A UNIFIED DATA FLOW MODEL FOR FAULT TOLERANT COMPUTERS

Final Report

T. R. Rao
V. V. Ramanamurthy
Abbas Youssefi

University of Southwestern Louisiana
Lafayette, Louisiana

and

K. M. Kavi
University of Texas at Arlington
Arlington, Texas

April 25, 1984

This report is supported by research grant NAG - 1 - 271

I. INTRODUCTION

We have used the Dataflow Simulation System (DFSS) at USL as the medium on which we have produced a functional simulation of SIFT. DFSS is written in PL/I and is supported by MULTICS. Within the simulation, all the interprocessor communication, fault simulation, system state data, and monitoring were implemented in dataflow and supported directly by DFSS. The actual processor level computation was carried out by the SIFT code in PASCAL. The interface between DFSS in PL/I and the SIFT code in PASCAL was supported under a mechanism in DFSS called a Node Realization Module (NRM).

There were several reasons for making these choices for this initial simulation. The first reason was that these choices allowed us to effectively use the SIFT code that was available. Also, DFSS supported the addition of appropriate monitoring and fault simulation mechanisms. Another reason involved the fact that this preliminary simulation should point out some of the limitations of the current dataflow simulator and provide useful information in designing the new dataflow simulator in PASCAL. The primary reason for using dataflow only at the highest levels of SIFT was related to the way interprocessor communication took place in the last version of SIFT. Virtually any procedural process can be simulated in dataflow on DFSS. However, as a system simulation medium, dataflow worked best in dealing with parallel, asychronous, communicating processes.

This report is divided into three sections and each of these sections describes different phases of the simulation of SIFT on DFSS. Section II outlines briefly the changes that have been made to the SIFT code so that the program could be run on the PASCAL compiler at USL. Section III gives the interface mechanism between the SIFT code in PASCAL and the PL/I program. This is necessary to include SIFT code as a node within the dataflow graph. In section IV the simulation of a single processor on the DFSS is described with the basic structure of the node. This section also describes an approach to the simulation of the complete system with n (n=5) processors. The interprocess communication buffers are concurrent access files with each of the processors having access to them as required by the operating system. This is achieved by means of the a PL/I interface program. The details are given in the specific section.

II. SIFT PASCAL CODE

Some functions in SIFT code were realized in the assembly language of BDX 930. In order to obtain the same kind of accuracy of numerical values computed by these functions, they were translated into PL/I. PL/I was used mainly to utilize the range of options available in obtaining the required accuracy.

The language dependent routines BAND (bitwise 'and') and BOR (bitwise 'or') were written in PL/I. To use these PL/1 routines the statements containing the actual functions were modified.

For example: (framecount mod scheds[vp+1])
was changed to
mod(framecount,scheds[vp+1]);

(eofbit band dbx*8)
was changed to
band(eofbit,dbx*8);

Another change was made to the code. All the location dependent variables were made independent of the system. The 'at' construct used in the original code was not present in either version of PASCAL compilers available at USL. So these variables were made independent of the locations in memory. In fact, because of the virtual memory system on MULTICS, no construct forcing static memory location is available. This problem is overcome simply by removing the 'at' {location} from the variable declaration part.

For example: dbfile at dbloc : integer;
is changed to
dbfile : integer;

The system dependent routine 'gprocessor' is not necessary because the processor id could be initialzed at the beginning of the process. The new version of the SIFT code is in Appendix - I.

III. INTERFACE PROGRAM

In order to realize the node realization module as described in [1], we used a PL/I program which called the SIFT code as a subroutine and there should be a correspondance between the two. While passing the parameters of type array or structure (record), a pointer to an array or a structure was passed. An array or a structure of same type was declared in the PL/I program with a pointer. This pointer was then equated to the external pointer (PASCAL pointer).

An example to reflect the parameter passing mechanism is given :

```
PASCAL declaration : dffile : array(0:1023) of integer;
                     dfptrpas :

PL/I  declaration : dcl dfile(0:1023) fixed bin(35) based(dfptr);

                    dfptr = dfptrpas;
```

With the above statement, any reference to the PL/I array element was reflected in the corresponding PASCAL array. All the other simple variables were external.This program is in Appendix - II.

This PL/I program also realized the interprocessor communication through a procedure which was called whenever the transaction pointer in the "broadcast" or "stobroadcast" routine of the SIFT code was set. The interprocessor buffers were concurrent access files and these could be accessed and checked after every subframe.

IV. SIMULATION OF SIFT ON DFSS

Each individual processor node within the dataflow graph of complete system is given in figure 1. The dataflow program to realize this basic node is given in Appendix - III. With each of these nodes representing a processor, complete simulation with n (5) processors was attempted. Each processor was supposed to execute the current schedule : compute, broadcast, and vote according to schedule and assume that the other processors are also doing the same thing. The behavior of a processor is not immediatly changed if communication breaks down or computation faults occur. The only changes in the behavior of the processor, caused by failures in interprocessor communication, are long term. If failures take place, eventually a reconfiguration will occur.

The error buffer data and the data buffer values could be examined through the PL/I interface program at the end of every subframe.

The details of the scheduling tables could not be worked out as expected due to the fact that these were wirtten in assembly language of BDX 930. In effect, the bottleneck in implementing the simulation was the scheduling tables.

ON DATA FLOW SIMULATOR

The major responsibility of the research team at the University of Texas at Arlington was the design of a data flow language and a data flow simulator for this language written in Pascal for VAX/780 system.

DFDLS (pronounced as daffodils) is such a data flow simulator written in Pascal and at present is available on DEC 20 system. It will be available on VAX/780 soon. Simulation of computer systems, both hardware and software, can be performed using data flow concepts. For an introduction to data flow, the reader is referred to [1].

The major features of DFDLS are given below.

1. Tokens on data flow arcs can be structured data items. Existing data flow computers and languages permit only elementary data types like integer, real and characters. DFDLS allows all the Pascal data types. At this time, record data types are not processed, but will be included soon.

2. Firing set semantics can be specified in DFDLS. In the basic data flow, a node is enabled for execution only when all input arcs contain tokens and no tokens are present on the output arcs. This firing semantics are extended in DFDLS by describing alternate firing sets. Each set defines the mandatory input values required for the node to execute.

3. Nodes in DFDLS can be data flow subgraphs. In most of the existing data flow systems only primitive functions such as ADD two numbers are permitted. In our system, a node can be a primitive function defined by the system, a data flow subgraph or a Pascal procedure provided by the user. These Pascal procedures are linked dynamically by the runtime environment.

4. The input language is very simple. DFDLS interprets simulation models expressed in our textual language. There is a one-to-one correspondence between the data flow graphs

and the textual representation. Thus, data flow graphs can be translated directly into the input language. This also provides for graphical interface that can be designed at a later date.

