University of Cape Town
Department of Computer Science

Parallelisation of Algorithms

by

Alexander Marius Schuilenburg

M.Sc Thesis Supplement

September, 1990

CONTENTS

Chapter 1 - Introduction 3

Chapter 2 - 3L Flood Fill Bug 4

Chapter 3 - SAPF Farm Libraries 10

Chapter 4 - OCCAM/Turbo Pascal Server 24

(i) Turbo Pascal Source 24

(ii) Occam Source 76

Chapter 5 - Airflow Model 97

Chapter 6 - Depth First Worm 135

The Head - Listing (i) 135

The Tail - Listing (ii) 141

Chapter 7 - Breadth First Worm 149

Identify Link Procedure - Listing (i) 149

The Mouth - Listing (ii) 152

The Foot - Listing (iii) 156

Chapter 8 - SAPF Source Listing 162

(i) SAPF Source 162

(ii) Relate Source 215

(iii) MAKECONF Source 218

Chapter 1 - Introduction

This document serves as a supplement to the M.Sc thesis of Alexander Marius Schuilenburg entitled *Parallelisation of Algorithms*. The thesis was submitted in fulfilment of the degree of Master of Science in the Department of Computer Science, University of Cape Town, under the supervision of Dr I. M. A. Gledhill and Prof P. S. Kritzinger.

The thesis involved the parallelisation of algorithms, amongst other aspects, and this thesis supplement is used to document the source code of most of the algorithms, as well as to include documents attaining to bugs discovered in commercially available products as well as the tools and other utilities created by the author in the development of the thesis. These source listings and other documentation were omitted from the body of the thesis in order to present a managable thesis with only immediately relevant documents appearing in the appendices. The thesis supplement thus provides a listing of the omitted documents in order to make these available to those who wish to continue research in these areas, as well as to provide a reference for those who simply wish to know how certain objectives were acheived and how the utilities and tools developed operate.

This supplement contains all the source listings, apart from that of //BSCAT. This listing is in over 9000 lines of Fortran code and may be obtained, with valid reason, by writing to the author directly.

Chapter 2 - 3L Flood Fill Bug

Summary of document sent to 3L to identify Flood-Fill problem

I have a small network of four T800 transputers in the form of the Microway Quadputer, with 1MB each, as well as the INMOS B004 board with a T414 and 2MB. They have been connected together as indicated in Figure 1.

I have been designing an application which would execute on a processor farm and was disappointed with the speed of execution on the system in Figure 1 as compared to that of a second network of three T800s with 1MB each connected as shown in Figure 2. After some investigation and debugging I discovered that one of the T800s on the quadputer was not being used as a worker. After some experimentation I learned that the code generated by the flood-fill configurer would either not always boot all the transputers (apart from the master) on the network as workers or route work to a free worker when there was a free worker available. This was disappointing since the only way to ensure all the transputers were made use of one had to write fixed configuration code with message passing/routing threads which is a lengthy and tedious process for larger networks (such as the third network of 40 T800s for which I also generate code).

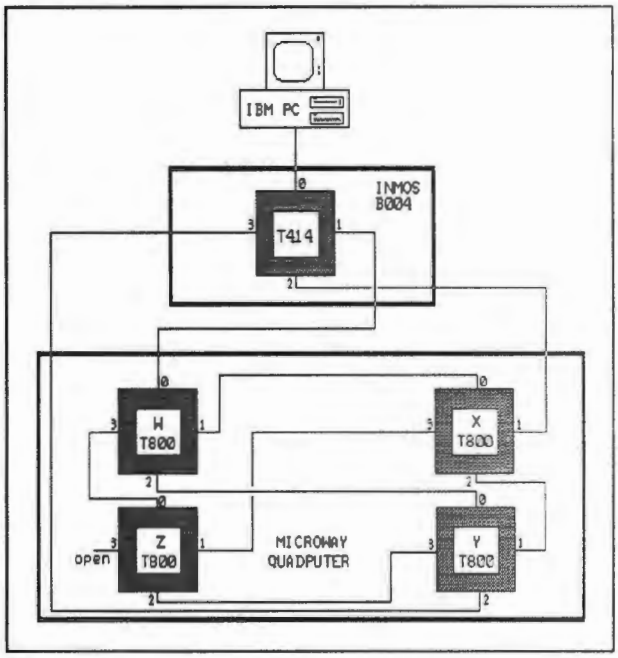

Figure 1

Since I produce a high volume of code for which the processor farm best suits the large majority of code topologies, and am forced to write network specific code making code transfer from one network configuration to the next tedious. I would therefore greatly appreciate it if you could suggest a method by which I could overcome this problem with FCONFIG, or if you could please send me a corrected version of the flood-fill configurer. I have already tried the FCONFIG supplied with your parallel C, however, this appears to have the same problem.

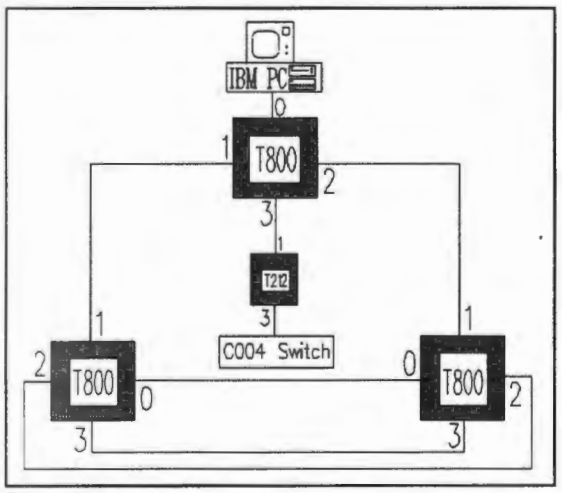

Figure 2

One rather unusual method of eliminating the problem was to either disconnect the link between link 1 of the B004 and link 0 of transputer W or to disconnect link 2 of the B004 and link 1 of transputer X. In these two cases all five transputers would be used with the T414 as the master and the four T800s as the worker. All code was compiled for the T414. If the third link to the quadputer was disconnected with the former being connected, or if any two of the links were disconnected, the same problem would occur (one of the four would not be booted or be given any work).

Once I determined there was nothing wrong with the quadputer or the B004 I wrote a small demonstration network to test the flood-fill configurer and ascertain the number of transputers being used. What follows is a discussion of the master and worker tasks, as well

as the output obtained[1]. The output listing represents the screen output obtained with some intermediary values omitted, as illustrated by lines of dots.

The Master

The master consists of three threads. The first merely initialises the second and third. The second thread is used to send the sequence 100, 200, 400, 800, 1600, ... 26214400, 52428800 of 32 bit integers to the workers where each number in the sequence represents a work packet. The third thread is used to receive and display the result packets from each workers as they arrive. These result packets are in the form of a two sequences of 32 bit integers, the first is terminated by an integer whose decimal representation ends with the digits 99, while the second is passed back in a 32 bit integer array where the amount of numbers in the sequence is implicit in the size of the array.

The Workers

Each worker maintains a cumulative 32 bit integer total with an array index, both initially 0, and a 32 bit integer array. The work packet is received as a 32 bit integer which is placed in the array, indicated by the index (which is first incremented), and then added to the cumulating total. During these operations, arbirary arithmetic operations are repeatedly performed in order to provide a time delay. The result packet it returns is the sequence produced by adding the cumulative total to each of the numbers in the sequence (1,2,3,4,5,6,7,8,9,10,99), followed by the partial array containing all the integer work packets received.

Results

The flood-fill nework produced was executed on the network illustrated in Figure 1. The output produced is also listed below, although it has been divided up into three columns, and shows only three workers are present since three groups of work results are evident. This may be determined from the fact that all 20 tasks are done by the transputers which receive 8, 2 and 1 as their initial tasks respectively. There is no possibility that one of these three becomes free before the forth can be given a task, with the new task being re-routed to the recently freed worker, as a delay is present in each worker before the worker is allowed to return its results and this delay is long enough for all the work packets to be completely transmitted by the master.

[1] Three types of tests were performed, although the last two have been omitted in this summary

Listings

```
1  C
2  C This is the master program which distributes the work packets 100, 200, 400,
3  C 800, 1600, 3200, 6400, 12800, 25600, ...., 26214400 and 52428800.  The workers
4  C continually return integer results which are displayed as they are received
5  C from the workers.  This has been done in an effort to determine how the
6  C messages are sent to the master and if it is possible for workers to return
7  C intermediate results or to make requests from the master to which the master
8  C may respond.
9  C
10 C The master task consists of three threads:
11 C  1) The main thread, which simply initialises the tasks
12 C  2) The send thread which simply sends the numbers indicated.
13 C  3) The receive thread which prints the number as they are received from the
14 C     workers.
15 C
16 C ****************************************************************
17 C The SEND subroutine
18 C
19        SUBROUTINE SEND(DUMMY)
20        IMPLICIT NONE
21        INTEGER DUMMY
22        INCLUDE 'NET.INC'
23        INTEGER I, TMP, CNT
24 C
25        CNT = 1
26        DO I=1,20
27           TMP = 100*CNT
28           CNT = CNT*2
29           CALL F77_NET_SEND(4,TMP,.TRUE.)
30        END DO
31        END
32 C
33 C ****************************************************************
34 C The RECEIVE subroutine
35 C
36        SUBROUTINE RECEIVE(DUMMY)
37        IMPLICIT NONE
38        INTEGER DUMMY
39        INCLUDE 'NET.INC'
40        INCLUDE 'THREAD.INC'
41        INTEGER I, TMP, CNT, A(100)
42        LOGICAL COMPLETE
43 C
44        CNT = 0
45 100    CONTINUE
46 C
47 C      Receive and print number
48        CALL F77_NET_RECEIVE(TMP,I,COMPLETE)
49        CALL F77_THREAD_USE_RTL
50        WRITE (*,*) I
51 C
52 C      If number + 99 then this is the last number so expect the array
53 C      indicating numbers received.
54        IF ((I - (I/100)*100).EQ.99) THEN
55           CALL F77_NET_RECEIVE(TMP,A,COMPLETE)
56           CNT = CNT + 1
57 C
58 C         Display the numbers received by this process
59           DO I=1,TMP/4
60             WRITE (*,99) A(I)/100, CNT
61 99          FORMAT(I10,'      ',I3)
62           END DO
63        END IF
64        CALL F77_THREAD_FREE_RTL
65        GOTO 100
66 C
67        END
68 C
```

```
C ****************************************************************
C  THE MAIN PROGRAM
C
      PROGRAM MAIN
      IMPLICIT NONE
      INCLUDE 'THREAD.INC'
      EXTERNAL SEND, RECEIVE
      INTEGER WS_SIZE, DUMMY
      PARAMETER (WS_SIZE=2500)
      INTEGER SEND_WS(WS_SIZE), RECEIVE_WS(WS_SIZE)
C
C     Start SEND and RECEIVE
      CALL F77_THREAD_START(SEND,SEND_WS,WS_SIZE*4,
     /                      F77_THREAD_URGENT, 1, DUMMY)
      CALL F77_THREAD_START(RECEIVE,RECEIVE_WS,WS_SIZE*4,
     /                      F77_THREAD_URGENT, 1, DUMMY)
C
C     Terminate
      CALL F77_THREAD_STOP
C
      END
C
C
C ****************************************************************
C This program is the test worker I have set up to simply get an integer value
C ADD from the master and store it, then add it to a running total ID.  The
C worker then must return ID with the values 1 to 10 added to it consecutively
C to the master.  Once all 10 values have been sent, ID value plus 99 is sent
C to indicate the worker list of numbers is complete and the list of numbers
C received by this worker is then returned to the master as the last message of
C the task.  The worker then gets the next value from the master.
C
      PROGRAM WORKER
      IMPLICIT NONE
      INCLUDE 'NET.INC'
      INTEGER add, ID, I, J, II, TMP, A(100)
      LOGICAL COMPLETE
C
      ID = 0
      II = 1
100   CONTINUE
C
C     Task waits here for initial message from master
      CALL F77_NET_RECEIVE(I,add,COMPLETE)
C
C     Arbitrary loop to cause a delay in this process
      A(II) = add
      ID = ID + add
      TMP = 0
      DO J = 1,1000000
        TMP = TMP - J
        TMP = TMP + (J+I)
      END DO
C
C     Send the numbers required to the master
      DO I = 1,10
        TMP = ID + I
        CALL F77_NET_SEND(4,TMP,.FALSE.)
      END DO
C
C     Send the number to indicate end of list and beginning of
C     numbers received list.
      TMP = ID + 99
      CALL F77_NET_SEND(4,TMP,.FALSE.)
C
C     Send the array of numbers received from the master
      CALL F77_NET_SEND(4*II,A,.TRUE.)
      II = II+1
C
      GOTO 100
      END
```

Output

```
801
802
.....
810
899
8 1

201
202
.....
210
299
2 2

101
102
.....
110
199
1 3

601
602
.....
610
699
2 4
4 4

4001
4002
.....
4010
4099
8 5
32 5

51801
51802
.....
51810
51899
2 6
4 6
512 6

1701
1702
.....
1710
1799
1 7

16 7

256601
256602
.....
256610
256699
2 8
4 8
512 8
2048 8

16801
16802
.....
16810
16899
8 9
32 9
128 9

1075801
1075802
.....
1075810
1075899
2 10
4 10
512 10
2048 10
8192 10

8101
8102
.....
8110
8199
1 11
16 11
64 11

4352601
4352602
.....
4352610
4352699
2 12
4 12
512 12
2048 12
8192 12
32768 12

119201
119202
.....
119210
119299
8 13
32 13
128 13
1024 13

17459801
17459802
.....
17459810
17459899
2 14
4 14
512 14
2048 14
8192 14
32768 14
131072 14

33701
33702
.....
33710
33799
1 15
16 15
64 15
256 15

69888601
69888602
.....
69888610
69888699
2 16
4 16
512 16
2048 16
8192 16
32768 16
131072 16
524288 16
```

```
1757601
1757602
.............
1757610
1757699
      8       17
     32       17
    128       17
   1024       17
  16384       17

 443301
 443302
 ...........
 443310
 443399
      1       18
     16       18
     64       18
    256       18
   4096       18

27972001
27972002
.............
27972010
27972099
      8       19
     32       19
    128       19
   1024       19
  16384       19
 262144       19

 6996901
 6996902
 ...........
 6996910
 6996999
      1       20
     16       20
     64       20
    256       20
   4096       20
  65536       20
```

Chapter 3 - SAPF Farm Libraries

SAPF Farm Libraries

There are three files which comprise the SAPF Farm Libraries. The first is an include file **NETW.INC** which must be included by any source code file in which the code references any of the library routines. The second file **NETWMAST.F77** contains the listing of the source code all the routines described in Appendix A.1 of the thesis while the third file **NETWWORK.F77** contains the listing of the source code for all the routines described Section 6.3.4 and Appendix A.2 of the thesis. The latter two files are used to create libraries which are used by the farm master and farm workers respectively. These libraries should be linked to the corresponding code of the master and worker object codes respectively. That is, the master object code should be linked to the library code produced by the code in NETWMAST.F77 and the worker object code should be linked to the library produced by the code in NETWWORK.F77.

The listings are:

NETW.INC

```
1  C
2  C CONSTANT DEFINITIONS FOR THE SUBROUTINES AND FUNCTIONS USED BY THE NEWTORK
3  C HARNESS GENERATED BY "MAKECONF".
4  C
5        INTEGER F77_NETW_MAX_WORKERS, F77_NETW_MAX_PACKET_LEN,
6       1        F77_NETW_NUM_WORKERS, F77_NETW_RESERVED,
7       2        F77_NETW_USE_WORKER, F77_NETW_FREE_WORKERS
8        PARAMETER
9       1  (F77_NETW_MAX_WORKERS = 64,
10      2   F77_NETW_MAX_PACKET_LEN = 1024)
```

