OF TEN }
    filler: string[41];                { NULLS FOR CARRIAGE DELAY }
    digits: set of '0'..'9';
```

```
pl: string;
chainname: string[23];   { chainer sets this - length > 0 means }
                         { next getcmd executes chainname }
unitable: array [unitnum] of { 0 NOT USED }
            record
              uvid: vid;        { VOLUME ID FOR UNIT }
              case uisblkd: boolean of
                true: (ueovblk: integer);
            end { unitable } ;
filename: array [sysfile] of string[23];
topofsibs: ^integer;
safediskmode : boolean ;
port : array [ports] of sercontrol;

.........Variable access by system U- programs ends here..........}

paraltrix : array [cards] of
              record case boolean of
                true : (pdevadd : integer);
                false : (parallel : ^paralrec);
              end;
flport : array [0..maxcard] of floppyio;
parsem : array [cards] of semaphore;
pariolock : array [cards] of semtrix;
enabletrix : memtrix;          { for enabling interrupts }
is64kmem: boolean; { set by initialize }
unitvalid: packed array [unitnum] of boolean;
unittype: array[unitnum] of devtype;
clockinfo: record
             lock,
             clocksem : semaphore;
             tickrate : real;
           end;
exceptint : semaphore ;
breaksem : semaphore;   { Signaled by 'kbddriver' on user break. }
seroutport : array [ports] of iorequest;
parport : array [cards] of iorequest;
dirlock : semaphore;
```

DIRECTORY ACCESS ON THE PASCAL MICROENGINE

A diskette is composed on granules called blocks. Each block contains 512 bytes. A single-sided, single-density diskette contains 494 blocks numbered from Ø - 493. A double-sided, double-density diskette contains 1,976 blocks.

The directory for a diskette resides on block numbers 2-5 (i.e., it occupies 4 disk blocks. If there is a duplicate directory, this resides on blocks 6-9. Among other things, the directory contains the name of the diskette, the name of each file on the diskette, information concerning the starting and ending block for each file, and the date of each file's creation.

The Pascal declaration for the directory is shown below. It is identical to that shown in the operating system globals.

```
direntry = record
             dfirstblk: Integer,   {FIRST PHYSICAL DISK ADDR}
             dlastblk: Integer,   {POINTS AT BLOCK FOLLOWING}
             case dfklnd: filekind of
               securedir,
               untypefile:  {ONLY IN DIR [Ø] ...  VOLUME INFO}
                 (dvld: vld;
                 deovblk: Integer,         {LASTBLK OF VOLUME}
                 dnumfiles: dlrrange;      {NUM FILES IN DIR}
                 dloadtime: Integer;       {TIME OF LAST ACCESS}
                 dlastboot: daterec);      {MOST RECENT DATE SETTING}
           xdskfile,codefile,textfile,Infofile,
           datafile,graffile,fotofile:
                 (dtld: tld;
                 dlastbyte: 1..fblksize;  {NUM BYTES IN LAST BLOCK}
                    daccess: daterec)      {LAST MODIFICATION DATE}
           end {DIRENTRY} ;

dirp = ^directory;

directory = array [dirrange] of direntry;
```

The following program fragment reads the directory from disk drive #4.

```
  VAR gdirp: dirp;

  begin
    new (gdirp);
    unitread (4, gdirp^, sizeof (directory), 2);
```

After this read from disk of the directory, the fields in the directory may be examined. For example, to access the date on the diskette:

```
  with gdirp  [Ø] . dlast boot do
    writeln ('today is', month, '/', day '/', year);
```