5. Block structures and Recursion: The present implementation of DFDLS permits a restricted block structure in that, all names must be unique. Recursion is not allowed. However, we are in the process of extending DFDLS to allow more general nesting of blocks and recursion. Because of our data structures and modular design of DFDLS, this addition is straight forward. Separate descriptor tables and node tables will be created for each block and display stack will be used to implement recursion and static scopes.

A complete description of DFDLS and a detailed outline of the design can be found in [3].

We made every attempt to complete the design and testing of DFDLS on VAX/780 before the end of 1983. However, due to unforeseen problems, we are only able to complete the design and test DFDLS on DEC 20. We would like to describe briefly the technical problems faced by the research team at UTA.

Due to the policies of the board of Regents of the state of Texas, the acquisition of VAX was delayed by several months. The hardware had arrived at UTA in December 1983. We are awaiting the installation of a new air-conditioning equipment before bringing up the VAX system.

The university and the department of Computer Science and Engineering have seen significant increases in the enrollment. The computing facilities are not adequate to handle the increase, resulting in severe restrictions on the use of disk space, access to terminals and in general, degraded response time.

The current version of DFDLS is being implemented on DEC 20/60 computer system, awaiting the arrival of VAX 780. Although most of the code for DFDLS is written in Pascal and thus portable, the section that dynamically links user supplied Pascal procedures with the runtime environment of DFDLS is very machine dependent and must be rewritten for VAX.

During the execution of a data flow program, the function of a data flow node can be realized by invoking either system provided Pascal procedures (Library functions) or user provided procedures. This facility enables us to add reliability calculations to the simulator. There is no easy way of linking programs dynamically on DEC 2060 system. Several alternatives have been considered and discarded due to the complexity of implementation.

A dynamic linker could be written with DFDLS to interface with the standard DEC linker provided. This requires significant amount of code in Macro-assember for DEC 2060. Since DFDLS will be transported to VAX, this solution was discarded. Pascal procedures could be added to the DFDLS and the entire system could be recompiled and linked for each run. This solution was not entertained because of the inefficiency.

Our next choice was to have the NRM (node realization module) to invoke the user supplied procedure as a separate task with the data space mapped into the data space of DFDLS. It was hoped that this option would allow access to parameters similar to call by

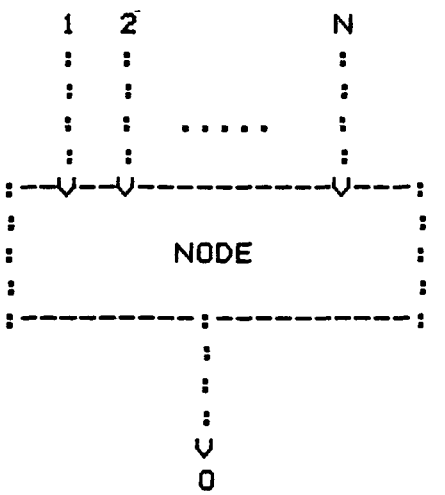
reference technique. However, two problems exist with this method. Sharing of data space is not possible because the sub-process (corresponding to the user supplied procedures) would overlay the DFDLS space resulting in the destruction of crucial data belonging to the simulator. The second problem arises from the fact that in order to invoke sub-tasks, it is necessary to make user procedures as independent programs. This would require that the user write programs instead of procedures to link with the runtime environment of DFDLS.

Despite of the restrictions imposed by the last solution, we have decided to pursue this approach in implementing DFDLS on DEC 2060. We will relax these restriction in VAX 780 implementation. In the present solution to the linking problem, user supplied Pascal procedures are made into complete programs and DFDLS will communicate with these programs through standard disk files. The parameters are written into files and read from files by both user programs and DFDLS.

2. REPORT ON RELIABILITY MODELS FOR DATA FLOW

The research team at UTA has also attempted to develop reliability models for data flow graphs, so that the reliability of computer systems modeled as data flow graphs can be predicted. To our knowledge there exists no published reliability studies of data flow systems. A survey of related models can be found in [3].

A recursive algorithmic method can be used to determine the reliability of a data flow graph. The reliability of the output from a node depends on the reliabilities of the inputs to the node and the reliability of the node itself.



$$R(O) = g(f(R1, R2, ..., RN), Rnode) \qquad (1)$$

FIG. 1 RELIABILITY OF A DATA FLOW NODE

Here f is a combinatorial function describing the input configuration of the node, R1, R2, ..., RN are the reliabilities of the inputs and Rnode is the reliability of the node. The node in the above calculation can be a subgraph, thus providing for a recursive definition.

2.1 Our Approach:

We have developed a method that combines Markov processes with the recursive algorithmic method described above. A path from an input of a data flow graph to an output of the data flow graph is defined as the alternating sequence of arcs and nodes, a1, n1, a2, n2, ..., ao, where a1 is the input arc and ao is the output arc. The reliability of the path can be determined using Markov methods.

A significant structural property of a data flow system is its capability for parallel processing with split and merge of job streams at various levels. This leads to multiple parallel paths between an input to the data flow graph and an output from the data flow graph. The parallel paths need not be independent. The dependencies will be handled by the algorithmic method.

For example, let there be M parallel paths between a given input and output of the system. Let R be the overall reliability of the output with respect to the given input, which is to be determined. Also, suppose R1, R2, ..., RM be the corresponding reliabilities of the M parallel paths, obtained using Markov methods. Then R is given by

$$R = g (R1, R2, ..., RM) \qquad (2)$$

where g is the function describing the inter-relationships between the parallel paths.

The reliability measure of the entire system can be obtained by calculating the reliabilities of all outputs from the system with respect to every input to the system. We are currently working with this method and developing the algorithms required to compute the inter-relationships between the parallel paths.

3. REFERENCES

[1]. Rao, T. R. and Myers L. Foreman, "A Unified Data Flow Model for Fault Tolerant Computers", Memorandum #3 01/03/1983.

[2]. Treleaven, P.C., Brownbridge, D.R. and Hopkins, R.P. "Data Driven and Demand Driven Computer Architecture", ACM Computing Surveys, March 1982.

[3]. Kavi, K.M. "Data Flow Models For Fault Tolerant Computers", Interim Progress Report for NASA Grant NAG-1-271, UTA Tech Rept. CSE 83-3, Nov. 1983.