NETWMAST.F77

```
C ================================================================================
C This is the block data used by the network subroutine and functions
C
      BLOCK DATA
      IMPLICIT NONE
      INCLUDE 'SEMA.INC'
      COMMON /F77_NETW_DATA/
     1       NumChans, INAddr(4), OUTAddr(4), Path_Length(64),
     2       Path(64), NumWorkers, FreeWorkers, Initialised,
     3       Reserved_Chan, Reserved_Worker,
     4       Workers_Free(F77_SEMA_SIZE), Workers_Sema(F77_SEMA_SIZE),
     5       Worker_Busy(64), Worker_Ptr
C
      INTEGER NumChans, INAddr, OUTAddr, Path_Length,
     1        NumWorkers, FreeWorkers, Reserved_Chan, Reserved_Worker,
     2        Workers_Free, Workers_Sema, Worker_Ptr
      CHARACTER*32 Path
      LOGICAL Worker_Busy, Initialised
      DATA Reserved_Worker /0/  Worker_Ptr /1/  Initialised /.FALSE./
      END
C
C ================================================================================
C This subroutine initialises the network from the master.
C It gets the number of workers (a maximum of 64) and the path (a maximum
C length of 32) to each of these workers.  It also sets the addresses of
C the ports, leaving port 1 for communication to the filer, and using ports
C in the order 0,2,3,4 for communication to workers.  It is designed to be
C used with the configuration file generator MAKECONF.
C
      SUBROUTINE F77_NETW_MASTER()
      IMPLICIT NONE
      INCLUDE 'CHAN.INC'
      INCLUDE 'SEMA.INC'
C
      COMMON /F77_NETW_DATA/
     1       NumChans, INAddr(4), OUTAddr(4), Path_Length(64),
     2       Path(64), NumWorkers, FreeWorkers, Initialised,
     3       Reserved_Chan, Reserved_Worker,
     4       Workers_Free(F77_SEMA_SIZE), Workers_Sema(F77_SEMA_SIZE),
     5       Worker_Busy(64), Worker_Ptr
C
      INTEGER NumChans, INAddr, OUTAddr, Path_Length,
     1        NumWorkers, FreeWorkers, Reserved_Chan, Reserved_Worker,
     2        Workers_Free, Workers_Sema, Worker_Ptr
      CHARACTER*32 Path
      LOGICAL Worker_Busy, Initialised
C
      INTEGER I, J, NextID, RID
C
      LOGICAL Initialise
C
C     Ensure that this is called only once
      IF (Initialised) GOTO 170
C
C     Initialise addresses to all channels (1 is reserved for filer comms)
      NumChans = F77_CHAN_IN_PORTS() - 1
      DO I = 1,NumChans
        IF (I.GE.2) THEN
          INAddr(I)  = F77_CHAN_IN_PORT(I)
          OUTAddr(I) = F77_CHAN_OUT_PORT(I)
        ELSE
          INAddr(I)  = F77_CHAN_IN_PORT(I-1)
          OUTAddr(I) = F77_CHAN_OUT_PORT(I-1)
        END IF
      END DO
C
C     Set the next available ID
      NextID = 1
C
```

```
C     Send IDs to all workers, channel for channel until no
C     more IDs can be sent
      DO 150 I = 1,NumChans
        Initialise = .TRUE.
        DO 140 WHILE (Initialise)
C         Send ID
          CALL F77_CHAN_OUT_WORD(NextID,OUTAddr(I))
C         Get Result
          CALL F77_CHAN_IN_WORD(RID,INAddr(I))
          Initialise = (RID.NE.-1)
          IF (Initialise) THEN
C           Get Path
            CALL F77_CHAN_IN_WORD(path_length(NextID),INAddr(I))
            IF (path_length(NextID).NE.0)
     1        CALL F77_CHAN_IN_MESSAGE(path_length(NextID),
     2                                 path(NextID)(2:32),INAddr(I))
            path_length(NextID) = path_length(NextID) + 1
            path(NextID)(1:1) = CHAR(I)
            NextID = RID
          END IF
  140   CONTINUE
  150 CONTINUE
C
C     Set the number of workers
      NumWorkers = NextID - 1
      FreeWorkers = NumWorkers
C
C     Initialise the worker array
      DO 160 I = 1,NumWorkers
        Worker_Busy(I) = .FALSE.
  160 CONTINUE
C
C     Initialise the semaphores
      CALL F77_SEMA_INIT(Workers_Free,FreeWorkers)
      CALL F77_SEMA_INIT(Workers_Sema,1)
C
      Initialised = .TRUE.
  170 RETURN
      END
C
C ============================================================================
C This function will return the number of workers in the network.
C
      INTEGER FUNCTION F77_NETW_NUM_WORKERS()
      IMPLICIT NONE
      INCLUDE 'SEMA.INC'
C
      COMMON /F77_NETW_DATA/
     1       NumChans, INAddr(4), OUTAddr(4), Path_Length(64),
     2       Path(64), NumWorkers, FreeWorkers, Initialised,
     3       Reserved_Chan, Reserved_Worker,
     4       Workers_Free(F77_SEMA_SIZE), Workers_Sema(F77_SEMA_SIZE),
     5       Worker_Busy(64), Worker_Ptr
C
      INTEGER NumChans, INAddr, OUTAddr, Path_Length,
     1        NumWorkers, FreeWorkers, Reserved_Chan, Reserved_Worker,
     2        Workers_Free, Workers_Sema, Worker_Ptr
      CHARACTER*32 Path
      LOGICAL Worker_Busy, Initialised
C
      F77_NETW_NUM_WORKERS = NumWorkers
C
      RETURN
      END
C
```

```
C ============================================================================
C This procesure broadcasts the message passed to it to all the workers.
C Variables passed : 1) Length  = Length (in bytes) of message to pass
C                    2) Buffer  = Pointer to start of message buffer to
C                                 be sent.
C
      SUBROUTINE F77_NETW_BROADCAST(Length, Buffer)
      IMPLICIT NONE
      INCLUDE 'CHAN.INC'
      INCLUDE 'SEMA.INC'
      INCLUDE 'THREAD.INC'
      INCLUDE 'NETW.INC'
C
      INTEGER Length, Port
      CHARACTER*(*) Buffer
C
      COMMON /F77_NETW_DATA/
     1       NumChans, INAddr(4), OUTAddr(4), Path_Length(64),
     2       Path(64), NumWorkers, FreeWorkers, Initialised,
     3       Reserved_Chan, Reserved_Worker,
     4       Workers_Free(F77_SEMA_SIZE), Workers_Sema(F77_SEMA_SIZE),
     5       Worker_Busy(64), Worker_Ptr
C
      INTEGER NumChans, INAddr, OUTAddr, Path_Length,
     1        NumWorkers, FreeWorkers, Reserved_Chan, Reserved_Worker,
     2        Workers_Free, Workers_Sema, Worker_Ptr
      CHARACTER*32 Path
      LOGICAL Worker_Busy, Initialised
C
      INTEGER OutPort, PL
C
C     Send along all output ports
      DO 500 Port = 1,NumChans
        OutPort = OutAddr(Port)
C
C       Send the broadcast header
        CALL F77_CHAN_OUT_WORD(-1,OutPort)
C
C       Send the buffer
        CALL F77_CHAN_OUT_WORD(Length,OutPort)
        CALL F77_CHAN_OUT_MESSAGE(Length,Buffer,OutPort)
C
500   CONTINUE
      RETURN
      END
C
C ============================================================================
C This procesure sends the message passed to it to the worker number passed.
C Variables passed : 1) Length  = Length (in bytes) of message to pass
C                    2) Buffer  = Pointer to start of message buffer to
C                                 be sent.
C                    3) Worker  = Integer number representing worker
C                                 number to pass to.
C
      SUBROUTINE F77_NETW_SEND(Length, Buffer, Worker)
      IMPLICIT NONE
      INCLUDE 'CHAN.INC'
      INCLUDE 'SEMA.INC'
      INCLUDE 'THREAD.INC'
      INCLUDE 'NETW.INC'
C
      INTEGER Worker, Length
      CHARACTER*(*) Buffer
C
      COMMON /F77_NETW_DATA/
     1       NumChans, INAddr(4), OUTAddr(4), Path_Length(64),
     2       Path(64), NumWorkers, FreeWorkers, Initialised,
     3       Reserved_Chan, Reserved_Worker,
     4       Workers_Free(F77_SEMA_SIZE), Workers_Sema(F77_SEMA_SIZE),
     5       Worker_Busy(64), Worker_Ptr
C
```

```
206         INTEGER NumChans, INAddr, OUTAddr, Path_Length,
207        1        NumWorkers, FreeWorkers, Reserved_Chan, Reserved_Worker,
208        2        Workers_Free, Workers_Sema, Worker_Ptr
209         CHARACTER*32 Path
210         LOGICAL Worker_Busy, Initialised
211   C
212         INTEGER OutPort, PL
213   C
214   C     Get Output Port
215         OutPort = OUTAddr(ICHAR(Path(Worker)(1:1)))
216         PL = Path_Length(Worker) - 1
217   C
218   C     Send the header
219         CALL F77_CHAN_OUT_WORD(PL,OutPort)
220         IF (PL.GT.0) CALL F77_CHAN_OUT_MESSAGE(PL,Path(Worker)(2:32),
221        1                                       OutPort)
222   C
223   C     Send the buffer
224         CALL F77_CHAN_OUT_WORD(Length,OutPort)
225         CALL F77_CHAN_OUT_MESSAGE(Length,Buffer,OutPort)
226   C
227         RETURN
228         END
229   C
230   C ============================================================================
231   C This subroutine receives a package sent from a worker
232   C Variables returned : 1) Length  = Length (in bytes) of message received.
233   C                      2) Buffer  = Pointer to start of buffer where
234   C                                   message is stored.
235   C                      3) Worker  = Integer number representing worker
236   C                                   number from where message came.
237   C
238         SUBROUTINE F77_NETW_RECEIVE(Length, Buffer, Worker)
239         IMPLICIT NONE
240         INCLUDE 'CHAN.INC'
241         INCLUDE 'SEMA.INC'
242         INCLUDE 'ALT.INC'
243         INCLUDE 'NETW.INC'
244   C
245         INTEGER Worker, Length
246         CHARACTER*(*) Buffer
247   C
248         COMMON /F77_NETW_DATA/
249        1       NumChans, INAddr(4), OUTAddr(4), Path_Length(64),
250        2       Path(64), NumWorkers, FreeWorkers, Initialised,
251        3       Reserved_Chan, Reserved_Worker,
252        4       Workers_Free(F77_SEMA_SIZE), Workers_Sema(F77_SEMA_SIZE),
253        5       Worker_Busy(64), Worker_Ptr
254   C
255         INTEGER NumChans, INAddr, OUTAddr, Path_Length,
256        1        NumWorkers, FreeWorkers, Reserved_Chan, Reserved_Worker,
257        2        Workers_Free, Workers_Sema, Worker_Ptr
258         CHARACTER*32 Path
259         LOGICAL Worker_Busy, Initialised
260   C
261         INTEGER InPort
262   C
263   C     Get Input Port
264   200   IF (Reserved_Worker.EQ.0) THEN
265           InPort = F77_ALT_WAIT_VEC(NumChans,INAddr)
266         ELSE
267           InPort = Reserved_Chan
268         END IF
269   C
270   C     Get the worker number
271         CALL F77_CHAN_IN_WORD(Worker,INAddr(InPort))
272   C
```

```
273  C     Check if worker requested network
274        IF (Worker.LE.0) THEN
275          IF (Reserved_Worker.EQ.0) THEN
276  C         Network is free, RESERVE
277            Reserved_Chan = InPort
278            Reserved_Worker = -Worker
279          ELSE
280  C         Network reserved, RELEASE
281            Reserved_Chan = 0
282            Reserved_Worker = 0
283          END IF
284          GOTO 200
285        END IF
286  C
287  C     Get the message
288        CALL F77_CHAN_IN_WORD(Length,INAddr(InPort))
289        CALL F77_CHAN_IN_MESSAGE(Length,Buffer,INAddr(InPort))
290  C
291        RETURN
292        END
293  C
294  C ============================================================================
295  C This function returns the number of the worker which has reserved the
296  C network.  If the number returned is 0, then the network is not reserved.
297  C This function is useful when debugging.
298  C
299        INTEGER FUNCTION F77_NETW_RESERVED()
300        IMPLICIT NONE
301        INCLUDE 'SEMA.INC'
302  C
303        COMMON /F77_NETW_DATA/
304       1       NumChans, INAddr(4), OUTAddr(4), Path_Length(64),
305       2       Path(64), NumWorkers, FreeWorkers, Initialised,
306       3       Reserved_Chan, Reserved_Worker,
307       4       Workers_Free(F77_SEMA_SIZE), Workers_Sema(F77_SEMA_SIZE),
308       5       Worker_Busy(64), Worker_Ptr
309  C
310        INTEGER NumChans, INAddr, OUTAddr, Path_Length,
311       1        NumWorkers, FreeWorkers, Reserved_Chan, Reserved_Worker,
312       2        Workers_Free, Workers_Sema, Worker_Ptr
313        CHARACTER*32 Path
314        LOGICAL Worker_Busy, Initialised
315  C
316        F77_NETW_RESERVED = Reserved_Worker
317        RETURN
318        END
319  C
320  C============================================================================
321  C This function returns the number of workers currently free to do work.
322  C
323        INTEGER FUNCTION F77_NETW_FREE_WORKERS()
324        IMPLICIT NONE
325        INCLUDE 'SEMA.INC'
326  C
327        COMMON /F77_NETW_DATA/
328       1       NumChans, INAddr(4), OUTAddr(4), Path_Length(64),
329       2       Path(64), NumWorkers, FreeWorkers, Initialised,
330       3       Reserved_Chan, Reserved_Worker,
331       4       Workers_Free(F77_SEMA_SIZE), Workers_Sema(F77_SEMA_SIZE),
332       5       Worker_Busy(64), Worker_Ptr
333  C
334        INTEGER NumChans, INAddr, OUTAddr, Path_Length,
335       1        NumWorkers, FreeWorkers, Reserved_Chan, Reserved_Worker,
336       2        Workers_Free, Workers_Sema, Worker_Ptr
337        CHARACTER*32 Path
338        LOGICAL Worker_Busy, Initialised
339  C
340        INTEGER Free
341  C
```

```
      CALL F77_SEMA_WAIT(Workers_Sema)
      Free = FreeWorkers
      CALL F77_SEMA_SIGNAL(Workers_Sema)
C
      F77_NETW_FREE_WORKERS = Free
      RETURN
      END
C
C=============================================================================
C This function is used to claim a free worker.
C
      INTEGER FUNCTION F77_NETW_USE_WORKER()
      IMPLICIT NONE
      INCLUDE 'SEMA.INC'
C
      COMMON /F77_NETW_DATA/
     1       NumChans, INAddr(4), OUTAddr(4), Path_Length(64),
     2       Path(64), NumWorkers, FreeWorkers, Initialised,
     3       Reserved_Chan, Reserved_Worker,
     4       Workers_Free(F77_SEMA_SIZE), Workers_Sema(F77_SEMA_SIZE),
     5       Worker_Busy(64), Worker_Ptr
C
      INTEGER NumChans, INAddr, OUTAddr, Path_Length,
     1        NumWorkers, FreeWorkers, Reserved_Chan, Reserved_Worker,
     2        Workers_Free, Workers_Sema, Worker_Ptr
      CHARACTER*32 Path
      LOGICAL Worker_Busy, Initialised
C
      INTEGER Free
C
C     Wait for a free worker and decrement the number available
      CALL F77_SEMA_WAIT(Workers_Free)
C
C     Lock shared variables, then claim the free worker
      CALL F77_SEMA_WAIT(Workers_Sema)
C
C     Search for the free worker
      DO 100 WHILE (Worker_Busy(Worker_Ptr))
C       Move on the pointer
        Worker_Ptr = Worker_Ptr + 1
        IF (Worker_Ptr.GT.NumWorkers) Worker_Ptr = 1
100   CONTINUE
C
C     Set the worker as busy
      Worker_Busy(Worker_Ptr) = .TRUE.
      FreeWorkers = FreeWorkers - 1
      Free = Worker_Ptr
C
C     Worker has been claimed, unlock the shared variables
      CALL F77_SEMA_SIGNAL(Workers_Sema)
C
      F77_NETW_USE_WORKER = Free
      RETURN
      END
C
```

```
C==============================================================================
C This subroutine is used to free a worker, but only if it was busy
C
      SUBROUTINE F77_NETW_FREE_WORKER(Worker)
      IMPLICIT NONE
      INTEGER Worker
      INCLUDE 'SEMA.INC'
C
      COMMON /F77_NETW_DATA/
     1       NumChans, INAddr(4), OUTAddr(4), Path_Length(64),
     2       Path(64), NumWorkers, FreeWorkers, Initialised,
     3       Reserved_Chan, Reserved_Worker,
     4       Workers_Free(F77_SEMA_SIZE), Workers_Sema(F77_SEMA_SIZE),
     5       Worker_Busy(64), Worker_Ptr
C
      INTEGER NumChans, INAddr, OUTAddr, Path_Length,
     1        NumWorkers, FreeWorkers, Reserved_Chan, Reserved_Worker,
     2        Workers_Free, Workers_Sema, Worker_Ptr
      CHARACTER*32 Path
      LOGICAL Worker_Busy, Initialised
C
      LOGICAL Busy
C
C     Check if the worker is actually busy, and reset if so
      CALL F77_SEMA_WAIT(Workers_Sema)
      Busy = Worker_Busy(Worker)
      IF (Busy) THEN
        Worker_Busy(Worker) = .FALSE.
        FreeWorkers = FreeWorkers + 1
      END IF
      CALL F77_SEMA_SIGNAL(Workers_Sema)
C
      IF (Busy) CALL F77_SEMA_SIGNAL(Workers_Free)
      RETURN
      END
```