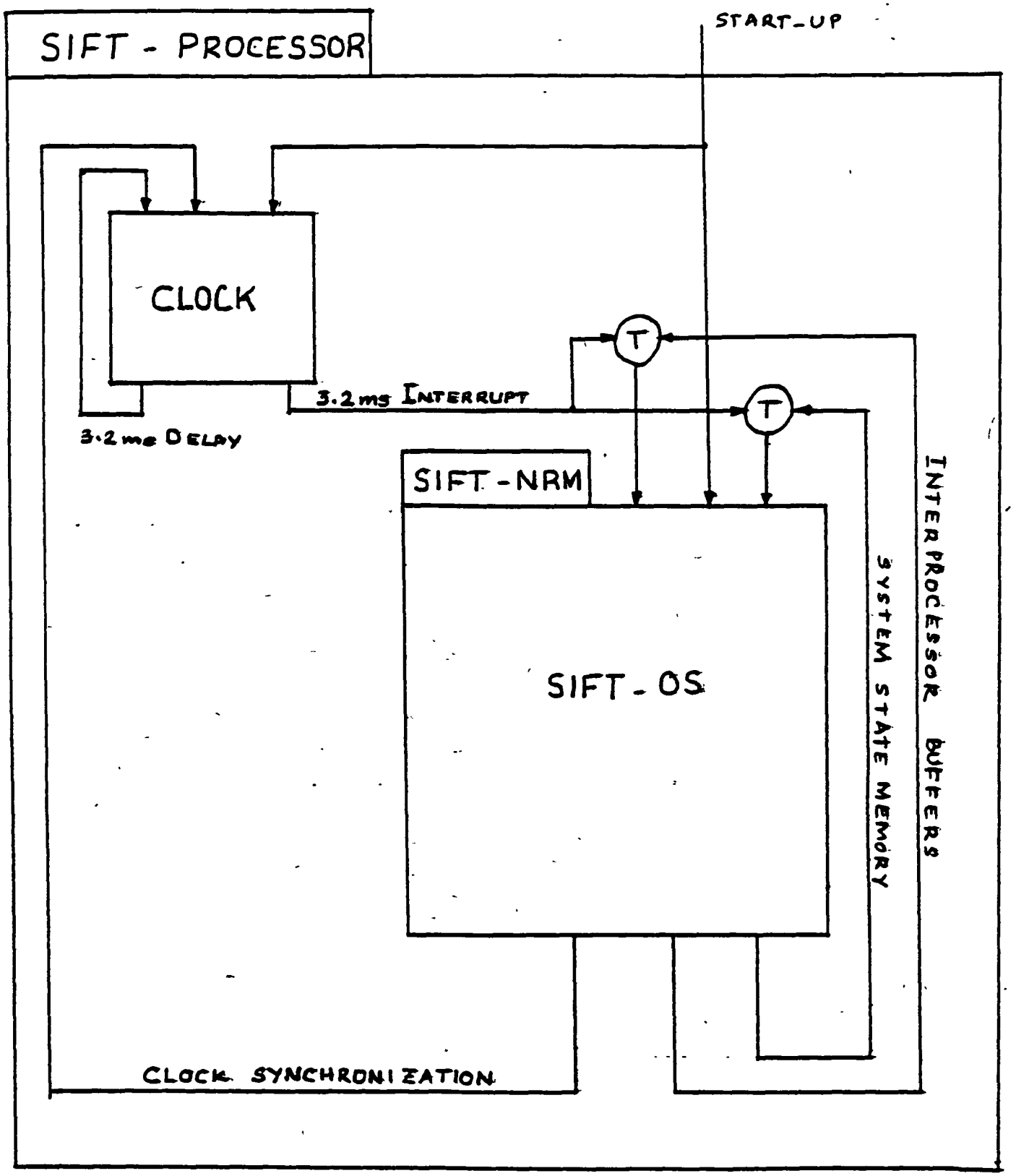

SIFT - PROCESSOR
START-UP
CLOCK
3.2ms INTERRUPT
3.2ms DELAY
T
T
SIFT-NRM
SIFT-OS
SYSTEM STATE MEMORY
INTERPROCESSOR BUFFERS
CLOCK SYNCHRONIZATION

Figure-1

Appendix - I

```
program newsift(input,output,datafile01);
%export clock,dfptrpas $
{*procedure band(var a,b : integer) ; external;*)
{*procedure bor( var a,b : integer) ; external;*)
const
   maxtime=71;              (* max skew allowed in clock task *)
   maxdata=1015,            (* highest address in the datafile *)
   maxtrans=1023,           (* highest address in the trans. file *)
   maxdb=127,               (* highest address in a databuffer *)
   dbsize=128;              (* size of a databuffer *)
   eofbit=32768,            (* end of file bit for transaction *)
   maxprocessors=7;         (* highest processor number less 1 *)
   maxstate=128,            (* largest number of items in a statevector *)
   maxframe=7;              (* Maximum frames in a cycle. *)
   maxsubframe=21,          (* last subframe in a frame *)
   maxreconfig=1791,        (* Number of elements in reconfig schedules *)
   maxbinf=200,             (* maximum for table which tasks broadcast *)
   tpbase=896;              (* minimum value of the transaction pointer *)
   magic=769;               (* 2*tpbase-1023 *)
   tpclock=1019;            (* clock in datafile. *)
   tasks=12;                (* number of tasks in the system *)
   maxbufs=119;             (* maximum number of buffers  *)
   tentrysize= 133,(* 5 + maxstate; *)  (* size of a task entry *)
   ttsize= 1729;(*tentrysize * (tasks+1);*)  (* size of the task table. *)

   (* The following constants define scheduling masks  *)

   datamask=4097;           (* Data portion of a schedule entry *)
   extendmask=8192,         (* Extended task entry *)
   contmask=4096,           (* Continue prior execution only *)
   suspmask=2048,           (* Task may be suspended at clock tick  *)

   (* the following are constants to be used when refering to buffers. *)

   errerr=33;
   gexecreconf=34;
   gexecmemory=35;
   expected=36;
   lock=37,
   ndr=38,

   xreset=39,

   (* The q series is the 1553a input value.  The a, b, and c series
   are the values re-broadcast as part of interactive consistency,
   corresponding to the 1st, 2nd and 3rd input processors. *)


   (* note  -- for phase 2 the q buffers have been eliminated. data is
   now input into a,b or c buffers for p's 1,2 and 3 respectively *)


   astart=40,
   aalpha=40, abeta=41, acmdalt=42; acmdhead=43; adistance=44,
   aglideslope=45, alocalizer=46, ap=47; aphi=48; aphitrn=49,
   apsi=50, aq=51, ar=52; aradius=53; arturn=54; atheta=55,
   au=56; ax3=57; axcntr=58; ay3=59; aycntr=60;
   alast=60;

   balpha=61, bbeta=62; bcmdalt=63, bcmdhead=64, bdistance=65,
```

```
    bglideslope=66, blocalizer=67; bp=68; bphi=69; bphitrn=70;
    bpsi=71, bq=72, br=73; bradius=74; brturn=75; btheta=76;
    bu=77, bx3=78, bxcntr=79, by3=80; bycntr=81;

    calpha=82, cbeta=83; ccmdalt=84; ccmdhead=85, cdistance=86,
    cglideslope=87, clocalizer=88; cp=89, cphi=90, cphitrn=91;
    cpsi=92; cq=93, cr=94; cradius=95; crturn=96; ctheta=97;
    cu=98; cx3=99; cxcntr=100; cy3=101; cycntr=102;

    (* The o series are the 1553a output values  *)

    ostart=103;                  (* must correspond to first of o series *)
    qcmdail=103, qcmdele=104; qcmdrud=105, qcmdthr=106,
    qdely=107, qdelz=108; qpitmo=109; qlatmo=110; qreconf=111,
    olast=111;                   (* must correspond to last of o series *)

    osynch=112,

    (* Internal values.  *)

    phin=113; psin=114; rn=115,
    qx=116; qy=117; qz=118; timer=119;

    (* 1553a values.  *)

    appnum = 16,      (*timer-ostart+1;*)
    onum = 103;   (*ostart,*)            (* beginning of saved region *)

    num1553a=21,   (*alast-astart+1,*) (* number of items to read *)
    onumi553a=9,(*olast-ostart+1,*) (* number of items to write *)
    bas1553a=936,(*tpbase+astart;*)  (* first input location *)
    mas1553a=255;(*16#00FF*,*)                    (* status bits *)
    out1553a=9; (*olast-ostart+1;*) (* number of items to transmit *)
    obas1553a=999,(*tpbase+ostart;*) (* first output location  *)
    sa0=0;                       (* subaddress 0*)
    sa1=32,                             (* subaddress 1*)
    rec1553a=1024,             (* Receive *)
    tra1553a=0;                          (* Transmit *)
    rt1=2048;                           (* remote terminal 1 *)
    sbas1553a=1008,(*tpbase+osynch,*) (* synch word  *)

    (* the following constants are to be used when refering to tasks  *)

    nullt=1,                     (* the null task *)

(* the following constants specify address of some preinited tables
tfloc=16#3400;                          Address of transaction file  *)
(* gfrlc=16#3800;                       Address of global framecount *)
(* sfclc=16#3801;                       Address of subframe count*)
(* dbloc=16#3802;                       Address of dbad.  *)
(* stackloc=16#5000,                    "Exec Stack" location - siftih *)
(* tloc=16#5500;                        Address of tt.  *)
(* numloc=16#6836;                      Address of numworking *)
(* pidloc=16#6837;                      Address of pid.  *)
(* virloc=16#6838;                      Address of virtno.  *)
(* pvloc=16#6840,                       Address of post vote buffer *)
(* bloc=16#68C0;                        Address of bt *)
(* sloc=16#6D00;                        Address of scheds.  *)
(* dfloc=16#7400;                       Address of datafile.  *)
(* pfloc=16#77F8;                       Address of pideof.  *)
```

```
  (* tploc=16#77F9;                          Address of trans pointer   *)
  (* s15loc=16#77F9;                         Address of sta1553a  *)
  (* clkloc=16#77FB;                         Address of real time clock *)
  (* c15loc=16#77FD;                         Address of cmd1553a  *)
  (* a15loc=16#77FF;                         Address of adr1553a  *)
  (* iloc=16#7800,                           Address of buffer info.*)
  (* dest=16#8000;                           eofbit bor loc zero *)
type
   data = file of integer;
   dfindex=0. maxdata,                       (* data file *)
   dbindex=0..maxdb;                                (* data buffer *)
   tpindex=0. maxtrans;                      (* transaction file *)
   processor=0..maxprocessors;                (* processor *)
   buffer=0. maxbufs,                        (* one for each buffer. *)
   task=0  tasks;                            (* one for each task  *)
   bufint=array[buffer] of integer,
   procint=array[processor] of integer,
   procbool=array[processor] of boolean; (* beware addr *)
   bufrec=record dbx:integer, ad.procint end; (* beware addr *)
   dftype=array[dfindex] of integer;
   tftype=array[tpindex] of integer,
   statevector=array[0. maxstate] of integer,
   schedcall=(tasktermination,clockinterrupt,systemstartup);
   dfptrty = ^dftype;
   taskentry=record
             status schedcall;             (* cause of the last pause  *)
             bufs integer,                  (* ptr to list of bufs broadcasted. *)
             errors.integer,                (* Number of task overrun errors  *)
             stkptr integer;                (* last stack pointer *)
             state statevector,             (* stack for task *)
             end,

var
   datafile01 . data,
   dfptrpas   dfptrty;
   datafile : dftype,
   transfile . tftype;
   transptr   integer;              (* transaction pointer *)
   pideof  . integer,                (* processor ID discrete (read) *)
   (* end of file discrete (write) *)
   scheds : array[0 .maxreconfig] of integer;