NETWWORK.F77

```
C ================================================================================
C This is the workspace areas used by the network threads
C
      BLOCK DATA
      IMPLICIT NONE
      COMMON /F77_NETW_WORKSPACE/
     1       OutWS(2048), InWS(2048)
      INTEGER InWS, OutWS
      END
C
C ================================================================================
C This is the block data used by the network subroutine and functions
C
      BLOCK DATA
      IMPLICIT NONE
      COMMON /F77_NETW_INTERNAL/
     1       Internal_ChannelS, Internal_ChannelR,
     2       Send_Addr, Receive_Addr,
     3       Worker_Number, NetW_Reserved, Initialised
      INTEGER Internal_ChannelS, Internal_ChannelR,
     1        Send_Addr, Receive_Addr, Worker_Number
      LOGICAL NetW_Reserved, Initialised
      DATA    NetW_Reserved /.FALSE./   Initialised /.FALSE./
      END
C
C ================================================================================
C This subroutine is a thread which handles the messages going from the
C master out to the workers.
C
      SUBROUTINE F77_NETW_OUT_MESSAGES(Dummy)
      IMPLICIT NONE
      INTEGER Dummy
      INCLUDE 'CHAN.INC'
      INCLUDE 'THREAD.INC'
C
      COMMON /F77_NETW_CHANNELS/
     1       NumChans, INAddr(4), OUTAddr(4)
C
      INTEGER NumChans, INAddr, OUTAddr, Port
C
      COMMON /F77_NETW_INTERNAL/
     1       Internal_ChannelS, Internal_ChannelR,
     2       Send_Addr, Receive_Addr,
     3       Worker_Number, NetW_Reserved, Initialised
      INTEGER Internal_ChannelS, Internal_ChannelR,
     1        Send_Addr, Receive_Addr, Worker_Number
      LOGICAL NetW_Reserved, Initialised
C
      INTEGER Path_Length, FromMaster, Message(256), Message_Length,
     1        SendTo
      CHARACTER*32 Path
C
      FromMaster = INAddr(1)
C
      DO 1000 WHILE (.TRUE.)
C       Get Path
        CALL F77_CHAN_IN_WORD(Path_Length,FromMaster)
        IF (Path_Length.GT.0)
     *    CALL F77_CHAN_IN_MESSAGE(Path_Length,Path,FromMaster)
C
C       Get Message
        CALL F77_CHAN_IN_WORD(Message_Length,FromMaster)
        CALL F77_CHAN_IN_MESSAGE(Message_Length,Message,FromMaster)
C
```

```
C       Forward Message
        IF (Path_Length.EQ.0) THEN
C         Send to this worker
          SendTo = Receive_Addr
        ELSE IF (Path_Length.LT.0) THEN
C         Broadcast to other processors
          DO 500 Port=2,NumChans
            SendTo = OUTAddr(Port)
            CALL F77_CHAN_OUT_WORD(Path_Length,SendTo)
            CALL F77_CHAN_OUT_WORD(Message_Length,SendTo)
            CALL F77_CHAN_OUT_MESSAGE(Message_Length,Message,SendTo)
500       CONTINUE
C
C         Send to this worker as well
          SendTo = Receive_Addr
        ELSE
C         Forward to next worker, start by sending path
          SendTo = OUTAddr(ICHAR(Path(1:1)))
          Path_Length = Path_Length - 1
          CALL F77_CHAN_OUT_WORD(Path_Length,SendTo)
          IF (Path_Length.GT.0)
     *      CALL F77_CHAN_OUT_MESSAGE(Path_Length,Path(2:32),SendTo)
        END IF
C
C       Send Actual Message
        CALL F77_CHAN_OUT_WORD(Message_Length,SendTo)
        CALL F77_CHAN_OUT_MESSAGE(Message_Length,Message,SendTo)
1000  CONTINUE
      RETURN
      END
C
C ============================================================================
C This subroutine is a thread which handles the messages going from the
C workers to the master.
C
      SUBROUTINE F77_NETW_IN_MESSAGES(Dummy)
      IMPLICIT NONE
      INTEGER Dummy
      INCLUDE 'ALT.INC'
C
      COMMON /F77_NETW_INTERNAL/
     1       Internal_ChannelS, Internal_ChannelR,
     2       Send_Addr, Receive_Addr,
     3       Worker_Number, NetW_Reserved, Initialised
      INTEGER Internal_ChannelS, Internal_ChannelR,
     1        Send_Addr, Receive_Addr, Worker_Number
      LOGICAL NetW_Reserved, Initialised
C
      COMMON /F77_NETW_CHANNELS/
     1       NumChans, INAddr(4), OUTAddr(4)
      INTEGER NumChans, INAddr, OUTAddr
C
      INTEGER Message(256), Message_Length,
     1        INS(4), ToMaster, MessFrom, WorkerNo, I
C
C     Intialise channel variables
      INS(1) = Send_Addr
      DO I = 2,NumChans
        INS(I) = INAddr(I)
      END DO
      ToMaster = OUTAddr(1)
C
```

```
      DO 1000 WHILE (.TRUE.)
C       Wait for a message to send
        MessFrom = F77_ALT_WAIT_VEC(NumChans,INS)
C
C      Get The Message
C       Get the worker number
        CALL F77_CHAN_IN_WORD(WorkerNo,INS(MessFrom))
C
C       If Worker number positive, get and send message
        IF (WorkerNo.GT.0) THEN
C         Get the actual message
          CALL F77_CHAN_IN_WORD(Message_Length,INS(MessFrom))
          CALL F77_CHAN_IN_MESSAGE(Message_Length,Message,
     1                             INS(MessFrom)          )
C                                   .
C         Pass on the message to the master
          CALL F77_CHAN_OUT_WORD(WorkerNo,ToMaster)
          CALL F77_CHAN_OUT_WORD(Message_Length,ToMaster)
          CALL F77_CHAN_OUT_MESSAGE(Message_Length,Message,ToMaster)
        ELSE
C         Claim the network for this worker
          CALL F77_CHAN_OUT_WORD(WorkerNo,ToMaster)
C
C         Loop, receiving all messages from this worker until
C         another negative worker number is received.
500       CALL F77_CHAN_IN_WORD(WorkerNo,INS(MessFrom))
          CALL F77_CHAN_OUT_WORD(WorkerNo,ToMaster)
C
C         If Worker number positive, get and send message
          IF (WorkerNo.GT.0) THEN
C           Get the actual message
            CALL F77_CHAN_IN_WORD(Message_Length,INS(MessFrom))
            CALL F77_CHAN_IN_MESSAGE(Message_Length,Message,
     1                               INS(MessFrom)          )
C
C           Pass on the message to the master
            CALL F77_CHAN_OUT_WORD(Message_Length,ToMaster)
            CALL F77_CHAN_OUT_MESSAGE(Message_Length,Message,ToMaster)
            GOTO 500
          END IF
        END IF
C
1000  CONTINUE
C
      RETURN
      END
C
C ============================================================================
C This subroutine will place the message passed to it on the network, with
C its destination being the master.
C Variables passed : 1) Length  = Length (in bytes) of message to pass
C                    2) Buffer  = Pointer to start of message buffer to
C                                 be sent.
C
      SUBROUTINE F77_NETW_SEND(Length, Buffer)
      IMPLICIT NONE
      INCLUDE 'CHAN.INC'
      INCLUDE 'NETW.INC'
C
      INTEGER Length
      CHARACTER*(*) Buffer
C
      COMMON /F77_NETW_INTERNAL/
     1       Internal_ChannelS, Internal_ChannelR,
     2       Send_Addr, Receive_Addr,
     3       Worker_Number, NetW_Reserved, Initialised
      INTEGER Internal_ChannelS, Internal_ChannelR,
     1        Send_Addr, Receive_Addr, Worker_Number
      LOGICAL NetW_Reserved, Initialised
C
      CALL F77_CHAN_OUT_WORD(Worker_Number,Send_Addr)
```

```
198         CALL F77_CHAN_OUT_WORD(Length,Send_Addr)
199         CALL F77_CHAN_OUT_MESSAGE(Length,Buffer,Send_Addr)
200   C
201         RETURN
202         END
203   C
204   C ===========================================================================
205   C This subroutine will receive a message from the network destined for this
206   C worker.
207   C Variables passed : 1) Length  = Length (in bytes) of message received
208   C                    2) Buffer  = Pointer to start of message buffer in
209   C                                 which the received message is stored.
210   C
211         SUBROUTINE F77_NETW_RECEIVE(Length, Buffer)
212         IMPLICIT NONE
213         INCLUDE 'CHAN.INC'
214         INCLUDE 'NETW.INC'
215   C
216         INTEGER Length
217         CHARACTER*(*) Buffer
218   C
219         COMMON /F77_NETW_INTERNAL/
220        1        Internal_ChannelS, Internal_ChannelR,
221        2        Send_Addr, Receive_Addr,
222        3        Worker_Number, NetW_Reserved, Initialised
223         INTEGER Internal_ChannelS, Internal_ChannelR,
224        1         Send_Addr, Receive_Addr, Worker_Number
225         LOGICAL NetW_Reserved, Initialised
226   C
227         CALL F77_CHAN_IN_WORD(Length,Receive_Addr)
228         CALL F77_CHAN_IN_MESSAGE(Length,Buffer,Receive_Addr)
229   C
230         RETURN
231         END
232   C
233   C ===========================================================================
234   C This subroutine is called to reserve the network for messages from this
235   C worker.  It has no effect if the network is already reserved.
236   C
237         SUBROUTINE F77_NETW_USE()
238   C
239         COMMON /F77_NETW_INTERNAL/
240        1        Internal_ChannelS, Internal_ChannelR,
241        2        Send_Addr, Receive_Addr,
242        3        Worker_Number, NetW_Reserved, Initialised
243         INTEGER Internal_ChannelS, Internal_ChannelR,
244        1         Send_Addr, Receive_Addr, Worker_Number
245         LOGICAL NetW_Reserved, Initialised
246   C
247         IF (.NOT.NetW_Reserved) THEN
248           NetW_Reserved = .TRUE.
249           CALL F77_CHAN_OUT_WORD(-Worker_Number,Send_Addr)
250         END IF
251   C
252         RETURN
253         END
254   C
255   C ===========================================================================
256   C This subroutine is called to free the network after it has been reserved.
257   C It has no effect if the network is already free.
258   C
259         SUBROUTINE F77_NETW_FREE()
260   C
261         COMMON /F77_NETW_INTERNAL/
262        1        Internal_ChannelS, Internal_ChannelR,
263        2        Send_Addr, Receive_Addr,
264        3        Worker_Number, NetW_Reserved, Initialised
265         INTEGER Internal_ChannelS, Internal_ChannelR,
266        1         Send_Addr, Receive_Addr, Worker_Number
267         LOGICAL NetW_Reserved, Initialised
268   C
```

```
      IF (NetW_Reserved) THEN
        NetW_Reserved = .FALSE.
        CALL F77_CHAN_OUT_WORD(-Worker_Number,Send_Addr)
      END IF
C
      RETURN
      END
C
C ================================================================================
C This subroutine is called to initialise the channel variables used as
C well as to start the threads which control the network message passing.
C
      SUBROUTINE F77_NETW_WORKER()
      IMPLICIT NONE
      INCLUDE 'CHAN.INC'
      INCLUDE 'THREAD.INC'
      EXTERNAL F77_NETW_IN_MESSAGES, F77_NETW_OUT_MESSAGES
C
      COMMON /F77_NETW_CHANNELS/
     1       NumChans, INAddr(4), OUTAddr(4)
      INTEGER NumChans, INAddr, OUTAddr
C
      COMMON /F77_NETW_INTERNAL/
     1       Internal_ChannelS, Internal_ChannelR,
     2       Send_Addr, Receive_Addr,
     3       Worker_Number, NetW_Reserved, Initialised
      INTEGER Internal_ChannelS, Internal_ChannelR,
     1        Send_Addr, Receive_Addr, Worker_Number
      LOGICAL NetW_Reserved, Initialised
C
      INTEGER InWSSize, OutWSSize
      PARAMETER
     1  (InWSSize  = 2048,
     2   OutWSSize = 2048 )
C
      COMMON /F77_NETW_WORKSPACE/
     1       OutWS(2048), InWS(2048)
      INTEGER InWS, OutWS
C
      INTEGER HostChanIN, HostChanOUT, PathLength,
     1        I, J, MessFrom, NextID, RID
      CHARACTER*32 Path
      LOGICAL Initialise
C
C     Ensure that this routine is called only once
      IF (Initialised) GOTO 160
C
C     Initialise addresses to all channels
      NumChans = F77_CHAN_IN_PORTS()
      DO 100 I = 1,NumChans
        J = I - 1
        INAddr(I)  = F77_CHAN_IN_PORT(J)
        OUTAddr(I) = F77_CHAN_OUT_PORT(J)
100   CONTINUE
C
C     Wait for a message from one of the closest transputers to the host.
C     Message is to be this transputers ID, and the ID increment is to
C     be returned back to the host, as well as a message specifying routing
C     path of 0, indicating this is to be the destination transputer.
      HostChanIN = INAddr(1)
      HostChanOUT = OUTAddr(1)
      CALL F77_CHAN_IN_WORD(Worker_Number,HostChanIN)
      CALL F77_CHAN_OUT_WORD(Worker_Number+1,HostChanOUT)
      CALL F77_CHAN_OUT_WORD(0,HostChanOUT)
C
C     Get Next ID
      CALL F77_CHAN_IN_WORD(NextID,HostChanIN)
C
```

```
337  C     Initialise channel variables
338        DO 150 I = 2,NumChans
339          Initialise = .TRUE.
340          DO 140 WHILE (Initialise)
341  C         Pass on to next T/P
342            CALL F77_CHAN_OUT_WORD(NextID,OUTAddr(I))
343  C         Get result
344            CALL F77_CHAN_IN_WORD(RID,INAddr(I))
345            Initialise = (RID.NE.-1)
346            IF (Initialise) THEN
347  C           Send back ID
348              CALL F77_CHAN_OUT_WORD(RID,HostChanOUT)
349  C           Get Path
350              CALL F77_CHAN_IN_WORD(PathLength,INAddr(I))
351              IF (PathLength.NE.0)
352       *        CALL F77_CHAN_IN_MESSAGE(PathLength,Path(2:32),INAddr(I))
353              Path(1:1) = CHAR(I)
354              PathLength = PathLength + 1
355  C           Send back path length
356              CALL F77_CHAN_OUT_WORD(PathLength,HostChanOUT)
357              CALL F77_CHAN_OUT_MESSAGE(PathLength,Path,HostChanOUT)
358  C           Get Next ID
359              CALL F77_CHAN_IN_WORD(NextID,HostChanIN)
360            END IF
361  140     CONTINUE
362  150   CONTINUE
363  C .
364  C     Done Network Initialisation
365        CALL F77_CHAN_OUT_WORD(-1,HostChanOUT)
366  C
367  C     Set up the internal channels
368        Send_Addr = F77_CHAN_ADDRESS(Internal_ChannelS)
369        Receive_Addr = F77_CHAN_ADDRESS(Internal_ChannelR)
370        CALL F77_CHAN_INIT(Send_Addr)
371        CALL F77_CHAN_INIT(Receive_Addr)
372  C
373  C     Start the send and receive threads
374        CALL F77_THREAD_START(F77_NETW_OUT_MESSAGES,OutWS,OutWSSize*4,
375       *                      F77_THREAD_URGENT,1,I)
376        CALL F77_THREAD_START(F77_NETW_IN_MESSAGES,InWS,InWSSize*4,
377       *                      F77_THREAD_URGENT,1,I)
378  C
379        Initialised = .TRUE.
380  160   RETURN
381        END
```

Chapter 4 - OCCAM/Turbo Pascal Server

This chapter is divided into two sections, the Turbo Pascal Source of the *Host Server* and the OCCAM 2 source of the *Transputer Server* or transputer libraries. In the latter section, the library source files all have the extensions **OCC** where the include files **TPSERVER.INC, TPSCREEN.INC, TPFILE.INC, TPDOS.INC** and **TPGRAPH.INC** must be included in when using the server, the screen libarary, file library, dos library and graph library respectively. The **TPSUBSYS.OCC** and **QUADPUTE.OCC** libraries may be included when a subsystem, such as the quadputer, is involved. A user manual for this system is provided as an appendix in the thesis.

(i) Turbo Pascal Source

"ALSERVER.PAS"

```
{$M 65520,0,500360}
Program TestServer;
Uses Crt, Dos, Graph, Printer, TranUtil, FileHand, Graphics;

Var
  OldBreakAddress                    : Pointer;
  OldBreakState                      : Byte;

{$I INLINE.PAS}
{$I KEYBOARD.PAS}
{$I SCREEN.PAS}
{$I DOS.PAS}

{-------------------------------------------------------------------------}
{ This procedure is the handler of the server.                            }
{                                                                         }
Procedure Handler;
Var Done, Error : Boolean;
Begin
  {START DIRECTLY BY CONTINUING}
  If KeyboardInOperation Then
   begin
     Case Instruction Of
       1 : TPKeypressed;
       2 : TPReadKey(FALSE);
       3 : TPReadKey(TRUE);
       4 : TPReadString(FALSE);
       5 : TPReadString(TRUE);
       6 : TPReadNumber(FALSE);
       7 : TPReadNumber(TRUE);
     End;
   end;

  {START THE MAIN LOOP}
  Done := FALSE;
  Repeat
    {GET INSTRUCTION}
    Get_Instruction(Instruction);
```

```
If (Instruction < 50)  Then
 begin
   KeyboardInOperation := TRUE;

   {KEYBOARD}
   Case Instruction Of
       1 : TPKeypressed;
       2 : TPReadKey(FALSE);
       3 : TPReadKey(TRUE);
       4 : TPReadString(FALSE);
       5 : TPReadString(TRUE);
       6 : TPReadNumber(FALSE);
       7 : TPReadNumber(TRUE);
       8 : BreakOn := (Get_Byte <> 0);
   End;

   KeyboardInOperation := FALSE;
 end            Else
If (Instruction < 100) Then
 begin
   {SCREEN}
   Case Instruction Of
      51 : ClrScr;
      52 : ClrEos;
      53 : ClrEol;
      54 : GotoXY(Get_Byte,Get_Byte);
      55 : NormVideo;
      56 : HighVideo;
      57 : LowVideo;
      58 : Sound(Get_Word);
      59 : NoSound;
      60 : TPWhereX;
      61 : TPWhereY;
      62 : TextBackground(Get_Byte);
      63 : TextColor(Get_Byte);
      64 : TextMode(Get_Byte);
      65 : Write(Get_String);
      66 : WriteLn(Get_String);
      80 : PrintScreen;
   End;
 end            Else
If (Instruction < 150) Then
 begin
   {FILES}
   Case Instruction Of
     101,
     102 : TPAssignFile;
     103 : TPResetFile;
     104 : TPRewriteFile;
     105 : TPAppendFile;
     106 : TPCloseFile;
     107 : TPRenameFile;
     108 : TPDeleteFile;
     109 : TPEndOfFile;
     110 : TPFileError;
     111 : TPReadTextLine;
     112 : TPReadTextNum;
     113 : TPReadBlock;
     114 : TPReadFile(1);       {BOOLEAN}
     115 : TPReadFile(1);       {BYTE}
     116 : TPReadFile(4);       {INT}
     117 : TPReadFile(2);       {INT16}
     118 : TPReadFile(4);       {INT32}
     119 : TPReadFile(8);       {INT64}
     120 : TPReadFile(4);       {REAL32}
     121 : TPReadFile(8);       {REAL64}
```

```
        131 : TPWriteTextLine;
        132 : TPWriteText;
        133 : TPWriteBlock;
        134 : TPWriteFile(1);      {BOOLEAN}
        135 : TPWriteFile(1);      {BYTE}
        136 : TPWriteFile(4);      {INT}
        137 : TPWriteFile(2);      {INT16}
        138 : TPWriteFile(4);      {INT32}
        139 : TPWriteFile(8);      {INT64}
        140 : TPWriteFile(4);      {REAL32}
        141 : TPWriteFile(8);      {REAL64}
      End;
    end           Else
  If (Instruction < 200) Then
    begin
      {DOS}
      Case Instruction Of
        151 : TPExec;
        152 : TPGetDate;
        153 : SetDate(Get_Word,Get_Word,Get_Word);
        154 : TPGetTime;
        155 : SetTime(Get_Word,Get_Word,Get_Word,Get_Word);
        156 : Send_Long(DiskFree(Get_Word));
        157 : Send_Long(DiskSize(Get_Word));
        158 : TPGetFAttr;
        159 : TPSetFAttr;
        160 : TPFileExists;
        161 : TPInterrupt;
        162 : Send_Byte(Port[Get_Word]);
        163 : TPWritePort;
        164 : TPFileSize;
      End;
    end           Else
  If (Instruction < 250) Then
    begin
      {GRAPH}
      Case Instruction Of
        201 : TPInitGraph;
        202 : TPSetGraphMode;
        203 : TPRestoreCrtMode;
        204 : TPCloseGraphics;
        205 : TPSetViewPort;
        206 : ClearViewPort;
        207 : ClearDevice;
        208 : TPDetectGraph;
        209 : TPSetLineStyle;
        210 : TPSetFillPattern;
        211 : TPSetFillStyle;
        212 : TPArc;
        213 : TPBar;
        214 : TPBar3D;
        215 : TPCircle;
        216 : TPDrawPoly;
        217 : TPLine;
        218 : TPLineTo;
        219 : TPMoveRel;
        220 : TPMoveTo;
        221 : TPPutPixel;
        222 : OutText(Get_String);
        223 : TPOutTextXY;
        224 : TPRectangle;
        225 : TPSetTextJustify;
        226 : TPSetTextStyle;
        227 : TPGetAspectRatio;
        228 : TPGetMaxX;
        229 : TPGetMaxY;
        230 : TPGraphResult;
        231 : TPSetColor;
        232 : TPSetBkColor;
        233 : TPGetColor;
        234 : TPGetBkColor;
```

```
        240 : TPEllipse;
        241 : TPFillEllipse;
        242 : TPFillPoly;
        243 : TPFloodFill;
        244 : TPGetArcCoords;
        245 : TPLineRel;
        246 : TPPieSlice;
        247 : TPSector;
      End;
    end           Else
    begin
      Case Instruction Of
        250 : TPSystemInfo;        {SEND SYSTEM INFO DETERMINED BEFORE BOOT}
        251 : TPBootSubsystem;     {SEND 2nd BOOT FILE (SIZE 1st) TO Host}
        252 : TPBootFileSubSys;    {SEND BOOT FILE PASSED BY NAME TO Host}
        255 : Done := TRUE;        {TERMINATE SERVER}
      End;
    end;
  Until Done;
End;

{-------------------------------------------------------------------------}
{ This procedure is the interrupt procedure for break.                    }

Procedure DoBreak; Interrupt;
Begin
  Blip;
  Grab_Attention := TRUE;
End;

{-------------------------------------------------------------------------}
{ This initialises the system.                                            }

Procedure Initialise;
Begin
  {INITIALISE BREAK MODE}
  BreakOn := TRUE;

  {TURN OFF BREAK CHECKING}
  CheckBreak := True;
  Inline($B8/$00/33/                {MOV AX,3300h        }
         $CD/$21/                   {INT 21h             }
         $88/$16/OldBreakState/     {MOV OldBreakState,DL}
         $B8/$01/$33/               {MOV AX,3301h        }
         $B2/$00/                   {MOV DL,00h          }
         $CD/$21                    {INT 21h             } );

  {Keep and reset old break address}
  GetIntVec($1B,OldBreakAddress);
  SetIntVec($1B,@DoBreak);.             {Set interrupt to break handler};
End;

{-------------------------------------------------------------------------}
{ This procedure closes off the system.                                   }

Procedure Finalise;
Begin
  {Restore break checking}
  Inline($B8/$01/$33/               {MOV AX,3301h        }
         $8A/$16/OldBreakState/     {MOV DL,OldBreakState}
         $CD/$21                    {INT 21h             } );

  {Restore Old Break Address}
  SetIntVec($1B,OldBreakAddress);
End;
```

```
{-------------------------------------------------------------------------}
Begin
  Initialise;
  Handler;
  Finalise;
End.
```

"PROTOCOL.PAS"

```
Const
  {KEYBOARD PROTOCOL   1 -  50 }
  TKeypressed          : Byte =   1;
  TReadKey             : Byte =   2;
  TReadEchoKey         : Byte =   3;
  TReadString          : Byte =   4;
  TReadEchoString      : Byte =   5;
  TReadNumber          : Byte =   6;
  TReadEchoNumber      : Byte =   7;
  TBreakOn             : Byte =   8;

  {SCREEN PROTOCOL    51 - 100 }
  TClrScr              : Byte =  51;
  TClrEOS              : Byte =  52;
  TClrEOl              : Byte =  53;
  TGotoXY              : Byte =  54;
  TNormVideo           : Byte =  55;
  THighVideo           : Byte =  56;
  TLowVideo            : Byte =  57;
  TSound               : Byte =  58;
  TNoSound             : Byte =  59;
  TWhereX              : Byte =  60;
  TWhereY              : Byte =  61;
  TTextBackground      : Byte =  62;
  TTextColor           : Byte =  63;
  TTextMode            : Byte =  64;
  TWriteString         : Byte =  65;
  TWriteLnString       : Byte =  66;

  {FILE PROTOCOL     101 - 150 }
  TAssignText          : Byte = 101;
  TAssignFile          : Byte = 102;
  TResetFile           : Byte = 103;
  TRewriteFile         : Byte = 104;
  TAppendText          : Byte = 105;
  TCloseFile           : Byte = 106;
  TRenameFile          : Byte = 107;
  TDeleteFile          : Byte = 108;
  TEndOfFile           : Byte = 109;
  TFileError           : Byte = 110;

  TReadTextLine        : Byte = 111;
  TReadTextNum         : Byte = 112;
  TReadBlock           : Byte = 113;
  TReadBool            : Byte = 114;
  TReadByte            : Byte = 115;
  TReadInt             : Byte = 116;
  TReadInt16           : Byte = 117;
  TReadInt32           : Byte = 118;
  TReadInt64           : Byte = 119;
  TReadReal32          : Byte = 120;
  TReadReal64          : Byte = 121;

  TWriteTextLine       : Byte = 131;
  TWriteText           : Byte = 132;
  TWriteBlock          : Byte = 133;
  TWriteBool           : Byte = 134;
  TWriteByte           : Byte = 135;
  TWriteInt            : Byte = 136;
  TWriteInt16          : Byte = 137;
  TWriteInt32          : Byte = 138;
```

```
  TWriteInt64          : Byte = 139;
  TWriteReal32         : Byte = 140;
  TWriteReal64         : Byte = 141;

  {DOS PROTOCOL      151 - 200 }
  TExec                : Byte = 151;
  TGetDate             : Byte = 152;
  TSetDate             : Byte = 153;
  TGetTime             : Byte = 154;
  TSetTime             : Byte = 155;
  TDiskFree            : Byte = 156;
  TDiskSize            : Byte = 157;
  TGetFAttr            : Byte = 158;
  TSetFAttr            : Byte = 159;
  TFileExists          : Byte = 160;
  TInterrupt           : Byte = 161;
  TReadPort            : Byte = 162;
  TWritePort           : Byte = 163;

  {GRAPH PROTOCOL    201 - 240 }
  TInitGraph           : Byte = 201;
  TSetGraphMode        : Byte = 202;
  TRestoreCrtMode      : Byte = 203;
  TCloseGraph          : Byte = 204;
  TSetViewPort         : Byte = 205;
  TClearViewPort       : Byte = 206;
  TClearDevice         : Byte = 207;
  TDetectGraph         : Byte = 208;
  TSetLineStyle        : Byte = 209;
  TSetFillPattern      : Byte = 210;
  TSetFillStyle        : Byte = 211;
  TArc                 : Byte = 212;
  TBar                 : Byte = 213;
  TBar3D               : Byte = 214;
  TCircle              : Byte = 215;
  TDrawPoly            : Byte = 216;
  TLine                : Byte = 217;
  TLineRes             : Byte = 218;
  TMoveRel             : Byte = 219;
  TMoveTo              : Byte = 220;
  TPutPixel            : Byte = 221;
  TOutText             : Byte = 222;
  TOutTextXY           : Byte = 223;
  TRectangle           : Byte = 224;
  TSetTextJustify      : Byte = 225;
  TSetTextStyle        : Byte = 226;
  TGetAspectRatio      : Byte = 227;
  TGetMaxX             : Byte = 228;
  TGetMaxY             : Byte = 229;
  TGraphResult         : Byte = 230;
  TSetColor            : Byte = 231;
  TSetBkColor          : Byte = 232;
  TGetColor            : Byte = 233;
  TGetBkColor          : Byte = 234;
  TEllipse             : Byte = 240;
  TFillEllipse         : Byte = 241;
  TFillPoly            : Byte = 242;
  TFloodFill           : Byte = 243;
  TGetArcCoords        : Byte = 244;
  TLineRel             : Byte = 245;
  TPieSlice            : Byte = 246;
  TSector              : Byte = 247;

  {SERVER PROTOCOL  241 - 255}
  TSystemInfo          : Byte = 250;
  TBootSubsystem       : Byte = 251;
  TBootFileSubSys      : Byte = 252;
  TEndServer           : Byte = 255;
```

"IOERRNOS.PAS"

```
{-------------------------------------------------------------------------}
{ This is the list of IOErrors which may occur.                           }
{                                                                         }

CONST
   NoError        : Byte =   0;
   FileNotFound   : Byte =   2;
   PathNotFound   : Byte =   3;
   TooManyFiles   : Byte =   4;
   AccessDenied   : Byte =   5;
   InvalidHandle  : Byte =   6;
   InvalidAccess  : Byte =  12;
   InvalidDrive   : Byte =  15;
   CantRemoveDir  : Byte =  16;
   CantRename     : Byte =  17;
   DiskReadError  : Byte = 100;
   DiskWriteError : Byte = 101;
   FileNotAssign  : Byte = 102;
   FileNotOpen    : Byte = 103;
   FileNotInput   : Byte = 104;
   FileNotOutput  : Byte = 105;
   InvalidNumeric : Byte = 106;
```

"INLINE.PAS"

```
{-------------------------------------------------------------------------}
{ This is the function which returns the value of the byte sent from the  }
{ host transputer.  For purposes of speed, it has been made INLINE.       }
{                                                                         }
Function Get_Byte : Byte;
Inline($8B/$16/Link_In_Status/          {      MOV DX,[Link_In_Status]     }
       $EC/                             {Loop: IN AL,DX                    }
       $24/$01/                         {      AND AL,01                   }
       $74/$FB/                         {      JZ Loop                     }
       $83/$EA/$02/                     {      SUB DX,02  ;Adjust for read.}
       $EC);                            {      IN AL,DX                    }

{-------------------------------------------------------------------------}
{ This is the function which returns the value of the word sent from the  }
{ host transputer.  For purposes of speed, it has been made INLINE.       }
{                                                                         }
Function Get_Word : Word;
Inline($8B/$16/Link_In_Status/          {      MOV DX,[Link_In_Status]     }
       $EC/                             {Loop: IN AL,DX                    }
       $24/$01/                         {      AND AL,01                   }
       $74/$FB/                         {      JZ Loop                     }
       $83/$EA/$02/                     {      SUB DX,02  ;Adjust for read.}
       $EC/                             {      IN AL,DX                    }
       $88/$C1/                         {      MOV CL,AL                   }
       $83/$C2/$02/                     {      ADD DX,02  ;Adjust to status}
       $EC/                             {Loop2:IN AL,DX                    }
       $24/$01/                         {      AND AL,01                   }
       $74/$FB/                         {      JZ Loop2                    }
       $83/$EA/$02/                     {      SUB DX,02  ;Adjust for read.}
       $EC/                             {      IN AL,DX                    }
       $88/$C4/                         {      MOV AH,AL                   }
       $88/$C8);                        {      MOV AL,CL                   }

{-------------------------------------------------------------------------}
{ This procedure tries to send a byte to the transputer.  For speed purposes }
{ it is INLINE.                                                           }
{                                                                         }
Procedure Send_Byte(Bite : Byte);
Inline($8B/$16/Link_Out_Status/         {      MOV DX,[Link_Out_Status]    }
       $EC/                             {Loop: IN AL,DX                    }
       $24/$01/                         {      AND AL,01                   }
       $74/$FB/                         {      JZ Loop                     }
       $83/$EA/$02/                     {      SUB DX,02  ;Adjust for write.}
       $58/                             {      POP AX                      }
       $EE);                            {      OUT DX,AL                   }
```

```
{-------------------------------------------------------------------------}
{ This procedure tries to send a word to the transputer and is also INLINE. }
{                                                                         }
Procedure Send_Word(Wrd : Word);
Inline($88/$16/Link_Out_Status/         {       MOV DX,[Link_Out_Status]    }
       $EC/                             {Loop: IN AL,DX                     }
       $24/$01/                         {       AND AL,01                   }
       $74/$FB/                         {       JZ Loop                     }
       $83/$EA/$02/                     {       SUB DX,02  ;Adjust for write. }
       $58/                             {       POP AX                      }
       $EE/                             {       OUT DX,AL                   }
       $83/$C2/$02/                     {       ADD DX,02  ;Adjust to status }
       $EC/                             {Loop2:IN AL,DX                     }
       $24/$01/                         {       AND AL,01                   }
       $74/$FB/                         {       JZ Loop2                    }
       $83/$EA/$02/                     {       SUB DX,02  ;Adjust for write. }
       $88/$E0/                         {       MOV AL,AH                   }
       $EE);                            {       OUT DX,AL                   }

{-------------------------------------------------------------------------}
{ This procedure tries to send a word to the transputer and is also INLINE. }
{                                                                         }
Procedure Send_Int(Int : Integer);
Inline($88/$16/Link_Out_Status/         {       MOV DX,[Link_Out_Status]    }
       $EC/                             {Loop: IN AL,DX                     }
       $24/$01/                         {       AND AL,01                   }
       $74/$FB/                         {       JZ Loop                     }
       $83/$EA/$02/                     {       SUB DX,02  ;Adjust for write. }
       $58/                             {       POP AX                      }
       $EE/                             {       OUT DX,AL                   }
       $83/$C2/$02/                     {       ADD DX,02  ;Adjust to status }
       $EC/                             {Loop2:IN AL,DX                     }
       $24/$01/                         {       AND AL,01                   }
       $74/$FB/                         {       JZ Loop2                    }
       $83/$EA/$02/                     {       SUB DX,02  ;Adjust for write. }
       $88/$E0/                         {       MOV AL,AH                   }
       $EE);                            {       OUT DX,AL                   }
```

"KEYBOARD.PAS"

```
{                                                                         }
{  THIS INCLUDE FILE CONTAINS THE KEYBOARD HANDLING ROUTINES FOR THE       }
{  TRANSPUTER SERVER.                                                     }
{                                                                         }

{-------------------------------------------------------------------------}
{ This procedure handles the keypressed function.                         }
{    Passed  : Nothing                                                    }
{    Returns : Byte = 0 for NOT KEYPRESSED                                }
{                     1 for KEYPRESSED                                    }
{                                                                         }
Procedure TPKeypressed;
Begin
  If KeyIsPressed Then Send_Byte(1)
                  Else Send_Byte(0);
End;

{-------------------------------------------------------------------------}
{ This procedure will read a key from the keyboard (without echoing).     }
{    Passed  : Nothing                                                    }
{    Returns : Char = Key that was pressed.                               }
{                                                                         }
Procedure TPReadKey(Echo : Boolean);
Var Ch : Char;
Begin
  Ch := ReadAKey;
  If Echo Then Write(Ch);
  Send_Byte(Ord(Ch));
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure will read a string from the keyboard.                    }
{    Passed  : Maximum String Size                                        }
{    Returns : Length of String Read.                                     }
{              Characters that make up the string.                        }
{                                                                         }
Procedure TPReadString(Echo : Boolean);
Var St            : String255;
    Ch            : Char;
    Done          : Boolean;
    I,J, MaxSize  : Byte;
Begin
  PreventExit := TRUE;

  {GET THE MAXIMUM SIZE OF THE STRING}
  MaxSize := Get_Byte;

  {READ THE STRING}
  I := 0;  Done := False;
  Repeat
    Ch := ReadAKey;
    If (Ch >= ' ') AND (Ch <= '~') AND (I < MaxSize)
     Then begin
            Inc(I);
            St[I] := Ch;
            If Echo Then Write(Ch);
          end
     Else If (Ch = #8) then
          begin
            if (I > 0)
             then begin
                    dec(i);
                    if echo then write(^H,' ',^H);
                  end
             else bell;
          end
     Else Done := (Ch = #13);
  Until Done;

  {SEND THE STRING}
  St[0] := CHR(I);
  Send_String(St);

  PreventExit := FALSE;
End;

{-------------------------------------------------------------------------}
{ This procedure will read a number from the keyboard, ensuring that the  }
{ number is a valid one.  It will then send the number in string format to }
{ the transputer.                                                         }
{                                                                         }
Procedure TPReadNumber(Echo : Boolean);
Var St           : String255;
    Ch           : Char;
    Done         : Boolean;
    I,J, MaxSize : Byte;
    K            : Word;
    Num          : Real;
    ECode        : Integer;
Begin
  PreventExit := TRUE;
  MaxSize := Get_Byte;
  REPEAT
    {READ THE STRING}
    I := 0;  Done := False;
```

```
    Repeat
      Ch := ReadAKey;
      If (Ch >= ' ') AND (Ch <= '~') AND (I < MaxSize)
       Then begin
              Inc(I);
              St[I] := Ch;
              If Echo Then Write(Ch);
            end
       Else If (Ch = #8) then
            begin
              if (I > 0)
               then begin
                      dec(i);
                      if echo then write(^H,' ',^H);
                    end
               else bell;
            end
       Else Done := (Ch = #13);
    Until (I = 255) OR Done;

    {SEE IF VALID}
    St[0] := Chr(I);
    Val(St,Num,ECode);
    Done := (ECode = 0) OR (St = '');
    If Not Done Then
     begin
       if echo then
         begin
            for j := 1 to i do  write (^H,' ',^H);
            write('Error in numeric format - Redo');
         end;
       bell;
       if echo then
         begin
            k := 0;
            while not keyispressed and (k < 4000) do
              inc(k);
            for j := 1 to 30  do write(^H,' ',^H);
         end;
     end;
  UNTIL DONE;

  {SEND THE STRING}
  Send_String(St);
  PreventExit := FALSE;
End;
```

"SCREEN.PAS"

```
{                                                                          }
{  THIS INCLUDE FILE CONTAINS THE SCREEN HANDLING ROUTINES FOR THE         }
{  TRANSPUTER SERVER.                                                      }
{                                                                          }

{--------------------------------------------------------------------------}
{ This procedure will clear the screen from the current cursor position.   }
{                                                                          }
Procedure ClrEos;
Var X, Y, I : Byte;
Begin
  x := wherex;    y := wherey;
  clreol;
  for i := (Y+1) to 25 do
   Begin
     GotoXY(1,i);    ClrEol;
   End;
  GotoXY(x,y);
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure will return to the transputer the X location of the cursor. }
{    Input  : None.                                                       }
{    Output : Byte giving X location.                                     }
{                                                                         }
Procedure TPWhereX;
Begin
  Send_Byte(WhereX);
End;

{-------------------------------------------------------------------------}
{ This procedure will return to the transputer the YX location of the cursor.}
{    Input  : None.                                                       }
{    Output : Byte giving Y location.                                     }
{                                                                         }
Procedure TPWhereY;
Begin
  Send_Byte(WhereY);
End;

{-------------------------------------------------------------------------}
{ This procedure will print the current screen (text or graphics) to the  }
{ printer.  It call the normal print screen routine if in TEXT mode,      }
{ otherwise it calls its own routine to dump the graphics.                }
{                                                                         }
Procedure PrintScreen;
Var LCount, LC8, I   : Integer;
    Bite             : Byte;
Begin
  If Not(InGraphicsMode) Then Inline($CD/$05) Else
   Begin
     lcount := succ(getmaxx) DIV 8;
     i := (succ(lcount) * 12) mod 216;
     writeln(lst,#27,#51,chr(i));  {Adjust line}
     writeln(lst,#27,#51,#24);     {Set line feed to 24/216}
     Writeln(lst);

     while (lcount > 0) do
      Begin
        Write(Lst,#27,#75,Chr(Succ(GetMaxY) MOD 256),Chr(Succ(GetMaxY) DIV 256));
        LC8 := LCount*8;
        For I := 0 To GetMaxY Do
         begin
           bite := 0;
           if getpixel(lc8-1,i) <> black then bite := bite + 128;
           if getpixel(lc8-2,i) <> black then bite := bite + 64;
           if getpixel(lc8-3,i) <> black then bite := bite + 32;
           if getpixel(lc8-4,i) <> black then bite := bite + 16;
           if getpixel(lc8-5,i) <> black then bite := bite + 8;
           if getpixel(lc8-6,i) <> black then bite := bite + 4;
           if getpixel(lc8-7,i) <> black then bite := bite + 2;
           if getpixel(lc8-8,i) <> black then bite := bite + 1;
           write(lst,chr(bite));
         end;
        Writeln(Lst);
        Dec(LCount);
      End;

     writeln(lst,#27,#50);          {Reset feed to 1/6}
   End;
End;
```

"DOS.PAS"

```
{-------------------------------------------------------------------------}
{ This procedure is used to execute the command line passed by the        }
{ transputer.                                                             }
{                                                                         }
Procedure TPExec;
Var Command, Parameter : String255;
Begin
  Command := Get_String;
  Parameter := Get_String;
  Exec(Command,Parameter);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to return the date to the transputer.            }
{                                                                         }
Procedure TPGetDate;
Var year, month, day, dayofweek : Word;
Begin
  GetDate(Year,Month,Day,DayOfWeek);
  Send_Word(Year);
  Send_Word(Month);
  Send_Word(Day);
  Send_Word(DayOfWeek);
End;


{-------------------------------------------------------------------------}
{ This procedure is used to return the time to the transputer.            }
{                                                                         }
Procedure TPGetTime;
Var hour, minute, second, sec100 : Word;
Begin
  GetTime(Hour,Minute,Second,Sec100);
  Send_Word(Hour);
  Send_Word(Minute);
  Send_Word(Second);
  Send_Word(Sec100);
End;

{-------------------------------------------------------------------------}
{ This procedure initiates the DOS interrupt call.                        }
{                                                                         }
Procedure TPInterrupt;
Var Regs  : Registers;
    IntNo : Byte;
Begin
  IntNo := Get_Byte;
  With Regs Do
   begin
     ax := get_word;
     bx := get_word;
     cx := get_word;
     dx := get_word;
     bp := get_word;
     si := get_word;
     di := get_word;
     ds := get_word;
     es := get_word;
     flags := get_word;
   end;

  Intr(IntNo,Regs);
```

```
  With Regs Do
   begin
     send_word(ax);
     send_word(bx);
     send_word(cx);
     send_word(dx);
     send_word(bp);
     send_word(si);
     send_word(di);
     send_word(ds);
     send_word(es);
     send_word(flags);
   end;
End;

{-------------------------------------------------------------------------}
{ This procedure will write to a specified port.                          }
{                                                                         }
Procedure TPWritePort;
Var PortNo : Word;
Begin
  PortNo := Get_Word;
  Port[PortNo] := Get_Byte;
End;
```