   (* Precomputed schedules forreconfiguration *)
   postvote   bufint;                (* post vote buffer *)
   dbad : procint;                   (* index to start of data buffer for each proce
   bt : array[buffer] of bufrec,  (* where and who broadcasting*)
   tt : array[task] of taskentry,(* Task Table *)
   binf : array[0. maxbinf] of integer; (* bufs where? *)
   clock : integer;                  (* real time clock (read/write)*)
   clk1  : integer;                  (* used to prevent optimization *)
   adr1553a   integer;               (* 1553a address register *)
   cmd1553a : integer,               (* 1553a command register *)
   sta1553a   integer,            (* 1553a status register *)
   pid : integer,                 (* My processor number *)
   numworking : integer;          (* Number of working processors *)
   virtno . procint;              (* Virtual processor numbers. *)
   gframe . integer,                 (* global frame count *)
   sfcount : integer;                (* sub frame count *)

(* p4deec var *)
```

```
    votecnt,vtime,wtime,delta: integer;
    working: procbool;                          (* Working processors *)
    p,v,errors: procint;                        (* voting *)
    p1,p2,p3,p4,p5,v1,v2,v3,v4,v5: integer;     (* more voting *)
    taskid  task;                               (* Number of currently running task *)
    taskbits: integer;                          (* Control bits associated with task *)
    lastconfig: integer;                        (* The last configuration *)
    pclock,cclock,bclock,aclock,                (* globals for clock synchronization *)
    tp,vp,cptr,                                 (* schedule pointers as above. *)
    tpi,vpi,cptri,                              (* start of schedule pointers. *)
    framecount. integer;                        (* The current frame count *)
    skew. procint;
    teatime:  integer,                          (* For timing the clktask *)



    lines,pages,pagelimit,reason: integer;
    fatal: boolean,
    hlttrue,errtr,vottr,rectr: boolean,

    runid     . integer,
    minutes   . integer,
    tskst     : integer;
    tskfn     : integer,
    hlttb     : boolean;
    errtrb    . boolean;
    vottrb    . boolean,
    rectrb    : boolean,

    sk    array[0. 801] of integer;
    skptr  integer,

    stop: integer;


(* include 'p4dec con';
include 'p4dec glo',
include 'p4dec.var',*)

(*procedure gprocessor;*)
  (* Set the current processor as a number between 0 and maxprocesors. *)
(*begin*)
(* pid .= ((pideof div 4000B) band 16#0F)-1*)
(*end,  gprocessor *)
function band(a,b.integer): integer,
begin  band := a + b ;       end;

function bor(a,b :integer)   integer,

begin   bor := a - b, end;

procedure dbaddrs;
   (* calculate the index of the start of each of the databuffers.
   This is harder than it seems because it is a function of the
   processor number. *)
var
   i,ad: integer,
begin
   ad .= 0;
   for i := 0 to pid-1 do
```

```
      begin
      dbad[i] := ad;writeln('dbad[i]=',dbad[i]);
      ad := ad+dbsize;writeln('ad =', ad);
      end;
   for i := pid+1 to maxprocessors do
      begin
      dbad[i] := ad;writeln('dbad[i] =', dbad[i]);
      ad := ad+dbsize;writeln('ad =', ad);
      end;
   dbad[pid] := ad; (* there isn't really one, but  .  *)
end; (* dbaddrs *)



procedure work;
   (* At startup, identify which processors are nominally working  *)
var
   i integer;
begin
   for i  = 0 to maxprocessors do datafile[dbad[i]]  = -1;
   writeln('dbad[i]=',dbad[i]);
(* wait(1);*)
   datafile[896] := pid;
   transfile[769] := 32768;
   transptr := 896 ;       (* initiate the broadcast. *)
(* wait(1);*)
   numworking := 1;
   for i := 0 to maxprocessors do
      if datafile[dbad[i]] = i then
           begin working[i] := true; numworking := numworking + 1;
           end
      else working[i] := false;
   working[pid]  = true;       (* I'm working *)
end; (* work *)



procedure synch;
const
   value = 43690;
var
   i,j integer;
begin (* At startup synchronize the processors  *)
   i := 7;
   while not working[i] do i := i-1;
   (* i points to the highest working processor  *)
   j := dbad[i];
   datafile[j] := 0;
   if i = pid then
      begin
   (* wait(1);*)
      datafile[896] := value;
      transfile[769] := 32768;
      transptr := 896;
      while pideof<0 do;
      end
```

```
   else while datafile[j]<>value do;
end; (* synch *)



procedure fail;
   (* All returned values are wrong, so report all processors involved.
   This could be failed inline, but it would take too much room.  The
   minor additional time that it takes to call the subroutine is
   probably worthwhile  *)
begin
   errors[p1]  = errors[p1]+1,
   errors[p2]  = errors[p2]+1;
   errors[p3]  = errors[p3]+1;
   errors[p4]  = errors[p4]+1;
   errors[p5]  = errors[p5]+1;
 (*if errtr then pause(43868)*)
end, (* fail *)



procedure err(p integer),
   (* Record an error for processor p.  *)
begin
   errors[p] := errors[p]+1,
(* if errtr then pause(43869)*)
end; (* err *)



function vote5(default:integer): integer,
   (* This is the five way voter.  It assumes that V1    V5 is
   initialized with the 5 values to be voted, and P1 .. P5
   has the corresponding processors.  Default is returned in the
   case that there is no majority value.  The procedure is basically
   a simple IF tree (pruned where possible) to achieve the quickest
   possible vote  *)
begin
   if v1 = v2 then
      if v1 = v3 then
         begin vote5  = v1;
           if v1 <> v4 then err(p4), if v1 <> v5 then err(p5);
           end
      else if v2 = v4 then
           begin err(p3); if v1 <> v5 then err(p5), vote5 := v1;
           end
      else if v1 = v5 then
           begin err(p3); err(p4); vote5 := v1;
           end
      else if v3 = v4 then
         if v3 = v5 then
              begin err(p1), err(p2); vote5  = v3;
              end
         else
```

```
                begin fail, vote5 := default,
                end
        else
             begin fail; vote5 := default;
             end
     else if v1 = v3 then
        if v1 = v4 then
             begin err(p2), if v1 <> v5 then err(p5); vote5 := v1;
             end
        else if v1 = v5 then
             begin err(p2); err(p4); vote5 := v1;
             end
       else if v2 = v4 then
           if v2 = v5 then
                begin err(p1); err(p3); vote5 := v2;
                end
             else
                begin fail, vote5 .= default,
                end
        else
             begin fail; vote5 := default,
             end
     else if v4 = v5 then
        if v2 = v4 then
             begin err(p1), if v2 <> v3 then err(p3); vote5 := v2,
             end
        else if v1 = v5 then
             begin err(p2); err(p3); vote5  = v1,
             end
        else if v3 = v5 then
             begin err(p1); err(p2), vote5 := v3,
             end
        else
             begin fail; vote5 := default,
             end
     else if v2 = v5 then
        if v2 = v3 then
             begin err(p1); err(p4), vote5 := v2,
             end
        else
             begin fail, vote5 := default;
             end
     else if v2 = v3 then
        if v2 = v4 then
             begin err(p1); err(p5); vote5 := v2,
             end
        else
             begin fail; vote5 := default,
             end
     else
        begin fail, vote5 := default;
        end,
end; (* vote5 *)



function vote3(default:integer):integer;
   (* This is the 3 way voter.  It assumes that V1 .  V3 contains