"TRANUTIL.PAS"

```
Unit TranUtil;

Interface

Uses Crt, Dos, Graph;

CONST
     Analyse_Flag : Boolean = FALSE;
     SectorSize   : Word = 512;


TYPE
     String255  = String[255];

     WordType = Record
                  Case Boolean Of
                    TRUE  : (W      : Word);
                    FALSE : (B1, B2 : Byte);
                  End;

     IntType =  Record
                  Case Boolean Of
                    TRUE  : (I      : Integer);
                    FALSE : (B1, B2 : Byte);
                  End;

     LongType = Record
                 Case Boolean Of
                   TRUE  : (Long      : LongInt);
                   FALSE : (Low, High : Word);
                 End;

     FileSector = Array[1..512] Of Byte;
```

```
    FileType = Record
                 Fyl        : File;
                 FullName   : String255;
                 Buffer     : FileSector;
                 FSect      : LongInt;
                 BufPos     : Word;
                 FillTo     : Word;
                 Reed       : Boolean;
                 EndLine    : Boolean;
                 EndFile    : Boolean;
                 FileStatus : Boolean;
                 FileOpen   : Boolean;
               End;

VAR
  Buffer                          : String255;

  NFile                           : Array[1..20] Of FileType;

  DriverName                      : String255;
  DriverLoaded                    : Boolean;
  DriverNumber                    : Integer;

  Grab_Attention,
  KeyboardInOperation,
  PreventExit,
  InGraphicsMode,
  BreakOn                         : Boolean;

  INSTRUCTION                     : Byte;
  Memory_Size                     : LongInt;

Const Link_Base_PC           : Word = $300;
      Link_Base_Not_PC       : Word = $150;
      Link_Read_Offset       : Word = $0;
      Link_Write_Offset      : Word = $1;
      Link_In_Status_Offset  : Word = $2;
      Link_Out_Status_Offset : Word = $3;
      Link_Reset_Offset      : Word = $10;
      Link_Error_Offset      : Word = $10;
      Link_Analyse_Offset    : Word = $11;


Var TimeOutCount,
    MaxLinkTime,
    ResetCount,
    AnalyseCount,
    Link_Read,
    Link_Write,
    Link_In_Status,
    Link_Out_Status,
    Link_Reset,
    Link_Error,
    Link_Analyse                  : Word;

    ParamName1, ParamName2, PathParam  : String[128];

  Procedure Bell;
  Procedure Blip;
  Function KeyIsPressed : Boolean;
  Function ReadAKey : Char;
  Procedure Get_Instruction(Var Bite : Byte);
  Function Get_String : String255;
  Procedure Send_String(Strng : String255);
  Function Get_Long : LongInt;
  Procedure Send_Long(Wrd : LongInt);
  Procedure TPSystemInfo;
  Procedure TPBootSubsystem;
  Procedure TPBootFileSubSys;
```

```
Implementation
Const  Version = '2.0';

{-------------------------------------------------------------------------}
{ This procedure is used to give a beep for 1/2 a second.                 }
{                                                                         }
Procedure Bell;
Begin
  Sound(500);
  Delay(500);
  NoSound;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to give a blip for 1/10 a second.                }
{                                                                         }
Procedure Blip;
Begin
  Sound(950);
  Delay(10);
  NoSound;
End;

{-------------------------------------------------------------------------}
{ This function returns the number of open files (text and other).        }
{                                                                         }
Function NumberOfAssignedFiles : Byte;
Var I, Total : Byte;
Begin
  Total := 0;
  For I := 1 To 20 Do
   if nfile[i].filestatus then inc(total);
  NumberOfAssignedFiles := Total;
End;

{-------------------------------------------------------------------------}
{ This procedure displays the message given on the bottom line of the screen }
{ until a key has been pressed.  Note: It does not kill any text which was   }
{ on the screen previously.                                               }
{                                                                         }
Procedure DisplayHelp(Message : String255);
Type TextLineType = Array[1..80] Of Word;

Var TH, TW           : Integer;
    TS               : TextSettingsType;
    VS               : ViewPortType;
    FS               : FillSettingsType;
    OldX, OldY, OldC : Word;
    GMem             : Pointer;
    TMem1, TMem2     : TextLineType;
    ActualLine1      : TextLineType Absolute $B000:$0F00;
    ActualLine2      : TextLineType Absolute $B800:$0F00;
    F,X,Y            : Byte;
Begin
  If InGraphicsMOde
   Then Begin  {WAS IN GRAPHICS MODE}
         {KEEP CURRENT SETTINGS.}
         oldx := getx;
         oldy := gety;
         oldc := getcolor;
         getfillsettings(fs);
         getviewsettings(vs);
         gettextsettings(ts);

         {SET TO OUR REQUIREMENTS}
         setviewport(0,0,getmaxx,getmaxy,TRUE);
         settextstyle(defaultfont,horizdir,1);
         settextjustify(lefttext,bottomtext);
         setfillstyle(emptyfill,0);
         setcolor(getmaxcolor);
```

```
          {SAVE THE SCREEN BELOW}
          tw := textwidth(message);
          th := textheight(message);
          getmem(gmem,imagesize(0,0,tw,th));
          getimage(0,getmaxy-th,tw,getmaxy,gmem^);

          {CLEAR DISPLAY AREA AND DISPLAY TEXT.}
          bar(0,getmaxy-th,tw,getmaxy);
          outtextxy(0,getmaxy,message);

          {WAIT UNTIL A KEY HAS BEEN PRESSED.}
          repeat until keypressed;

          {RESTORE SCREEN}
          putimage(0,getmaxy-th,gmem^,normalput);
          freemem(gmem,imagesize(1,1,tw,th));

          {RESTORE SETTINGS.}
          settextjustify(ts.horiz,ts.vert);
          settextstyle(ts.font,ts.direction,ts.charsize);
          setviewport(vs.x1, vs.y1, vs.x2, vs.y2, vs.clip);
          setfillstyle(fs.pattern,fs.color);
          setcolor(oldc);
          moveto(oldx,oldy);
        End
   Else Begin  {WAS IN TEXT MODE}
          {KEEP CURRENT SETTINGS}
          x := wherex;  y := wherey;

          {KEEP DISPLAY LINE OF BOTH PAGES.}
          tmem1 := actualline1;
          tmem2 := actualline2;

          {DISPLAY MESSAGE}
          gotoxy(1,25);
          write(message);  clreol;

          {WAIT UNTIL A KEY IS PRESSED.}
          repeat until keypressed;

          {RESTORE BOTTOM LINES OF BOTH SCREENS}
          actualline1 := tmem1;
          actualline2 := tmem2;

          {RETURN TO OLD SETTINGS.}
          gotoxy(x,y);
        End;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to reset the transputer.                         }
{ This is done by pulsing RESET while holding ANALYSE low.                }
{                                                                         }
Procedure Reset_Link;
Var I : Word;
Begin
  {DEASSERT ANALYSE AND RESET}
  Port[Link_Analyse] := 0;
  Port[Link_Reset] := 0;

  {WAIT}
  For I := 0 To ResetCount Do begin {NOTHING} end;

  {PULSE RESET}
  Port[Link_Reset] := $FF;                          {ASSERT RESET}
  For I := 0 To ResetCount Do begin {NOTHING} end;  {WAIT}
  Port[Link_Reset] := 0;                            {DEASSERT RESET}
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to reset the transputer and set it in analyse mode. }
{ This is done by pulsing RESET while holding ANALYSE high.                 }
{                                                                           }
Procedure Reset_Analyse_Link;
Var I : Word;
Begin
  {DEASSERT ANALYSE AND RESET}
  Port[Link_Reset] := 0;
  Port[Link_Analyse] := 0;

  {WAIT}
  For I := 0 To ResetCount Do begin {NOTHING} end;

  {ASSERT ANALYSE}
  Port[Link_Analyse] := $FF;

  {WAIT}
  For I := 0 To AnalyseCount Do begin {NOTHING} end;

  {PULSE RESET}
  Port[Link_Reset] := $FF;                          {ASSERT RESET}
  For I := 0 To ResetCount Do begin {NOTHING} end;  {WAIT}
  Port[Link_Reset] := 0;                            {DEASSERT RESET}

  {DEASSERT ANALYSE}
  Port[Link_Analyse] := 0;
End;

{-------------------------------------------------------------------------}
{ This function will return whether there is an error with the transputer.  }
{                                                                           }
Function Transputer_Error : Boolean;
Begin
  Transputer_Error := ((Port[Link_Error] AND 1) = 0);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to try and save the status of the server before    }
{ exiting to dos.  In order to do this it requires 5K of disk space.        }
{ If none exists on the current drive, it allows the user to specify where  }
{ to save the data.                                                         }
{                                                                           }
Procedure ExitToDos;
Type GSCType = Array[1..1] Of Byte;
Var  TempStr            : String255;
     I, OffSt           : Word;
     D, NoFiles, NoProc : Byte;
     Space              : LongInt;
     GotSpace, Abort    : Boolean;
     Ch                 : Char;
     SaveFile           : Text;
     ScreenFile         : File;
     FillInfo           : FillSettingsType;
     FillPattern        : FillPatternType;
     LineInfo           : LineSettingsType;
     Palette            : PaletteType;
     TextInfo           : TextSettingsType;
     ViewPort           : ViewPortType;
     SSize              : Word;
     Ptr                : Pointer;
     GSCMem             : ^GSCType;
     GDriver, GMode     : Integer;
Begin
 {CHECK IF THERE IS ENOUGH SPACE.}
  {Determine Space Required.}
  If InGraphicsMode Then Space := ImageSize(0,0,GetMaxX,GetMaxY){Save Screen.}
                    Else Space := 4000;
  Space := Space + 2048;
```

```
{Get current Drive}
GetDir(0,TempStr);
D := ORD(TempStr[1]) - 64;   Abort := FALSE;
Repeat
  gotspace := (diskfree(d) > Space);
  if not gotspace then
   Begin
     displayhelp('No Room on current drive to store current transputer operating data');
     ch := readkey;
     displayhelp('Enter Drive # to store transputer data (A, B, C etc. <X> Aborts');
     ch := upcase(readkey);
     abort := (ch = 'X');
     d := ORD(CH) - 64;
   End;
Until GotSpace Or Abort;
If GotSpace Then
 begin   {SAVE THE INFORMATION}
   assign(savefile,'ALSERVER.DAT');
   rewrite(savefile);
   writeln(savefile,keyboardinoperation);
   if keyboardinoperation then writeln(savefile,instruction);

   writeln(savefile,driverloaded);       {GRAPHICS DRIVER ?}
   if driverloaded then
    begin
      writeln(savefile,drivername);      {YES, SAVE NAME WHERE LOADED.}

      {SAVE GRAPHICS SETTINGS}
      writeln(savefile,drivernumber);
      writeln(savefile,ingraphicsmode);
    end;
   if ingraphicsmode then  {SAVE GRAPHICS SETTINGS}
    begin
      writeln(savefile,getgraphmode);
      writeln(savefile,getbkcolor,' ',getcolor,' ',getx,' ',gety);

      getfillsettings(fillinfo);
      writeln(savefile,fillinfo.pattern,' ',fillinfo.color);
      if (fillinfo.pattern = userfill) then
       begin
         getfillpattern(fillpattern);
         writeln(savefile,fillpattern[1],' ',fillpattern[2],' ',
                          fillpattern[3],' ',fillpattern[4],' ',
                          fillpattern[5],' ',fillpattern[6],' ',
                          fillpattern[7],' ',fillpattern[8]);
       end;

      getlinesettings(lineinfo);
      with lineinfo do writeln(savefile,linestyle,' ',pattern,' ',thickness);

      getpalette(palette);
      write(savefile,palette.size,' ');
      if (palette.size <> 0) Then
       for i := 0 to pred(palette.size) do
         write(savefile,palette.colors[i],' ');
      writeln(savefile);

      gettextsettings(textinfo);
      with textinfo do
       writeln(savefile,font,' ',direction,' ',charsize,' ',horiz,' ',vert);

      getviewsettings(viewport);
      with viewport do
       writeln(savefile,x1,' ',y1,' ',x2,' ',y2);
      writeln(savefile,viewport.clip);

      {SAVE THE SCREEN}
      setviewport(0,0,getmaxx,getmaxy,FALSE);
      SSize := ImageSize(0,0,GetMaxX,GetMaxY);
      writeln(savefile,ssize);
```

```
      GetMem(Ptr,SSize);
      GetImage(0,0,GetMaxX,GetMaxY,Ptr^);
      Assign(ScreenFile,'ALSERVER.SCR');
      Rewrite(ScreenFile,1);
      GSCMem := Ptr;
      {For I := 1 To SSize Do   write(ScreenFile,GSCMem^[I]);}
      BlockWrite(ScreenFile,GSCMem^[1],SSize);
      Close(ScreenFile);
      FreeMem(Ptr,SSize);
    end else
    begin   {SAVE TEXT SETTINGS}
      writeln(savefile,wherex,' ',wherey);
      gdriver := detect;
      detectgraph(gdriver,gmode);
      if (gdriver = hercmono) Then OffSt := $0
                              Else OffSt := $800;
      writeln(savefile,offst);

      {Save Text Screen}
      Assign(ScreenFile,'ALSERVER.SCR');
      Rewrite(ScreenFile,1);
      {For I := 0 To 3999 Do   write(ScreenFile,Mem[$B000+OffSt:I]);}
      BlockWrite(ScreenFile,Mem[$B000+OffSt:0],4000);
      Close(ScreenFile);
    end;

  {SAVE THE FILE INFORMATION}
  NoFiles := NumberOfAssignedFiles;
  Writeln(SaveFile,NoFiles);       {NUMBER OF FILES BEING PROCESSED.}

  D := 0;  NoProc := 0;
  While (NoProc < NoFiles) AND (D < 20) Do
   begin
     inc(d);
     if nfile[d].filestatus then    {FILE IS ASSIGNED SO SAVE STATUS}
     with nfile[d] do
      Begin
        Inc(NoProc);                          {Next File Found.}
        Writeln(SaveFile,FullName);           {Full file name.}
        Writeln(SaveFile,D);                  {Save File Handle.}

        Writeln(SaveFile,FileOpen);           {OPEN ?}
        If FileOpen Then
         begin
           Writeln(SaveFile,Reed);            {READ OR WRITE}
           Space := FSect + BufPos;
           Writeln(SaveFile,Space);

           {$I-}
           {CLOSE THE FILE.}
           If Not(Reed) AND (BufPos > 0) Then
             {FLUSH BUFFER}
             BlockWrite(Fyl,Buffer,BufPos);

           Close(Fyl);
         end;

        {$I+}

        FileStatus := FALSE;
        FileOpen := FALSE;
      End;
    end;

  Close(SaveFile);
  halt(5);
 end;
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to read keys into the keyboard buffer.           }
{                                                                         }
Procedure Update_Buffer;

Const  AskString :
       String = '(T)erminate Server, (E)xit to Dos, (C)ontinue, (I)gnore, (F)lush Keyboard';

       TerminateString :
       String = 'Are you sure you want to terminate (Y/N) ?';

Var Ch               : Char;
    Add              : Boolean;
Begin
  While KeyPressed Do
   begin
     add := TRUE;

     ch := readkey;

     if (ch = #3) AND BreakOn
      Then
       Begin
         {Find out what to do.}
         blip;
         repeat
           displayhelp(askstring);
           ch := upcase(readkey);
           if (ch = 'T') Then
            Begin
              Bell;
              DisplayHelp(TerminateString);
              ch := upcase(readkey);
              if ch = 'Y' then ch := 'T';
            End;
         until (ch IN ['T','E','C','I','F']);

         {EXECUTE COMMAND}
         case ch of
          'T' : Begin
                  GotoXY(1,24);
                  Reset_Link;
                  If Transputer_Error Then
                   begin
                     Reset_Analyse_Link;
                     Reset_Link;
                   end;
                  If Transputer_Error
                   Then Halt(1)    {RESET TRANSPUTER FAILED.    }
                   Else Halt(5);   {RESET TRANSPUTER SUCCESSFUL.}
                End;
          'E' : Begin
                  If Not PreventExit Then ExitToDos;
                  add := FALSE;
                End;
          'C' : if (length(Buffer) < 255) then buffer := buffer + #3;
          'I' : add := FALSE;
          'F' : Begin
                  Buffer := '';
                  Add := False;
                End;
         end;
       End
      Else
        {ENSURE ROOM IN BUFFER, THEN ADD ELSE BLIP.}
        if (length(Buffer) < 255) AND (ch <> #3)
         Then buffer := buffer + ch
         Else Blip;
   end;
End;
```

```
{-------------------------------------------------------------------------}
{ This function simulates the keypressed function.                        }
{                                                                         }
Function KeyIsPressed : Boolean;
Begin
  Update_Buffer;
  KeyIsPressed := (Length(Buffer) > 0);
End;

{-------------------------------------------------------------------------}
{ This function simulates the readkey function.                           }
{                                                                         }
Function ReadAKey : Char;
Var Ch : Char;
Begin
  Repeat Until KeyIsPressed;
  Ch := Buffer[1];
  Delete(Buffer,1,1);
  ReadAKey := Ch;
End;

{-------------------------------------------------------------------------}
{ This procedure tries to get a byte from the transputer.                 }
{                                                                         }
Procedure Get_Instruction(Var Bite : Byte);
Var Got : Boolean;
Begin
  Got := FALSE;
  Repeat
    If Grab_Attention Then Begin
      Grab_Attention := FALSE;
      Update_Buffer
    End Else
    If ((Port[Link_In_Status] AND 1) = 1) Then
     begin
       Bite := Port[Link_Read];
       Got := TRUE;
     end
    Else if keypressed then Update_Buffer;
  Until Got;
End;

{$I INLINE.PAS}

{-------------------------------------------------------------------------}
{ This function attempts to read a string from the transputer.            }
{                                                                         }
Function Get_String : String255;
Var St      : String255;
    I,J,Tmp : Byte;
Begin
  I := Get_Byte;
  For J := 1 To I Do
    st[j] := CHR(get_byte);

  st[0] := CHR(i);
  Get_String := St;
End;


{-------------------------------------------------------------------------}
{ This procedure tries to get a long word from the transputer.            }
{                                                                         }
Function Get_Long : LongInt;
Var Temp : LongType;
Begin
  Temp.Low := Get_Word;
  Temp.High := Get_Word;
  Get_Long := Temp.Long;
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure tries to send a long word to the transputer.             }
{                                                                         }
Procedure Send_Long(Wrd : LongInt);
Var Temp : LongType;
Begin
  Temp.Long := Wrd;
  Send_Word(Temp.Low);
  Send_Word(Temp.High);
End;

{-------------------------------------------------------------------------}
{ This procedure tries to get a byte from the transputer.  It will abort  }
{ if the transputer does not send the byte within a certain time  period. }
{                                                                         }
Procedure Time_Get_Byte(Var Bite : Byte; Var Error : Boolean);
Var Gotten : Boolean;
    I      : Word;
Begin
  I := 0;   Gotten := FALSE;
  Repeat
    If ((Port[Link_In_Status] AND 1) = 1) Then
     begin
       Bite := Port[Link_Read];
       Gotten := TRUE;
     end;
    Inc(I);
  Until Gotten OR (I > MaxLinkTime);
  Error := Not(Gotten);
End;

{-------------------------------------------------------------------------}
{ This procedure tries to send a byte to the transputer.  It will abort   }
{ if the transputer does not acknowledge the byte within a certain time   }
{ period.                                                                 }
{                                                                         }
Procedure Time_Send_Byte(Bite : Byte; Var Error : Boolean);
Var Sent : Boolean;
    I    : Word;
Begin
  I := 0;   Sent := FALSE;
  Repeat
    If ((Port[Link_Out_Status] AND 1) = 1) Then
     begin
       Port[Link_Write] := Bite;
       Sent := TRUE;
     end;
    Inc(I);
  Until Sent OR (I > MaxLinkTime);
  Error := Not(Sent);
End;

{-------------------------------------------------------------------------}
{ This procedure tries to send a double word to the transputer.           }
{                                                                         }
Procedure Time_Send_Double_Word(W1,W2 : Word; Var Error : Boolean);
Var Temp : WordType;
Begin
  Temp.W := W1;
  Time_Send_Byte(Temp.B1,Error);   If Error Then Exit;
  Time_Send_Byte(Temp.B2,Error);   If Error Then Exit;
  Temp.W := W2;
  Time_Send_Byte(Temp.B1,Error);   If Error Then Exit;
  Time_Send_Byte(Temp.B2,Error);
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure tries to get a double word from the transputer.          }
{                                                                         }
Procedure Time_Get_Double_Word(Var W1,W2 : Word; Var Error : Boolean);
Var Temp : WordType;
Begin
  Time_Get_Byte(Temp.B1,Error);   If Error Then Exit;
  Time_Get_Byte(Temp.B2,Error);   If Error Then Exit;
  W1 := Temp.W;

  Time_Get_Byte(Temp.B1,Error);   If Error Then Exit;
  Time_Get_Byte(Temp.B2,Error);
  W2 := Temp.W;
End;

{-------------------------------------------------------------------------}
{ This function will return whether the Host exists on the PC or not.     }
{                                                                         }
Function Host_In_Computer : Boolean;
Label BoardNotFound;
Const PeekVal         : Byte = $1;
      PokeVal         : Byte = $0;
      MostNegIntHigh  : Word = $8000;
      MostNegIntLow   : Word = $0000;
      TestPatternHigh : Word = $4567;
      TestPatternLow  : Word = $0123;
      LocHighWord     : Word = $0000;
      LocLowWord      : Word = $0000;
      MinInt          : LongInt = $80000000;

Var Exists, Error, Found              : Boolean;
    Peek1, Peek2                      : Word;
    ThisAddr, OldAddr, OldVal, Counter : LongInt;
Begin
  Exists := FALSE;   Memory_Size := 0;

  {TEST IF WE CAN PEEK A VALUE FROM THE BOARD.}
  Time_Send_Byte(PeekVal,Error);
  If Error Then Goto BoardNotFound;

  {SEND THE ADDRESS TO PEEK.}
  Time_Send_Double_Word(MostNegIntLow,MostNegIntHigh,Error);
  If Error Then Goto BoardNotFound;

  {GET THE VALUE PEEKED.}
  Time_Get_Double_Word(Peek1,Peek2,Error);
  If Error Then Goto BoardNotFound;

  {NOW POKE THE ADDRESS.}
  Send_Byte(PokeVal);                          {Poke CMD.}
  Send_Word(MostNegIntLow);                    {Poke Address.}
  Send_Word(MostNegIntHigh);                   { "      "    }
  Send_Word(TestPatternLow);                   {Poke Value.}
  Send_Word(TestPatternHigh);                  { "     "   }

  {PEEK AT WHAT WE POKED.}
  Send_Byte(PeekVal);                          {Peek Command.}
  Send_Word(MostNegIntLow);                    {Peek Address.}
  Send_Word(MostNegIntHigh);                   { "      "    }
```

```
  {CHECK IF THE SAME WHAT WE POKED.}
  If (Get_Word = TestPatternLow) AND (Get_Word = TestPatternHigh) Then
   begin
     exists := TRUE;       {BOARD FOUND.}

     {RESTORE OLD PEEKED VALUE.}
     Send_Byte(PokeVal);                {Poke CMD.}
     Send_Word(MostNegIntLow);          {Poke Address.}
     Send_Word(MostNegIntHigh);         { "      "    }
     Send_Word(Peek1);                  {Poke Old Peeked Value.}
     Send_Word(Peek2);                  { "     "   "      "   }
   end;

  {FIND THE MEMORY SIZE}
  Counter := $20000000;
  OldAddr := MinInt OR Counter;
  OldVal  := OldAddr;
  Send_Byte(PokeVal);  Send_Long(OldAddr);  Send_Long(OldAddr);
  Found := FALSE;
  Repeat
    counter := counter shr 1;
    thisaddr := counter OR minint;
    send_byte(pokeval); send_long(thisaddr);  send_long(thisaddr);
    send_byte(peekval); send_long(oldaddr);
    oldval := get_long;
    found := (OldVal <> ThisAddr)
  Until Found;
  Memory_Size := Counter SHL 1;

  If Analyse_Flag Then Reset_Analyse_Link
                  Else Reset_Link;

  {SET WHAT WE FOUND}
  BoardNotFound:
  Host_In_Computer := Exists;
End;

{-------------------------------------------------------------------------}
{ This procedure will boot the open transputer file passed.               }
{ It will return an error if the boot was unsuccessful.                   }
{                                                                         }
Procedure Send_Boot_File(Var Fyl : File;   Message : String255);
Var Bytes      : Array[1..512] Of Byte;
    NumRead, I : Word;
    Total      : LongInt;
    Error      : Boolean;
Begin
  Total := 0;  Error := FALSE;
  write(message);
  repeat
    BlockRead(Fyl,Bytes,512,NumRead);
    i := 1;
    while (i <= numread) AND Not(Error) Do
     begin
       Time_Send_Byte(Bytes[i],Error);
       Inc(I);
     end;
    Total := Total + I;
  until (numread = 0) OR (numread <> 512) Or Error;
  close(fyl);
  if error then Begin
                  WriteLn('Boot Failed');
                  WriteLn('Timeout occurred on ',Total,'th byte sent');
                  WriteLn;
                  Halt(2);
                End
           else Writeln('Boot Successful')
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to define the global transputer link variable.   }
{                                                                         }
Procedure Initialise_Link;
Const  PC_ID : Byte = $FF;
       XT_ID : Byte = $FE;
       XT_I2 : Byte = $FB;
       AT_ID : Byte = $FC;
Var Link_Base : Word;
    PC_Type   : Byte Absolute $FFFF:$000E;
Begin
  If (PC_Type = PC_ID)
    then Link_Base := Link_Base_PC
    else Link_Base := Link_Base_Not_PC;

  Link_Read          := Link_Base + Link_Read_Offset;
  Link_Write         := Link_Base + Link_Write_Offset;
  Link_In_Status     := Link_Base + Link_In_Status_Offset;
  Link_Out_Status    := Link_Base + Link_Out_Status_Offset;
  Link_Reset         := Link_Base + Link_Reset_Offset;
  Link_Error         := Link_Base + Link_Error_Offset;
  Link_Analyse       := Link_Base + Link_Analyse_Offset;


  TimeOutCount := 64000;
  ResetCount   := 10000;
  AnalyseCount := 10000;
  MaxLinkTime  := 32000;
End;

{-------------------------------------------------------------------------}
{ This procedure will send the string passed to the transputer.           }
{                                                                         }
Procedure Send_String(Strng : String255);
Var Lngth, I : Byte;
Begin
  Lngth := Length(Strng);
  For I := 0 To Lngth Do
    Send_Byte(Ord(Strng[I]));
End;

{-------------------------------------------------------------------------}
{ This function will return TRUE if the transputer running status files are  }
{ found, and are loaded correctly.  Otherwise it will return FALSE.       }
{                                                                         }
Function Running_Status : Boolean;
Type GSCType = Array[1..1] Of Byte;
Var   SaveFile    : Text;
      ScreenFile  : File;
      Return      : Boolean;
      Str         : String255;
      X,Y,I,OffSt : Word;
      GMode       : Integer;
      ImSize      : Word;
      Ptr         : Pointer;
      GSCMem      : ^GSCType;
      FillInfo    : FillSettingsType;
      FillPattern : FillPatternType;
      LineInfo    : LineSettingsType;
      Palette     : PaletteType;
      TextInfo    : TextSettingsType;
      ViewPort    : ViewPortType;
      NoFiles, H  : Byte;
      FPosn       : LongInt;
Begin
  Return := FALSE;   KeyboardInOperation := FALSE;

  {$I-}
  Assign(SaveFile,PathParam+'ALSERVER.DAT');
  Reset(SaveFile);
  Return := (IOResult = 0);
```

```
If Return Then
 begin      {STATUS FILE FOUND.}
   readln(savefile,str);
   keyboardinoperation := (str = 'TRUE');   {START WITH KEYBOARD INSTRUCTION?}
   if keyboardinoperation then readln(savefile,instruction);

   readln(savefile,str);
   driverloaded := (str = 'TRUE');   {LOAD GRAPHICS DRIVER}
   ingraphicsmode := FALSE;
   if driverloaded then
    begin
      readln(savefile,drivername);
      readln(savefile,drivernumber);
      gmode := 0;
      initgraph(drivernumber,gmode,drivername);
      return := (drivernumber > 0);

      readln(savefile,str);
      ingraphicsmode := (str = 'TRUE');
      if ingraphicsmode AND Return
       Then Begin
              Readln(SaveFile,GMode);
              SetGraphMode(GMode);
              Return := (GraphResult = grOK);
            End
       Else RestoreCrtMode;
    end;

   if return then
   if ingraphicsmode then
    begin
      {GET AND SET COLORS}
      read(savefile,x,y);
      setbkcolor(x);    setcolor(y);

      {GET X and Y CP}
      readln(savefile,x,y);

      {GET AND SET FILL SETTINGS}
      with fillinfo do  readln(savefile,pattern,color);
      if (fillinfo.pattern = userfill) then
       begin
         readln(savefile,fillpattern[1],fillpattern[2],
                         fillpattern[3],fillpattern[4],
                         fillpattern[5],fillpattern[6],
                         fillpattern[7],fillpattern[8]);
         setfillpattern(fillpattern,fillinfo.color);
       end else setfillstyle(fillinfo.pattern,fillinfo.color);

      {GET AND SET LINE STYLE}
      with lineinfo do
       begin
         readln(savefile,linestyle,pattern,thickness);
         setlinestyle(linestyle,pattern,thickness);
       end;

      {GET AND SET PALETTE}
      read(savefile,palette.size);
      if (palette.size <> 0) Then
       begin
         for i := 0 to pred(palette.size) do
           read(savefile,palette.colors[i]);
         setallpalette(palette);
       end;
      readln(savefile);
```

```
  {GET AND SET TEXT INFO}
  with textinfo do
   Begin
     readln(savefile,font,direction,charsize,horiz,vert);
     settextstyle(font,direction,charsize);
     settextjustify(horiz,vert);
   End;

  {GET VIEWPORT}
  with viewport do
   readln(savefile,x1,y1,x2,y2);
  readln(savefile,str);
  viewport.clip := (str = 'TRUE');

  {GET IMAGE SIZE}
  readln(savefile,imsize);
  return := (ioresult = 0);

  {CLOSE DATA FILE AND DELETE.}
  close(savefile);
  erase(savefile);

  if return then
   Begin  {O.K. TO CONTINUE, SO SEE IF SCREEN FILE AVAILABLE.}
     Assign(ScreenFile,PathParam+'ALSERVER.SCR');
     Reset(ScreenFile,1);
     Return := (IOResult = 0);
   End;

  if return then
   Begin  {O.K, SCREEN FILE AVAILABLE, SO LOAD SCREEN.}
     GetMem(Ptr,ImSize);
     GSCMem := Ptr;
     BlockRead(screenfile,gscmem^[1],ImSize,I);

     SetViewPort(0,0,GetMaxX,GetMaxY,FALSE);
     PutImage(0,0,GSCMem^,NormalPut);
     FreeMem(Ptr,ImSize);

     {CLOSE AND DELETE SCREEN FILE.}
     Close(ScreenFile);
     Erase(ScreenFile);
   End;

  {SET VIEWPORT}
  with viewport do  setviewport(x1,y1,x2,y2,clip);

  {SET CURSOR POSITION}
  moveto(x,y);
end else
begin  {RECALL TEXT SCREEN}
  {GET CURSOR POSITION}
  readln(savefile,x,y);
  gotoxy(x,y);

  {GET SCREEN MEMORY LOCATION}
  readln(savefile,offst);
  return := (ioresult = 0);

  {GET FILE STATUS}
  readln(savefile,nofiles);
  if (nofiles > 0) and return then
   repeat
     ReadLn(SaveFile,Str);
     ReadLn(SaveFile,H);
     Return := (IOResult = 0) AND (H > 0) AND (H <= 20);
```

```
          If Return then {Valid File Handle} With NFile[H] Do
           Begin
             fullname := str;
             assign(fyl,str);
             FileStatus := TRUE;
             ReadLn(SaveFile,Str);
             FileOpen := (Str = 'TRUE');
             If FileOpen Then
              begin
                reset(fyl,1);
                return := (ioresult = 0);
                if return then
                 Begin
                   ReadLn(SaveFile,Str);
                   NFile[H].Reed := (Str = 'TRUE');
                   ReadLn(SaveFile,FPosn);
                   Return := (IOResult = 0);
                   If Return Then  With NFile[H] Do
                    Begin
                      Seek(Fyl,FPosn);
                      BufPos := 0;
                      FSect  := FPosn;
                      If Reed
                       Then begin
                              FillTo := 0;
                              BlockRead(Fyl,Buffer,SectorSize,FillTo);
                              EndFile := (FillTo <> SectorSize);
                            end
                       Else begin
                              FillTo := SectorSize;
                              EndFile := TRUE;
                            end;
                    End;
                 End;
               end;
           End;
           Dec(NoFiles);
        until (nofiles = 0) OR Not(Return);

       {CLOSE DATA FILE AND DELETE.}
       close(savefile);
       erase(savefile);

       if return then
        Begin  {O.K. TO CONTINUE, SO SEE IF SCREEN FILE AVAILABLE.}
          Assign(ScreenFile,PathParam+'ALSERVER.SCR');
          Reset(ScreenFile,1);
          Return := (IOResult = 0);
        End;

       if return then
        Begin  {O.K, SCREEN FILE AVAILABLE, SO LOAD SCREEN.}
          BlockRead(screenfile,mem[$B000+OffSt:0],4000,i);
          Return := (IOResult = 0);

          {CLOSE AND DELETE SCREEN FILE.}
          Close(ScreenFile);
          Erase(ScreenFile);
        End;
      end;
   end;
  {$I+}
  Running_Status := Return;
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure sends the booted transputer the system information acquired }
{ before the system was actually booted.  The system information is the      }
{ transputer type and memory size.                                          }
{                                                                           }
Procedure TPSystemInfo;
Begin
  Send_Long(Memory_Size);
End;

{-------------------------------------------------------------------------}
{ This procedure does the sending of the subsystem file to the Host.        }
{                                                                           }
Procedure SendSubsystemFile(FName : String255; Flag : Boolean);
Var Fyl   : File;
    Error : Boolean;
    FSize : LongType;
Begin
  If FName = '/*'
   Then begin
          Time_Send_Double_Word(0,0,Error);
          If Error Then writeln('Host not expecting Quadputer File');
        end
   Else begin
          Assign(Fyl,FName);
          {$I-}  Reset(Fyl,1);  {$I+}
          If (IOResult = 0) Then
           begin
             FSize.Long := FileSize(Fyl);
             Time_Send_Double_Word(FSize.Low,FSize.High,Error);
             If Error
              Then writeln('Host not expecting Quadputer File')
              Else Send_Boot_File(Fyl,'Booting Quadputer .....     ');
           end Else
           begin   {FILE NOT FOUND.}
             FSize.Long := -1;
             Time_Send_Double_Word(FSize.Low,FSize.High,Error);
             If Error
              Then writeln('Host not expecting Quadputer File')
              Else if flag then
                   writeln('Quadputer Boot File ',FName,' not found !');
           end;
        end;
End;

{-------------------------------------------------------------------------}
{ This procedure will send the second boot file (quadputer) to the Host in  }
{ order to allow the Host to boot up the quadputer.                         }
{                                                                           }
Procedure TPBootSubsystem;
Var Error : Boolean;
    LW    : LongType;
Begin
  {SEE IF NAME EXISTS}
  If (ParamName2 <> '')
   Then  SendSubsystemFile(ParamName2,TRUE) {TRY SEND FILE TO BOOT QUADPUTER}
   Else  begin
           LW.Long := -1;
           Time_Send_Double_Word(LW.Low,LW.High,Error);       {FILE NOT FOUND}
           If Error Then writeln('Host not expecting Quadputer File');
         end;
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure will send the boot file passed by the Host to the Host.  }
{ The it will send the 32 bit integer -1 if the file is not found.        }
{                                                                         }
Procedure TPBootFileSubSys;
Begin
  {GET BOOT FILE NAME}
  SendSubsystemFile(Get_String,FALSE);
End;

{-------------------------------------------------------------------------}
{ This procedure processes the parameter line.                            }
{                                                                         }
Procedure Process_Parameter_Line;
Const ErrConst : Word = 65535;
Var Option          : String255;
    P, Paddr, Value  : Word;

    Function Number(Nstr : String) : Word;
    Var Value, Q : Word;
    Begin
      If Nstr[1] = '#' Then begin  {HEX ADDRESS}
        delete(nstr,1,1);
        for q := 1 to length(nstr) do nstr[Q] := Upcase(nstr[Q]);

        value := 0;
        for q := 1 to length(NStr) Do Begin
          if (nstr[q] >= 'A') AND (nstr[q] <= 'F')
              Then Value := Value*16 + Ord(NStr[Q])-55
          else if (nstr[q] >= '0') AND (nstr[q] <= '9')
              Then Value := Value*16 + Ord(NStr[Q])-48
          else value := errconst;
        End;
      end else begin
        Val(NStr,Value,Q);
        If Q <> 0 Then Value := ErrConst;
      end;

      Number := Value;
    End;
Begin
  For P := 1 TO ParamCount Do
   begin
     Option := ParamStr(P);
     If (Option[1] = '/') AND (Upcase(Option[2]) = 'L') Then
      begin
        {SET LINK PORT BASE ADDRESS FOR PC}
        Delete(Option,1,2);
        Paddr := Number(Option);
        If Paddr <> ErrConst Then Link_Base_Not_PC := PAddr;
      end else if (Option[1] = '/') AND (Upcase(Option[2]) = 'K') Then
      begin
        {INITIALISE KEYSTROKES}
        Delete(Option,1,2);
        Repeat
          Value := Pos('^',Option);
          If Value <> 0 Then
           begin
             Delete(Option,Value,1);
             If Option[Value] <> '^' Then begin
               Paddr := Number(Copy(Option,Value,3));
               If (Paddr = ErrConst) OR (Paddr > 255)
               Then Delete(Option,Value,3) Else Begin
                 Delete(Option,Value,2);
                 Option[Value] := Chr(PAddr);
               End;
             end;
           end;
        Until (Value = 0);
        Buffer := Option;
      end else if (Option[1] = '/') AND (Upcase(Option[2]) = 'P') Then
```

```
      begin
        {INITIALISE PATH FOR ALSERVER FILES}
        Delete(Option,1,2);
        PathParam := Option;
        If (PathParam[Length(PathParam)] <> '\') OR
           (PathParam[Length(PathParam)] <> ':') Then
         pathparam := pathparam + '\';
      end else if (Option[1] = '/') AND (Upcase(Option[2]) = 'B') Then
      begin
        {INITIALISE BREAK ON/OFF}
        Delete(Option,1,2);
        BreakOn := Option = '+';
      end else if paramname1 = ''
      then paramname1 := Option else if paramname2 = ''
      then paramname2 := Option;
   end;
End;

{++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++}
Var  I     : Integer;
     Fyl   : File;
     FName : String255;

Begin
  {SET UP THE FILE VARIABLES.}
  For I := 1 to 20 Do
   Begin
     nfile[i].filestatus := FALSE;
     nfile[i].fileopen := FALSE;
   End;

  {INITIALISE GRAPHICS VARIABLES}
  InGraphicsMode := FALSE;    DriverLoaded := FALSE;
  DriverName := '';

  {INITIALISE THE ROOT TRANSPUTER}
  WriteLn;
  WriteLn('Transputer Pascal Server V'+Version+'  -- (C) Alex Schuilenburg 1989');


  {GET THE PARAMETER NAMES}
  Buffer := '';       {KEYBOARD BUFFER}
  Grab_Attention := FALSE;
  KeyboardInOperation := FALSE;
  PreventExit := FALSE;
  ParamName1 := '';   ParamName2 := '';    PathParam := '';
  Process_Parameter_Line;

  {BOOT ?}
  If (ParamName1 = '') Then
   begin
     {TRANSPUTER STATUS FILE DOES NOT EXIST SO TRANSPUTER NOT RUNNING.}
     Writeln;
     Writeln('Transputer Boot file OR /* Required as parameter');
     Halt(0);
   end;

  If ParamName1 = '/*' Then {ASSUME TRANSPUTER RUNNING.}
   begin
     {TRY AND GET TRANSPUTER RUNNING INFORMATION.}
     Initialise_Link;
     If Not Running_Status Then
      begin
        Writeln;
        Writeln('Error loading transputer data files -- CONTACT AUTHOR');
        Halt(1);
      end;
   end   Else              {ASSUME BOOT TRANSPUTER FILE.}
```

```
  begin
    Write('Initialising Transputer          ');
    Initialise_Link;

    If Analyse_Flag Then Reset_Analyse_Link
                    Else Reset_Link;
    If Host_In_Computer Then
     begin
       WriteLn('Host found');
       WriteLn('Memory size is ',Memory_Size);
     end Else
     begin
       writeln('Host not found');
       halt(1);
    end;

    {TRY OPEN AND BOOT FILE}
    FName := ParamName1;
    Assign(Fyl,FName);
    {$I-}  Reset(Fyl,1);  {$I+}
    If (IOResult = 0)
     Then  Send_Boot_File(Fyl,'Booting Root Transputer .....   ')
     Else  begin
             Writeln;
             Writeln('Transputer Boot File ',FNAme,' not found !');
             Halt(4);
           end;
    {If Transputer_Error Then      (COMMENTED OUT 24 April 1989)
     begin                         (Mini-Cluster does not like!)
       Writeln('Error in booting transputer');
       Halt(2);
     end;}
  end;
End.
```