```

```
    the 3 values to be voted, and that P1 .. P3 contains the
    processors. *)
  begin
    if v1 = v2 then
       begin
       if v1<>v3 then err(p3);
       vote3 := v1;
       end
    else if v1 = v3 then
       begin err(p2); vote3  = v1,
       end
    else if v2 = v3 then
       begin err(p1), vote3  = v2;
       end
    else
       begin err(p1); err(p2); err(p3);
       vote3 := default,
       end,
  end, (* vote3 *)



procedure vote(b buffer;default integer);
var
   i, j, k:integer,
begin
   vtime := clock,
   (* vote buffer b  This involves either five way or three way voting. *)
(* if vottr then pause(16#ABC2),*)
   j := 0; i := 0;
   repeat
      k := bt[b].ad[i],
      if k >= 0 then
           begin
           j := j+1,
           p[j] := i,
           v[j] := datafile[k]
           end,
      i := i+1;
   until (j=5) or  (i>maxprocessors);
   if j < 3 then
      postvote[b] := v[1]
   else
      begin
      v1 := v[1]; v2 := v[2]; v3 := v[3],
      p1 := p[1]; p2  = p[2]; p3 := p[3];
      if j<5 then
           postvote[b] := vote3(default)
      else
           begin
           v4 := v[4]; v5 := v[5];
           p4 := p[4], p5 := p[5];
           postvote[b] := vote5(default)
           end;
      end,
   datafile[tpbase+b] := postvote[b];
end; (* vote *)
```

```
function getvote(b:buffer):integer;
   (* this phase two module lets us remove the postvote declaration
   from the applications task module *)
begin
    getvote := postvote[b];
end; (* getvote *)



procedure broadcast(b:buffer);
   (* Broadcast buffer b.  This is provided for applications tasks, and
   those executive tasks that don't do it themselves.   Note: this
   routine does not wait for completion before or after.  If that
   is required (for timing reasons) call waitbroadcast. *)
var
   dbx,tp integer;
begin
   dbx := bt[b].dbx; tp := dbx+tpbase;
   transfile[2*tp-1023] := bor(eofbit,dbx*8);
   transptr := tp;              (* initiate the broadcast. *)
   rewrite(datafile01);
   datafile01^ := b;
   put(datafile01);
end; (* broadcast *)



procedure stobroadcast(b,v:integer);
   (* Store v in buffer b and broadcast it  *)
var
   dbx,tp:integer;
begin
   dbx := bt[b].dbx; tp := dbx+tpbase; datafile[tp] := v;
   transfile[2*tp-1023] := bor(eofbit,dbx*8);
   transptr := tp;              (* initiate the broadcast. *)
end; (* stobroadcast *)



procedure waitbroadcast;
begin
   (* Wait for a broadcast operation to complete. *)
   while pideof<0 do;
end; (* waitbroadcast *)
```

```
procedure vschedule;
   (* Vote those items scheduled for this moment. *)
var
   k:integer;
begin
   k := scheds[vp];
   while k>0 do
      begin
      if (band(k,extendmask))= 0 then begin vote(k,-1), vp  = vp+1 end
      else if (framecount mod scheds[vp+1]+1) = scheds[vp+2] then
           begin
           vote(band(k,datamask),-1);
           vp := vp+3
           end;
      k  = scheds[vp]
      end; (* while *)
   if k >= 0 then vp := vp+1;
end, (* vschedule *)



procedure copschedule,
   (* Copy buffers scheduled for this moment. *)
var
   c,k:integer,
begin
   c := scheds[cptr];
   while c >= 0 do
      begin
      if (framecount mod scheds[cptr+2]) = scheds[cptr+3] then
         postvote[band(scheds[cptr+1],datamask)] := postvote[c];
      cptr  = cptr+4;
      c := scheds[cptr]
      end;
   if c >= 0 then cptr := cptr+1,
end, (* copschedule *)



procedure tschedule;
   (* Find the next task to schedule  *)
var
   tk,more:integer,
begin
   more := 0; tk := scheds[tp];
   while more = 0 do
      if tk = 0 then more := 1
      else if tk = -1 then more  = 1
      else if band(tk,extendmask) = 0 then more := 1
      else if (framecount mod scheds[tp+1]+1) = scheds[tp+2]
      then more  = 3
      else begin tp := tp+3; tk := scheds[tp] end,
   if tk <= 0 then begin taskid := nullt, taskbits := 0 end
   else
      begin
      taskid := band(tk,datamask);
```

```
      taskbits  = tk-taskid;
      (* if the repeat loop gets executed more than once, more should be 3. *)
      repeat
           tp  = tp+more
      until scheds[tp] = 0;
      end;
   if tk >= 0 then tp := tp+1
end; (* tschedule *)




procedure buildtask(taskname:integer);
   (* Initialize a task table entry *)
begin
(* reinit(tt[taskname].stkptr,tt[taskname].state);*)
   tt[taskname].status  = taskterminatıon
end; (* buildtask *)




function scheduler(ret1,ret2,oldpc:integer;
                        cause schedcall; state integer):integer;
var
   i:integer;
begin                    (* See large comment in file SCHED.BDX *)
   tskfn  = clock;
   tt[taskid].stkptr := state;
   if cause<>taskterminatıon then
      begin
      if (taskid<>nullt) then
           if taskid<>0 then
              if band(suspmask,taskbits) = 0 then
              begin
           (* pause(16#5500 bor taskid);*)
              tt[taskid].errors := tt[taskid].errors+1;
              buildtask(taskid)
              end
           else tt[taskid].status := clockinterrupt;
      if sfcount >= maxsubframe then
           begin
           if framecount >= maxframe then framecount := 0
           else framecount := framecount+1;
           gframe := gframe+1;
           sfcount := 0; vp := vpi; cptr  = cptri; tp  = tpi
           end
      else sfcount := sfcount+1;
      vschedule;
      (*  copschedule;  *)
      tschedule;    (* changes taskid and taskbits *)
      end
   else
      begin taskid := nullt; taskbits := 0;
      end;
   tskst  = clock;
   scheduler  = tt[taskid] stkptr;
```

```
end, (* scheduler *)


(* function nulltask: integer;  *)
   (* This is the task that wastes time   It never terminates   In  *)
(* the final system the nulltask will be the diagnostic task.     *)
(* begin*)
(* while true do _loop forever_       *)
(* end,    *)


function errtask: integer,
   (* Compute and broadcast a word with bits 7 through 0
   indicating whether processors 8 through 1 have
   failed (1) or are ok (0). *)
const
   threshold = 3,
var
   err, i: integer,
begin
   err  = 0, i := maxprocessors,
   repeat
      err := err*2,
      if (not working[i]) or (errors[i]>threshold) then err := err+1;
      errors[i]  = 0;
      i := i-1
      until i<0,
   stobroadcast(errerr, err);
   errtask := 0,
end, (* errtask *)




function gexectask integer;
   (* Compare values from the errtasks   Those that are reported
   by two or more processors (other than itself) for more than
   one frame, are considered bad.  The rest are considered good.
   The report consists of a word, bits 7 through 0 of which
   represent processors 8 through 1.  (1 failed, 0 working.)*)
var
   procs: procbool; err, i, j, count, reconf: integer;

begin
   i := 0;
   repeat
      count  = 0; j  = 0;
      repeat
           if working[j] then
              begin
              err := bt[errerr] ad[j];
              if i<>j then
                 if working[i] then
                  if odd(datafile[err]) then count := count+1;
              datafile[err] := datafile[err] div 2
              end;
```

```
            j := j+1
         until j>maxprocessors;
         if working[i] then
            if count>1 then procs[i] := false
            else procs[i] := true
         else procs[i] := false;
         i := i+1;
      until i>maxprocessors;
      reconf := 0;
      i := maxprocessors;
      repeat
         reconf := reconf*2;
         if not procs[i] then reconf := reconf+1;
         i := i-1
      until i<0;
      stobroadcast(gexecreconf,band(reconf,postvote[gexecmemory]));
      waitbroadcast;
      stobroadcast(gexecmemory,reconf);
      gexectask := 0
   end; (* gexectask *)



   procedure clrbufs;
      (* Set the buffer table so that no assumptions are made about what
      processor is computing the value. *)
   var
      i,j:integer;
   begin
      i := 0;
      repeat
         j := 0;
         repeat bt[i].ad[j] := -1; j := j+1 until j>maxprocessors;
         i := i+1
      until i>maxbufs;
   end; (* clrbufs *)



   procedure newvc(s:integer);
      (* S points to the vote and copy schedules  Copy them into
      the real schedules. *)
   begin
      s := s+3; vpi := s;
      while scheds[s] >= 0 do s := s+1;
      cptri := s+1
   end; (* newvc *)



   procedure recbufs(s,p:integer);
      (* s points to the task schedule corresponding to real processor p.
      Figure out what buffers are computed. *)
   var
```

```
    b,t,i:integer;
begin
    s := s+3;
    while scheds[s]<>-1 do
        if scheds[s] = 0 then s := s+1
        else
            begin
            t := band(scheds[s],datamask);
            i  = tt[t].bufs; b  = binf[i],
            t := dbad[p];
            while b>0 do
               begin
               with bt[b] do ad[p] := t+dbx;
               i := i+1, b := binf[i];
               end;
            if (band(scheds[s],extendmask))<>0 then s := s+3
            else s := s+1,
            end,
end; (* recbufs *)



function xrecf(reconf:integer):integer;
var
    i,s,nw.integer,
begin                    (* See big comment in file RECF BDX *)
    bclock:=clock;
(* disable,*)
    nw := -1; i  = 0; s  = reconf,
    repeat
        if odd(s) then working[i] := false
        else
            begin
            working[i] := true;
            nw := nw+1;
            virtno[nw] := i
            end;
        s := s div 2,
        i := i+1;
    until i>maxprocessors;
    if lastconfig<>reconf then
        begin
  (*    if rectr then pause(16#ABC4);*)
        lastconfig := reconf;
        datafile[tpbase+qreconf] := reconf,
        s := 0;
        (*if nw>5 then nw := 5;                                    phase 1*)
        while scheds[s]<>nw do s  = s+scheds[s+2];
        clrbufs; i := 0,
        tpi:=0,
        repeat
            recbufs(s,virtno[i]),
            if virtno[i] = pid then tpi := s+3;
            s := s+scheds[s+2];
            i := i+1
        until i>nw;
(*      if tpi=0 then pause(16#ABC5);*)
        newvc(s),
```

```
      numworking := nw+1;
      sfcount := maxsubframe+1;
      framecount := maxframe+1
      end;
   clock =bclock,
   xrecf := 0
end;



function recftask.integer,
   (* The reconfiguration task calls xrecf to do the real work. *)
begin
   recftask := xrecf(postvote[gexecreconf])
end; (* recftask *)



function clktask: integer;
const
   maxskew = 40,
   commdelay = 24;
   dest = 32768,                          (* Destination 0 *)

var
   i,num,sum,term,x  integer,
   delta,epsilon: integer;
   unseen: array[0  maxprocessors] of boolean;
   wkset  integer;

begin
 (*disable;*)
   for i := maxprocessors downto 0 do datafile[dbad[1]] := 0;   (* dp *)
   bclock := clock;     (* begin time *)
   wkset := 0,
   transfile[magic] := dest;   (* once is enough *)
   unseen[pid] := false;
    for i  = maxprocessors downto 0 do
      begin
      skew[i] := 0,
      while (band (clock,32))<> 0 do ;
      while (band (clk1,32))=  0 do ;   (* whoa mule *)
      teatime:=clock;
      if i = pid then
           repeat                                    (* the Broadcast *)
              if pideof>0 then
                 begin
                 datafile[tpbase]:=clock;
                 transptr:=tpbase,
                 end;
           until clock-teatime > maxtime
      else
         begin
           unseen[i] := true;
           x:=dbad[i];
           pclock := datafile[x];
```

```
            repeat
               cclock := datafile[x];
               aclock:=clock;
               if cclock <> pclock then
                  begin
                  skew[i]:= cclock + commdelay - aclock;
                  unseen[i].=false;
                  repeat
                  until clock - teatime > maxtime;
                  end;
            until clock-teatime > maxtime
            end;
      end,
   (* Calculate the clock correction  *)

(* enable,*)

   sum .= 0;   num .= 0;
   for i .= 0 to maxprocessors do
      begin
      wkset := 2*wkset;
      sk[skptr+i]  = skew[i];
      if working[i] then
         begin
           wkset  = wkset+1;
           term := skew[i],
           if term >  maxskew then
            begin
              term :=  maxskew;
              reason  = reason+1,
            end,
           if term < -maxskew then
            begin
              term := -maxskew,
              reason := reason+1,
            end;
           if unseen[i] then
            begin
              term  = 0,
              fatal .= true;
              reason .= bor(reason,1024);
            end,
           sum  = sum+term;
           num  = num+1;
         end
      end;
   delta  = (sum div num),

   (*  lets wait for the 1.6 msec interrupt  *)

   repeat
      cclock := band(clock,1023),
   until (cclock > 512 + maxskew) OR (cclock < 271),

   cclock := delta+clock,
   clock := cclock;            (* Adjust the clock value.  *)
   pclock := clock;

   epsilon .= pclock-cclock,
   if (epsilon > 2) or (epsilon < -2) then
```

```
    reason .= bor(reason,2048),

    sk[skptr+pid] := 43680+pid,
    sk[skptr+maxprocessors] := wkset;
    sk[skptr+8]   = gframe,
    sk[skptr+9]   := postvote[gexecreconf];
    sk[skptr+10] := postvote[gexecmemory];
    sk[skptr+11]  = bclock;
    sk[skptr+12] := delta;
    sk[skptr+13] := runid;
    sk[skptr+14] .= pages;
    sk[skptr+15] := reason;
    skptr  = skptr+16;

    lines := lines+1,
    if fatal then stop:=stop+1;
    if hlttrue then
     (*if stop=3 then pause(16#555);*)
    if lines > 48 then
       begin
       lines := 1;
       pages  = pages+1,
       if hlttrue then
            begin
       (*   if reason > 16#FF then pause(16#333),*)
       (*   if pages > pagelimit then pause(16#444);*)
            end,
       skptr := 0
       end; (* if lines > 48 *)

    clktask .= 0;
    (* enable   dp  *)
end; (* clktask *)




(* The following routines have to do with system initialization. *)
(* ------------------------------------------------------------- *)



procedure initialize;
    (* initialize the processor numbers and pointers and errors *)
var
    i,j,reconf.integer;
begin

    skptr := 0;  lines := 1;  pages := 1;
    reason := 0;      fatal := false;  stop:=0;
    if (minutes > 1354) or (minutes < 1) then
       minutes := 1354;
    pagelimit := 24*minutes+((6*minutes) div 31);

    sk[800]  = pagelimit,  sk[801] := runid;
    votecnt .= 0;
    errtr .= errtrb; vottr .= vottrb; rectr := rectrb;
    hlttrue.=hlttb,
(* gprocessor ;*)  dbaddrs, work, synch;
```

```
    taskbits := 0; lastconfig := 0; reconf  = 0, gframe .= -1;
    taskid .= 0; sfcount := -1; framecount .= 0, clock  = 0;
    for i := 0 to maxbufs do postvote[i] := 0,
    for i  = 0 to tasks do
       begin buildtask(i), tt[i] errors := 0
       end;
    for i .= maxprocessors downto 0 do
       begin
       errors[i] := 0,
       reconf := reconf*2;
       if not working[i] then reconf  = reconf+1
       end,
(* appinit;*)
(* icinit,*)
postvote[gexecmemory] := reconf,
i .= xrecf(reconf);
end;
begin
read(pid);
writeln(pid),
rewrite(datafile01);
datafile01^ := pid,
put(datafile01);
writeln(pid),
doaddrs;
work,
synch,
initialize;
end.
```