"FILEHAND.PAS"

```
Unit FileHand;

Interface

Uses Dos, Printer, TranUtil;

  Procedure TPAssignFile;
  Procedure TPResetFile;
  Procedure TPRewriteFile;
  Procedure TPAppendFile;
  Procedure TPCloseFile;
  Procedure TPRenameFile;
  Procedure TPDeleteFile;
  Procedure TPEndOfFile;
  Procedure TPFileSize;
  Procedure TPGetFAttr;
  Procedure TPSetFAttr;
  Procedure TPFileExists;
  Procedure TPFileError;
  Procedure TPReadTextLine;
  Procedure TPReadTextNum;
  Procedure TPReadBlock;
  Procedure TPReadFile(NumBytes : Word);
  Procedure TPWriteTextLine;
  Procedure TPWriteText;
  Procedure TPWriteBlock;
  Procedure TPWriteFile(NumBytes : Byte);

Implementation

Var OldExit      : Pointer;
    ErrorCode    : Byte;
```

```
{$I IOERRNOS.PAS}
{$I INLINE.PAS}

{-------------------------------------------------------------------------}
{ This procedure is used to try and assign a file for the transputer.     }
{    Input  : Normal File Name                                            }
{    Result : File Handle is returned to the transputer                   }
{                                                                         }
Procedure TPAssignFile;
Var Name : String255;
    I    : Byte;
Begin
  {GET FILE NAME}
  Name := Get_String;

  {Search for available handle.}
  I := 1;
  While (I < 20)  AND  NFile[I].FileStatus Do
    inc(i);

  If NFile[I].FileStatus
   Then begin
          send_byte(0);                          {INVALID FILE HANDLE}
          errorcode := toomanyfiles;             {TOO MANY FILES}
        end
   Else begin
          send_byte(i);                          {FILE HANDLE}
          NFile[I].FileStatus := TRUE;
          Assign(NFile[I].Fyl,Name);             {ASSIGN FILE}
          errorcode := noerror;
          NFile[I].FullName := FExpand(name);
        end;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to reset the file passed.                        }
{                                                                         }
Procedure TPResetFile;
Var Handle : Byte;
    AFile  : File;
Begin
  Handle := Get_Byte;
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[Handle].FileStatus)
    then errorcode := InvalidHandle
    else Begin
           {$I-}
           Reset(NFile[Handle].Fyl,1);
           {$I+}
           ErrorCode := IOResult;
           If (ErrorCode <> 0) Then Begin
             {CREATE THE FILE}
             {$I-}
             Rewrite(NFile[Handle].Fyl,1);
             Close(NFile[Handle].Fyl);
             Reset(NFile[Handle].Fyl,1);
             {$I+}
             ErrorCode := IOResult;
           End;
           With NFile[Handle] Do
            begin
              FileOpen := TRUE;
              BufPos := 0;
              FillTo := 0;
              Reed := TRUE;
              FSect := 0;
              If (ErrorCode = NoError) Then
               begin   {START WITH INFO IN THE BUFFER}
                 blockread(fyl,buffer,SectorSize,fillto);
                 endfile := (fillto <> SectorSize);
               end else endfile := TRUE;
            end;
```

```
        End;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to rewrite the file passed.                      }
{                                                                         }
Procedure TPRewriteFile;
Var Handle : Byte;
Begin
  Handle := Get_Byte;
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[Handle].FileStatus)
    then errorcode := InvalidHandle
    else Begin
           {$I-}
           Rewrite(NFile[Handle].Fyl,1);
           {$I+}
           ErrorCode := IOResult;
           With NFile[Handle] Do
            begin
              FileOpen := TRUE;
              FSect  := 0;
              BufPos := 0;
              FillTo := SectorSize;
              Reed := FALSE;
              EndFile := TRUE;
            end;
         End;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to append to the file passed.                    }
{                                                                         }
Procedure TPAppendFile;
Var Handle : Byte;
    FPosn  : LongInt;
Begin
  Handle := Get_Byte;
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[Handle].FileStatus)
    then errorcode := InvalidHandle
    else With NFile[Handle] Do Begin
           {$I-}
           Reset(Fyl,1);
           {$I+}
           ErrorCode := IOResult;
           If (ErrorCode <> NoError) Then Begin
             {CREATE THE FILE}
             {$I-}
             Rewrite(NFile[Handle].Fyl,1);
             {$I+}
             FPosn := 0;
           End Else Begin {POSITION AT THE END OF THE FILE}
             FPosn := FileSize(Fyl);
             Seek(Fyl,fposn);
           End;

           FileOpen := TRUE;
           BufPos := 0;
           FSect  := FPosn;
           FillTo := SectorSize;
           EndFile := TRUE;
           Reed := FALSE;
         End;
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to close the file passed.                        }
{                                                                         }
Procedure TPCloseFile;
Var Handle : Byte;
Begin
  Handle := Get_Byte;
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[Handle].FileStatus)
   then errorcode := InvalidHandle else
   Begin   {NORMAL FILE}
     ErrorCode := NoError;
     With NFile[Handle] Do
     If Not(Reed) AND (BufPos > 0) Then
      begin  {FLUSH BUFFER}
        {$I-}
        BlockWrite(Fyl,Buffer,BufPos);
        {$I+}
        ErrorCode := IOResult;
      end;
     {$I-}
     Close(NFile[Handle].Fyl);
     {$I+}
     If (ErrorCode = NoError) Then  ErrorCode := IOResult;
     NFile[handle].FileStatus := FALSE;
     NFile[handle].FileOpen := FALSE;
   End;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to delete the file passed.                       }
{                                                                         }
Procedure TPRenameFile;
Var Handle        : Byte;
    NewName       : String255;
Begin
  Handle := Get_Byte;
  NewName := Get_String;

  If (Handle = 0) OR (Handle > 20) Or Not(NFile[handle].FileStatus)
   then errorcode := InvalidHandle else
   Begin
     {$I-}
     Rename(NFile[Handle].Fyl,NewName);
     {$I+}
     ErrorCode := IOResult;
   End;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to delete the file passed.                       }
{                                                                         }
Procedure TPDeleteFile;
Var Handle : Byte;
Begin
  Handle := Get_Byte;
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[handle].FileStatus)
   then errorcode := InvalidHandle else
   Begin   {NORMAL FILE}
     {$I-}
     Erase(NFile[Handle].Fyl);
     {$I+}
     ErrorCode := IOResult;
   End;
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to return whether the end of the file has been   }
{ reached yet.                                                            }
{                                                                         }
Procedure TPEndOfFile;
Var Handle, RetCode : Byte;
Begin
  Handle := Get_Byte;
  RetCode := 0;
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[handle].FileStatus)
    then errorcode := InvalidHandle else
    Begin   {NORMAL FILE}
      With NFile[Handle] Do
       if EndFile AND (bufpos = fillto)
        then retcode := 1
        else retcode := 0;
      {$I+}
      ErrorCode := IOResult;
    End;

  Send_Byte(RetCode);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to return the size of the file to the transputer.  }
{                                                                         }
Procedure TPFileSize;
Var Handle : Byte;
    Size   : LongInt;
Begin
  Handle := Get_Byte;
  Size := 0;
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[handle].FileStatus)
    then errorcode := InvalidHandle else
    Begin   {NORMAL FILE}
      {$I-}
      Size := FileSize(NFile[Handle].Fyl) + NFile[Handle].BufPos;
      ErrorCode := IOResult;
      {$I+}
    End;

  Send_Long(Size);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to return the attribute of the file passed to the  }
{ transputer.                                                             }
{                                                                         }
Procedure TPGetFAttr;
Var Handle, Attribute : Byte;
    Attr              : Word;
Begin
  Attr := 0;
  Handle := Get_Byte;
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[handle].FileStatus)
    then errorcode := InvalidHandle else
    Begin   {NORMAL FILE}
      {$I-}
      GetFAttr(NFile[Handle],Attr);
      {$I+}
      ErrorCode := IOResult;
      NFile[handle].FileStatus := FALSE;
    End;
  Attribute := Attr;
  Send_Byte(Attribute);
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to set the attribute of the file passed by the   }
{ transputer.                                                             }
{                                                                         }
Procedure TPSetFAttr;
Var Handle : Byte;
Begin
  Handle := Get_Byte;
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[handle].FileStatus)
    then errorcode := InvalidHandle else
    Begin   {NORMAL FILE}
      {$I-}
      SetFAttr(NFile[Handle],Get_Byte);
      {$I+}
      ErrorCode := IOResult;
      NFile[handle].FileStatus := FALSE;
    End;
End;

{-------------------------------------------------------------------------}
{ This procedure will return a true or false whether a file (assigned)    }
{ already exists or not.                                                  }
{                                                                         }
Procedure TPFileExists;
Var Handle : Byte;
    Does   : Boolean;
    S      : SearchRec;
Begin
  FindFirst(Get_String,AnyFile,S);
  ErrorCode := DosError;
  If (ErrorCode = NoError) Then Send_Byte(1) Else Send_Byte(0);
End;

{-------------------------------------------------------------------------}
{ This procedure returns the error code of the last file operation.       }
{                                                                         }
Procedure TPFileError;
Begin
  Send_Byte(ErrorCode);
  ErrorCode := NoError;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to read a specific number of bytes from the file }
{ passed.                                                                 }
{                                                                         }
Procedure ReadBlock(Handle      : Byte;     NumToRead : Word;
                    Var Block;          Var NumRead   : Word );

Type Memory = Array[0..65520] Of Byte;

Var NumCanRead, TransNo, I  : Word;
    NoMore                  : Boolean;

Begin
  NumRead := 0;        NoMore := FALSE;
  ErrorCode := NoError;

  {TRY READ BLOCK FROM FILE}
  If (Handle < 1) OR (Handle > 20) Or Not(NFile[handle].FileStatus)
    then errorcode := InvalidHandle else
     with NFile[handle] do
      if reed then
        repeat
          {FILL BLOCK USING (BUFPOS + 1) TO FILLTO}
          NumCanRead := FillTo - BufPos;
          If (NumCanRead > 0)
           Then
```

```
            Begin
              {FIND HOW MANY BYES TO READ OUT OF FILE'S BUFFER}
              if (numcanread >= (numtoread - numread))
                then  begin
                        transno := numtoread - numread;
                        nomore := TRUE;
                      end
                else  begin
                        transno := numcanread;
                        nomore := FALSE;
                      end;

              {READ THE NUMBER OF BYTES}
              move(buffer[BufPos+1],memory(block)[numread],transno);
              numread := numread + transno;
              BufPos := BufPos + TransNo;
              If (BufPos >= fillto) Then
               begin
                 bufpos := 0;
                 blockread(fyl,buffer,SectorSize,fillto);
                 endfile := (fillto <> SectorSize);
                 if not endfile then fsect := fsect + sectorsize;
               end;
            End
          Else If EndFile Then ErrorCode := DiskReadError
          Else Begin
                 bufpos := 0;
                 blockread(fyl,buffer,SectorSize,fillto);
                 endfile := (fillto <> SectorSize);
                 if not endfile then fsect := fsect + sectorsize;
               End;
       until nomore OR (ErrorCode <> NOError)

     else {TRYING TO READ FROM A FILE OPEN FOR WRITING}
          errorcode := FileNotInput;
End;

{---------------------------------------------------------------------------}
{ This procedure is used to read a line from the text file passed.          }
{                                                                           }
Procedure TPReadTextLine;
Var Strng                   : String255;
    Handle, MaxSize, LSize : Byte;
    Done                    : Boolean;
    NRead                   : Word;
Begin
  Handle := Get_Byte;
  MaxSize := Get_Byte;
  Done := FALSE;   LSize := 0;
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[handle].FileStatus)
   then errorcode := InvalidHandle else With NFile[Handle] Do
   repeat
     readblock(handle,1,strng[lsize+1],nread);
     if (nread = 1) then
      if (strng[lsize+1] = #13) OR (strng[lsize+1] = #10) {<CR>,<LF> ?}
       then begin  {SKIP <LF> AS WELL ?}
              done := TRUE;
              if buffer[bufpos+1] = 10 then
                readblock(handle,1,strng[lsize+1],nread);
            end
       else begin
              inc(lsize);
              if lsize = maxsize then done := TRUE;
              if lsize = 255 then lsize := 254;
            end;
   until  done or (nread = 0);

  Strng[0] := Chr(LSize);
  Send_String(Strng);
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to read a text number from the text file passed.  }
{                                                                          }
Procedure TPReadTextNum;
Var Strng                  : String255;
    Handle, MaxSize, LSize : Byte;
    Done                   : Boolean;
    NRead                  : Word;
    ECodeInt               : Integer;
    Num                    : Real;
Begin
  Handle := Get_Byte;
  MaxSize := Get_Byte;
  Done := FALSE;   LSize := 0;  ErrorCode := NoError;  Strng := '';
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[handle].FileStatus)
   then errorcode := InvalidHandle else With NFile[Handle] Do
   begin
     {SKIP LEADING SPACES}
     repeat
       readblock(handle,1,strng[lsize+1],nread);
     until (strng[lsize+1] > ' ') AND (strng[lsize+1] <= #127) OR (NRead = 0);

     if (Nread = 0) AND (lsize = 0) Then ErrorCode := DiskReadError Else
      repeat
        {READ THE STRING}
        inc(lsize);
        readblock(handle,1,strng[lsize+1],nread);
        done := (nread = 1) AND ((strng[lsize+1] <= ' ') OR (strng[lsize+1] > #127));
      until done or (nread = 0);

     If LSize > MaxSize Then LSize := MaxSize;
     Strng[0] := Chr(LSize);
     Val(Strng,Num,ECodeInt);
     If (ECodeInt <> NOError) Then ErrorCode := InvalidNumeric;
   end;

  Send_String(Strng);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to read the appropriate number of bytes to the    }
{ transputer for a read from the file passed.                              }
{                                                                          }
Procedure TPReadBlock;
Type MemArray = Array[1..65521] Of Byte;
     MemPtr   = ^MemArray;
Var Handle                 : Byte;
    NumToRead, NumRead, I  : Word;
    Memory                 : MemPtr;
Begin
  Handle := Get_Byte;
  NumToRead := Get_Word;           {GET BLOCK SIZE}
  GetMem(Memory,NumToRead);        {GET MEMORY FOR BLOCK}

  ReadBlock(Handle,NumToRead,Memory^,NumRead);

  Send_Word(NumRead);              {SEND HOW MANY BYTES READ.}
  For I := 1 To NumRead Do         {SEND BLOCK}
    Send_Byte(Memory^[I]);

  FreeMem(Memory,NumToRead);       {FREE THAT MEMORY USED.}
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to read the appropriate number of bytes to the   }
{ transputer for a read from the file passed.                             }
{                                                                         }
Procedure TPReadFile(NumBytes : Word);
Var Handle        : Byte;
    NumRead, I    : Word;
    Memory        : Array[1..8] Of Byte;
Begin
  Handle := Get_Byte;
  ReadBlock(Handle,NumBytes,Memory,NumRead);

  {SEND THE BYTES - irrespective whether all were read or not}
  {                 as the transputer expects them anyway.   }

  For I := 1 To NumBytes Do
    Send_Byte(Memory[I]);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to write a specific number of bytes to the file  }
{ passed.                                                                 }
{                                                                         }
Procedure WriteBlock(Handle      : Byte;     NumToWrite : Word;
                     Var Block;          Var NumWritten : Word);

Type Memory = Array[0..65520] Of Byte;

Var NumCanWrite, TransNo, I  : Word;
    NoMore                   : Boolean;

Begin
  NumWritten := 0;      ErrorCode := NoError;

  {TRY READ BLOCK FROM FILE}
  If (Handle < 1) OR (Handle > 20) Or Not(NFile[handle].FileStatus)
    then errorcode := InvalidHandle
    else
     with NFile[handle] do
      if not reed then
        repeat
          {FILL BLOCK USING (BUFPOS + 1) TO FILLTO}
          NumCanWrite := FillTo - BufPos;
          If (NumCanWrite > 0)
           Then
             Begin
               {FIND HOW MANY BYES TO WRITE TO THE FILE'S BUFFER}
               if (numcanwrite > (numtowrite - numwritten))
                 then  begin
                         transno := numtowrite - numwritten;
                         nomore := TRUE;
                       end
                 else  begin
                         transno := numcanwrite;
                         nomore := FALSE;
                       end;

               {READ THE NUMBER OF BYTES}
               move(memory(block)[numwritten],buffer[BufPos+1],transno);
               numwritten := numwritten + transno;
               BufPos := BufPos + TransNo;
             End
           Else Begin
                  bufpos := 0;
                  blockwrite(fyl,buffer,SectorSize,fillto);
                  if (fillto <> SectorSize) then
                    Begin
                      errorcode := DiskWriteError;
                      nomore := TRUE;
                    End
                    Else fsect := fsect + sectorsize;
```

```
                End;
        until nomore

      else {TRYING TO WRITE TO A FILE OPEN FOR READING}
           errorcode := FileNotOutput;
End;

{---------------------------------------------------------------------------}
{ This procedure is used to write a text line to the text file passed.      }
{                                                                           }
Procedure TPWriteTextLine;
Var Strng         : String255;
    Handle        : Byte;
    NWritten      : Word;
Const Newline : Array[1..2] of Byte = (13,10);
Begin
  Handle := Get_Byte;
  If (Handle = 255)  {THE PRINTER}
  then begin
          {$I-}
          writeln(LST,Get_String);
          {$I+}
          ErrorCode := IOResult;
       end
  Else
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[handle].FileStatus)
    then errorcode := InvalidHandle
    else begin
          Strng := Get_String;
          WriteBlock(Handle,Length(Strng),Strng[1],NWritten);
          WriteBlock(Handle,2,Newline,NWritten);
        end;
End;

{---------------------------------------------------------------------------}
{ This procedure is used to write some text to the text file passed.        }
{                                                                           }
Procedure TPWriteText;
Var Strng         : String255;
    Handle        : Byte;
    NWritten      : Word;
Begin
  Handle := Get_Byte;
  If (Handle = 255)  {THE PRINTER}
  then begin
          {$I-}
          writeln(LST,Get_String);
          {$I+}
          ErrorCode := IOResult;
       end
  Else
  If (Handle = 0) OR (Handle > 20) Or Not(NFile[handle].FileStatus)
    then errorcode := InvalidHandle
    else begin
          Strng := Get_String;
          WriteBlock(Handle,Length(Strng),Strng[1],NWritten);
        end;
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to write the appropriate number of bytes to the  }
{ transputer for a write from the file passed.                            }
{                                                                         }
Procedure TPWriteBlock;
Type MemArray = Array[1..65521] Of Byte;
     MemPtr   = ^MemArray;
Var Handle                      : Byte;
    NumToWrite, NumWritten, I   : Word;
    Memory                      : MemPtr;
Begin
  Handle := Get_Byte;
  NumToWrite := Get_Word;           {GET BLOCK SIZE}
  GetMem(Memory,NumToWrite);        {GET MEMORY FOR BLOCK}
  For I := 1 To NumToWrite Do       {GET BLOCK}
    Memory^[I] := Get_Byte;

  WriteBlock(Handle,NumToWrite,Memory^,NumWritten);

  Send_Word(NumWritten);                    {SEND HOW MANY BYTES WRITTEN.}
  FreeMem(Memory,NumToWrite);               {FREE THAT MEMORY USED.}
End;

{-------------------------------------------------------------------------}
{ This procedure is used to write the appropriate number of bytes to the  }
{ transputer for a write from the file passed.                            }
{                                                                         }
Procedure TPWriteFile(NumBytes : Byte);
Var Handle         : Byte;
    Memory         : Array[1..8] Of Byte;
    NumWritten, I  : Word;
Begin
  Handle := Get_Byte;
  For I := 1 To NumBytes Do       {GET BLOCK}
    Memory[I] := Get_Byte;
  WriteBlock(Handle,NumBytes,Memory,NumWritten);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to close any open files.                         }
{                                                                         }
{$F+} Procedure CloseFileSystem;  {$F-}
Var Handle : Byte;
Begin
  {Close any open files.}
  For Handle := 1 To 20 Do
   If NFile[handle].FileStatus Then
    begin
      With NFile[Handle] Do
       If FileOpen Then Begin
         If Not(Reed) AND (BufPos > 0) Then
          begin  {FLUSH BUFFER}
            {$I-}
            BlockWrite(Fyl,Buffer,BufPos);
            {$I+}
            ErrorCode := IOResult;
          end;

         {$I-}
         Close(NFile[Handle].Fyl);
         {$I+}
       End;

      ErrorCode := IOResult;
      NFile[handle].FileStatus := FALSE;
      NFile[handle].FileOpen := FALSE;
    end;

  {NORMAL TURBO EXIT FROM UNIT.}
  ExitProc := OldExit;
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure initialises the file variables.                          }
{                                                                         }
Var I : Integer;
Begin
  {INITIALISE FILE ERROR CODE}
  ErrorCode := NoError;

  {SET UP FINALISATION PROCEDURE.}
  OldExit := ExitProc;
  ErrorCode := NoError;
  ExitProc := @CloseFileSystem;
End.
```