Appendix - II

```
siftpl : proc;

dcl sysin                file;
dcl sysprint             file;
dcl newsift              entry options (variable);
dcl dfile(0:1015)        fixed bin (35) based (dfptr);
dcl dfptr                ptr;
dcl dfptrpas             ptr external static;
dcl clk                  fixed bin external static;
dcl transfile(0:1015)    fixed bin(35) based (transptr);
dcl transptrpas          ptr external static;
dcl transptr             ptr;
dcl pideof               fixed bin(35) external static;
dcl scheds_pl (0:1791)   fixed bin(35) external static;
dcl schedsptrpas         ptr  external static;
dcl schedsptr            ptr;
dcl postvote_pl(0:119)   fixed bin(35) based(postptr);
dcl postptr              ptr;
dcl postptrpas           ptr external static;
dcl dbad_pl(0:7)         fixed bin(35) based(dbadptr);
dcl dbadptr              ptr;
dcl dbadptrpas           ptr external static;
dcl binf_pl(0:200)       fixed bin(35) based(binfptr);
dcl binfptr              ptr;
dcl binfptrpas           ptr external static;
dcl clock                fixed bin(35) external static;
dcl adr1553a             fixed bin(35) external static;
dcl cmd1553a             fixed bin(35) external static;
dcl sta1553a             fixed bin(35) external static;
dcl pid                  fixed bin(35) external static;
dcl numworking           fixed bin(35) external static;
dcl virtno(0:7)          fixed bin(35) external static;
dcl virtnoptrpas         ptr external static;
dcl virtnoptr            ptr;
dcl gframe               ptr external static;
dcl sfcount              fixed bin(35) external static;
dcl votecnt              fixed bin(35) external static;
dcl vtime                fixed bin(35) external static;
dcl wtime                fixed bin(35) external static;
dcl delta                fixed bin(35) external static;
dcl working(0:7)         fixed bin(35) based (workingptr) ;
dcl workingptr           ptr;
dcl workptrpas           ptr external static;
dcl p_pl                 fixed bin (35) based (pptr);
dcl pptrpas              ptr external static;
dcl pptr                 ptr;
dcl v_pl(0:7)            fixed bin(35) based(vptr);
dcl vpas                 ptr external static;
dcl vptr                 ptr ;
dcl errors(0:7)          fixed bin(35) based (errorptr);
dcl errorpas             ptr external static;
dcl errorptr             ptr ;
dcl P1                   fixed bin(35) external static;
dcl p2                   fixed bin(35) external static;
dcl p3                   fixed bin(35) external static;
dcl P3                   fixed bin(35) external static;
dcl p4                   fixed bin(35) external static;
dcl p5                   fixed bin(35) external static;
dcl v1                   fixed bin(35) external static;
```

```
dcl v2                  fixed bin(35) external static;
dcl v3                  fixed bin(35) external static;
dcl v4                  fixed bin(35) external static;
dcl v5                  fixed bin(35) external static;
dcl taskid_pl(0:12)     fixed bin(35) based (taskidptr);
dcl taskidpas           ptr external static;
dcl taskidptr           ptr;
dcl skew_pl(0:7)        fixed bin(35) based(skewptr);
dcl skewptr             ptr;
dcl skewptrpas          ptr external static;
dcl sk_pl               fixed bin(35) based(skptr);
dcl skptr               ptr;
dcl skptrpas            ptr external static;
dcl taskbits            fixed bin(35) external static;
dcl lastconfig          fixed bin(35) external static;
dcl pclock              fixed bin(35) external static;
dcl cclock              fixed bin(35) external static;
dcl bclock              fixed bin(35) external static;
dcl aclock              fixed bin(35) external static;
dcl tp                  fixed bin(35) external static;
dcl vp                  fixed bin(35) external static;
dcl cptrl               fixed bin(35) external static;
dcl tpi                 fixed bin(35) external static;
dcl vpi                 fixed bin(35) external static;
dcl cptri               fixed bin(35) external static;
dcl framecount          fixed bin(35) external static;
dcl teatime             fixed bin(35) external static;

dcl lines               fixed bin(35) external static;
dcl pages               fixed bin(35) external static;
dcl pagelimit           fixed bin(35) external static;
dcl reason              fixed bin(35) external static;
dcl fatal               bit external static;
dcl hlttrue             bit external static;
dcl errtr               bit external static;
dcl vottr               bit external static;
dcl rectr               bit external static;
dcl runid               fixed bin(35) external static;
dcl minutes             fixed bin(35) external static;
dcl tskst               fixed bin(35) external static;
dcl tskfn               fixed bin(35) external static;
dcl hlttb               fixed bin(35) external static;
dcl errtrb              fixed bin(35) external static;
dcl vottrb              fixed bin(35) external static;
dcl rectrb              fixed bin(35) external static;
dcl stop                fixed bin(35) external static;
```

```
bor : proc (a,b) returns (fixed bin(35));

dcl a                   fixed bin(35);
dcl b                   fixed bin(35);
dcl c                   bit (16) init("0000000000000000"b);
dcl temp1               bit (16) ;
dcl temp2               bit (16) ;
dcl i                   fixed bin;

temp1 = substr(unspec(a),21);
temp2 = substr(unspec(b),21);

do i = 1 to 16;
   if substr(temp1,i,1) = "1"b | substr(temp2,i,1) = "1"b
   then substr(c,i,1) = "1"b;
end;

return (fixed (c));

end /* bor */;

band : proc(a,b) returns (fixed bin(35));

dcl a                   fixed bin(35);
dcl b                   fixed bin(35);
dcl c                   bit (16) init("0000000000000000"b);
dcl temp1               bit (16);
dcl temp2               bit (16);
dcl i                   fixed bin;

temp1 = substr(unspec(a),21);
temp2 = substr(unspec(b),21);

do i = 1 to 16;
   if substr(temp1,i,1) = "1"b & substr(temp2,i,1) = "1"b
   then substr(c,i,1) = "1"b;
end;

return (fixed (c));

end /* band */;
```

```
dfptr = dfptrpas;
transptr = transptrpas;
schedsptr = schedsptrpas;
postptr = postptrpas;
dbadptr = dbadptrpas;
binfptr = binfptrpas;
workingptr = workptrpas;
pptr = pptrpas;
vptr = vpas;
errorptr  = errorpas;
taskidptr = taskidpas;
skewptr = skewptrpas;
skptr = skptrpas;
/* call newsift;*/

put skip list ("clock =",clock);
put skip list("dfile = ", dfptr -> dfile(0),dfptr -> dfile(128));

end;
```

Appendix - III