"GRAPHICS.PAS"

```
Unit Graphics;

Interface

Uses Dos, Graph, TranUtil;

  Procedure TPInitGraph;
  Procedure TPRestoreCRTMode;
  Procedure TPCloseGraphics;
  Procedure TPSetGraphMode;
  Procedure TPSetViewPort;
  Procedure TPDetectGraph;
  Procedure TPSetLineStyle;
  Procedure TPSetFillPattern;
  Procedure TPSetFillStyle;
  Procedure TPArc;
  Procedure TPBar;
  Procedure TPBar3D;
  Procedure TPCircle;
  Procedure TPEllipse;
  Procedure TPFillEllipse;
  Procedure TPDrawPoly;
  Procedure TPFillPoly;
  Procedure TPFloodFill;
  Procedure TPGetArcCoords;
  Procedure TPLine;
  Procedure TPLineTo;
  Procedure TPLineRel;
  Procedure TPMoveRel;
  Procedure TPMoveTo;
  Procedure TPPieSlice;
  Procedure TPPutPixel;
  Procedure TPOutTextXY;
  Procedure TPRectangle;
  Procedure TPSector;
  Procedure TPSetTextJustify;
  Procedure TPSetTextStyle;
  Procedure TPGetMaxX;
  Procedure TPGetMaxY;
  Procedure TPGetAspectRatio;
  Procedure TPGraphResult;
  Procedure TPSetColor;
  Procedure TPSetBkColor;
  Procedure TPGetColor;
  Procedure TPGetBkColor;

Implementation

Var OldExit : Pointer;

{$I INLINE.PAS}
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to detect the graphics drivers and mode.         }
{                                                                         }
Procedure TPDetectGraph;
Var GDriver, GMode : Integer;
Begin
  DetectGraph(GDriver,GMode);
  Send_Word(GDriver);
  Send_Word(GMode);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to set the line style, pattern and thickness for  }
{ lines to be drawn.                                                      }
{                                                                         }
Procedure TPSetLineStyle;
Var Style, Pattern, Thickness : Word;
Begin
  Style := Get_Word;
  Pattern := Get_Word;
  Thickness := Get_Word;
  If DriverLoaded Then SetLineStyle(Style,Pattern,Thickness);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to set the fill pattern.                         }
{                                                                         }
Procedure TPSetFillPattern;
Var Pattern : FillPatternType;
    Color   : Word;
    I       : Byte;
Begin
  For I := 1 To 8 Do
    Pattern[I] := Get_Byte;
  Color := Get_Word;
  If DriverLoaded Then SetFillPattern(Pattern,Color)
End;

{-------------------------------------------------------------------------}
{ This procedure is used to set the fill style.                           }
{                                                                         }
Procedure TPSetFillStyle;
Var Pattern, Color   : Word;
Begin
  Pattern := Get_Word;
  Color := Get_Word;
  If DriverLoaded Then SetFillStyle(Pattern,Color)
End;

{-------------------------------------------------------------------------}
{ This procedure is used to draw an arc on the graphics screen.           }
{                                                                         }
Procedure TPArc;
Var X, Y                    : Integer;
    StAngle, EndAngle, Radius : Word;
Begin
  X := Get_Word;
  Y := Get_Word;
  StAngle := Get_Word;
  EndAngle := Get_Word;
  Radius := Get_Word;
  If DriverLoaded Then Arc(X,Y,StAngle,EndAngle,Radius);
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to draw a bar.                                   }
{                                                                         }
Procedure TPBar;
Var X1, Y1, X2, Y2 : Integer;
Begin
  X1 := Get_Word;
  Y1 := Get_Word;
  X2 := Get_Word;
  Y2 := Get_Word;
  If DriverLoaded Then Bar(X1,Y1,X2,Y2);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to draw a 3D bar.                                }
{                                                                         }
Procedure TPBar3D;
Var X1, Y1, X2, Y2 : Integer;
    Depth          : Word;
    Top            : Boolean;
Begin
  X1 := Get_Word;
  Y1 := Get_Word;
  X2 := Get_Word;
  Y2 := Get_Word;
  Depth := Get_Word;
  Top := (Get_Byte <> 0);
  If DriverLoaded Then Bar3D(X1,Y1,X2,Y2,Depth,Top);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to draw a circle.                                }
{                                                                         }
Procedure TPCircle;
Var X, Y   : Integer;
    Radius : Word;
Begin
  X := Get_Word;
  Y := Get_Word;
  Radius := Get_Word;
  If DriverLoaded Then Circle(X,Y,Radius);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to draw an ellipse on the graphics screen.       }
{                                                                         }
Procedure TPEllipse;
Var X, Y                        : Integer;
    StAngle, EndAngle, XRad, YRad : Word;
Begin
  X := Get_Word;
  Y := Get_Word;
  StAngle := Get_Word;
  EndAngle := Get_Word;
  XRad :=.Get_Word;
  YRad := Get_Word;
  If DriverLoaded Then Ellipse(X,Y,StAngle,EndAngle,XRad,YRad);
End;
```

```
{---------------------------------------------------------------------------}
{ This procedure is used to draw and fill an ellipse on the graphics screen. }
{                                                                           }
Procedure TPFillEllipse;
Var X, Y       : Integer;
    XRad, YRad : Word;
Begin
  X := Get_Word;
  Y := Get_Word;
  XRad := Get_Word;
  YRad := Get_Word;
  If DriverLoaded Then FillEllipse(X,Y,XRad,YRad);
End;

{---------------------------------------------------------------------------}
{ This procedure is used to draw a polygon.                                 }
{                                                                           }
Procedure TPDrawPoly;
Type PointRec = Record
                  X, Y : Word;
                End;
     PointArr = Array[1..1000] Of PointRec;

Var NumPoint, I : Word;
    Mem         : ^PointArr;
Begin
  NumPoint := Get_Word;
  GetMem(Mem, NumPoint * SizeOf(PointRec));
  For I := 1 To NumPoint Do
   begin
     mem^[i].x := Get_Word;
     mem^[i].y := Get_Word;
   end;
  If DriverLoaded Then DrawPoly(NumPoint,Mem^);
  FreeMem(Mem, NumPoint * SizeOf(PointRec));
End;

{---------------------------------------------------------------------------}
{ This procedure is used to draw and fill a polygon.                        }
{                                                                           }
Procedure TPFillPoly;
Type PointRec = Record
                  X, Y : Word;
                End;
     PointArr = Array[1..1000] Of PointRec;

Var NumPoint, I : Word;
    Mem         : ^PointArr;
Begin
  NumPoint := Get_Word;
  GetMem(Mem, NumPoint * SizeOf(PointRec));
  For I := 1 To NumPoint Do
   begin
     mem^[i].x := Get_Word;
     mem^[i].y := Get_Word;
   end;
  If DriverLoaded Then FillPoly(NumPoint,Mem^);
  FreeMem(Mem, NumPoint * SizeOf(PointRec));
End;

{---------------------------------------------------------------------------}
{ This procedure is used to fill a bounded region.                          }
{                                                                           }
Procedure TPFloodFill;
Var X, Y   : Integer;
    Border : Word;
Begin
  X := Get_Word;
  Y := Get_Word;
  Border := Get_Word;
  If DriverLoaded Then FloodFill(X,Y,Border);
```

```
End;

{-------------------------------------------------------------------------}
{ This procedure is used to get coordinates of the arc last plotted.      }
{                                                                         }
Procedure TPGetArcCoords;
Var CoOrds : ArcCoordsType;
Begin
  If DriverLoaded Then GetArcCoords(CoOrds);
  With CoOrds Do
   begin
     send_word(x);
     send_word(y);
     send_word(xstart);
     send_word(ystart);
     send_word(xend);
     send_word(yend);
   end;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to draw a line.                                  }
{                                                                         }
Procedure TPLine;
Var X1, Y1, X2, Y2 : Integer;
Begin
  X1 := Get_Word;
  Y1 := Get_Word;
  X2 := Get_Word;
  Y2 := Get_Word;
  If DriverLoaded Then Line(X1,Y1,X2,Y2);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to draw a line from the current cursor position to  }
{ the position passed.                                                    }
{                                                                         }
Procedure TPLineTo;
Var X, Y : Integer;
Begin
  X := Get_Word;
  Y := Get_Word;
  If DriverLoaded Then LineTo(X,Y);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to draw a line from the current cursor position to  }
{ the relative position passed.                                           }
{                                                                         }
Procedure TPLineRel;
Var X, Y : Integer;
Begin
  X := Get_Word;
  Y := Get_Word;
  If DriverLoaded Then LineRel(X,Y);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to move the graphics cursor to a point releative to }
{ its current location.                                                   }
{                                                                         }
Procedure TPMoveRel;
Var DX, DY : Integer;
Begin
  DX := Get_Word;
  DY := Get_Word;
  If DriverLoaded Then MoveRel(DX,DY);
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to move the graphics cursor to an exact point.   }
{                                                                         }
Procedure TPMoveTo;
Var X, Y : Integer;
Begin
  X := Get_Word;
  Y := Get_Word;
  If DriverLoaded Then MoveTo(X,Y);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to draw a pie slice on the graphics screen.      }
{                                                                         }
Procedure TPPieSlice;
Var X, Y                    : Integer;
    StAngle, EndAngle, Radius : Word;
Begin
  X := Get_Word;
  Y := Get_Word;
  StAngle := Get_Word;
  EndAngle := Get_Word;
  Radius := Get_Word;
  If DriverLoaded Then PieSlice(X,Y,StAngle,EndAngle,Radius);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to plot a pixel at a specific location.          }
{                                                                         }
Procedure TPPutPixel;
Var X, Y  : Integer;
    Pixel : Word;
Begin
  X := Get_Word;
  Y := Get_Word;
  Pixel := Get_Word;
  If DriverLoaded Then PutPixel(X,Y,Pixel);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to display the text at the specified cursor      }
{ location.                                                               }
{                                                                         }
Procedure TPOutTextXY;
Var X, Y  : Integer;
    Str   : String255;
Begin
  X := Get_Word;
  Y := Get_Word;
  Str := Get_String;
  If DriverLoaded Then OutTextXY(X,Y,Str);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to plot a rectangle given two opposite corner    }
{ points.                                                                 }
{                                                                         }
Procedure TPRectangle;
Var X1, Y1, X2, Y2 : Integer;
Begin
  X1 := Get_Word;
  Y1 := Get_Word;
  X2 := Get_Word;
  Y2 := Get_Word;
  If DriverLoaded Then Rectangle(X1,Y1,X2,Y2);
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to draw and fill an ellipse on the graphics screen. }
{                                                                         }
Procedure TPSector;
Var X, Y                          : Integer;
    StAngle, EndAngle, XRad, YRad : Word;
Begin
  X := Get_Word;
  Y := Get_Word;
  StAngle := Get_Word;
  EndAngle := Get_Word;
  XRad := Get_Word;
  YRad := Get_Word;
  If DriverLoaded Then Sector(X,Y,StAngle,EndAngle,XRad,YRad);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to set the graphics text justification.          }
{                                                                         }
Procedure TPSetTextJustify;
Var Horiz, Vert : Word;
Begin
  Horiz := Get_Word;
  Vert := Get_Word;
  If DriverLoaded Then SetTextJustify(Horiz,Vert);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to set the graphics text style.                  }
{                                                                         }
Procedure TPSetTextStyle;
Var Font, Direction : Word;
    ChSize          : 1..10;
Begin
  Font := Get_Word;
  Direction := Get_Word;
  ChSize := Get_Byte;
  If DriverLoaded Then SetTextStyle(Font,Direction,ChSize);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to return the maximum X value of the graphics    }
{ screen.                                                                 }
{                                                                         }
Procedure TPGetMaxX;
Var MaxX : Word;
Begin
  If DriverLoaded Then MaxX := GetMaxX
                  Else MaxX := 0;
  Send_Word(MaxX);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to return the maximum Y value of the graphics    }
{ screen.                                                                 }
{                                                                         }
Procedure TPGetMaxY;
Var MaxY : Word;
Begin
  If DriverLoaded Then MaxY := GetMaxY
                  Else MaxY :=  0;
  Send_Word(MaxY);
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to return the aspect ratio to the transputer.    }
{                                                                         }
Procedure TPGetAspectRatio;
Var AspX, AspY : Word;
Begin
  If DriverLoaded Then GetAspectRatio(AspX,AspY)
                  Else Begin
                         AspX := 0;  AspY := 0;
                       End;
  Send_Word(AspX);
  Send_Word(AspY);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to return the result of the last graphics operation }
{ to the transputer.                                                      }
{                                                                         }
Procedure TPGraphResult;
Begin
  If DriverLoaded Then Send_Word(GraphResult)
                  Else Send_Int(grNoInitGraph);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to set the foreground color.                     }
{                                                                         }
Procedure TPSetColor;
Var Color : Integer;
Begin
  Color := Get_Word;
  If DriverLoaded Then SetColor(Color);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to set the background color.                     }
{                                                                         }
Procedure TPSetBkColor;
Var Color : Integer;
Begin
  Color := Get_Word;
  If DriverLoaded Then SetBkColor(Color);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to get the foreground color.                     }
{                                                                         }
Procedure TPGetColor;
Begin
  If DriverLoaded Then Send_Word(GetColor)
                  Else Send_Int(grNoInitGraph);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to get the background color.                     }
{                                                                         }
Procedure TPGetBkColor;
Var Color : Integer;
Begin
  If DriverLoaded Then Send_Word(GetBkColor)
                  Else Send_Int(grNoInitGraph);
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to close off the graphics system of the transputer. }
{                                                                         }
{$F+} Procedure ExitGraphics; {$F-}
Begin
  {Close down the graphics system}
  If InGraphicsMode Then CloseGraph;
  ExitProc := OldExit;
End;

{-------------------------------------------------------------------------}
{ This initialises the graphics system.                                   }

Begin
  OldExit := ExitProc;
  ExitProc := @ExitGraphics;
End.
```

(ii) Occam Source

"TPSERVER.INC"

```
-- Colors
VAL Black        IS  0:
VAL Blue         IS  1:
VAL Green        IS  2:
VAL Cyan         IS  3:
VAL Red          IS  4:
VAL Magenta      IS  5:
VAL Brown        IS  6:
VAL LightGray    IS  7:
VAL DarkGray     IS  8:
VAL LightBlue    IS  9:
VAL LightGreen   IS  10:
VAL LightCyan    IS  11:
VAL LightRed     IS  12:
VAL LightMagenta IS  13:
VAL Yellow       IS  14:
VAL White        IS  15:
VAL MaxColors    IS  15:
```

"TPSCREEN.INC"

```
VAL BW40   IS 0:    -- 40x25 B/W on color adaptor
VAL CO40   IS 1:    -- 40x25 color on color adaptor
VAL BW80   IS 2:    -- 80x25 B/W on color adaptor
VAL CO80   IS 3:    -- 80x25 color on color adaptor
VAL Mono   IS 7:    -- 80x25 on monochrome adaptor
```

"TPFILE.INC"

```
VAL NoError        IS INT   0:
VAL FileNotFound   IS INT   2:
VAL PathNotFound   IS INT   3:
VAL TooManyFiles   IS INT   4:
VAL AccessDenied   IS INT   5:
VAL InvalidHandle  IS INT   6:
VAL InvalidAccess  IS INT  12:
VAL InvalidDrive   IS INT  15:
VAL CantRemoveDir  IS INT  16:
VAL CantRename     IS INT  17:
VAL DiskReadError  IS INT 100:
VAL DiskWriteError IS INT 101:
VAL FileNotAssign  IS INT 102:
VAL FileNotOpen    IS INT 103:
VAL FileNotInput   IS INT 104:
VAL FileNotOutput  IS INT 105:
VAL InvalidNumeric IS INT 106:
VAL PRINTER        IS BYTE 255:
```

"TPGRAPH.INC"

```
-- GraphResult error return codes
VAL grOk               IS    0:
VAL grNoInitGraph      IS  (-1):
VAL grNotDetected      IS  (-2):
VAL grFileNotFound     IS  (-3):
VAL grInvalidDriver    IS  (-4):
VAL grNoLoadMem        IS  (-5):
VAL grNoScanMem        IS  (-6):
VAL grNoFloodMem       IS  (-7):
VAL grFontNotFound     IS  (-8):
VAL grNoFontMem        IS  (-9):
VAL grInvalidMode      IS (-10):
VAL grError            IS (-11):   -- generic error
VAL grIOerror          IS (-12):
```

```
VAL grInvalidFont      IS (-13):
VAL grInvalidFontNum   IS (-14):
VAL grInvalidDeviceNum IS (-15):

-- define graphics drivers
VAL Detect   IS  0:    -- requests autodetection
VAL CGA      IS  1:
VAL MCGA     IS  2:
VAL EGA      IS  3:
VAL EGA64    IS  4:
VAL EGAMono  IS  5:
VAL RESERVED IS  6:
VAL HercMono IS  7:
VAL ATT400   IS  8:
VAL VGA      IS  9:
VAL PC3270   IS  10:

-- graphics modes for each driver
VAL CGAC0      IS  0:  -- 320x200 pal 0 L/Green, L/Red, Yellow: 1 pg
VAL CGAC1      IS  1:  -- 320x200 pal 1 L/Cyan, L/Magenta, White: 1 pg
VAL CGAC2      IS  2:  -- 320x200 pal 2 Green, Red, Brown: 1 pg
VAL CGAC3      IS  3:  -- 320x200 pal 3 Cyan, Magenta, L/Gray: 1 pg
VAL CGAHi      IS  4:  -- 640x200 1 pg
VAL MCGAC0     IS  0:  -- 320x200 pal 0 L/Green, L/Red, Yellow: 1 pg
VAL MCGAC1     IS  1:  -- 320x200 pal 1 L/Cyan, L/Magenta, White: 1 pg
VAL MCGAC2     IS  2:  -- 320x200 pal 2 Green, Red, Brown: 1 pg
VAL MCGAC3     IS  3:  -- 320x200 pal 3 Cyan, Magenta, L/Gray: 1 pg
VAL MCGAMed    IS  4:  -- 640x200 1 pg
VAL MCGAHi     IS  5:  -- 640x480 1 pg
VAL EGALo      IS  0:  -- 640x200 16 color 4 pg
VAL EGAHi      IS  1:  -- 640x350 16 color 2 pg
VAL EGA64Lo    IS  0:  -- 640x200 16 color 1 pg
VAL EGA64Hi    IS  1:  -- 640x350 4 color  1 pg
VAL EGAMonoHi  IS  3:  -- 640x350 64K on card, 1 pg: 256K on card, 2 pg
VAL HercMonoHi IS  0:  -- 720x348 2 pg
VAL ATT400C0   IS  0:  -- 320x200 pal 0 L/Green, L/Red, Yellow: 1 pg
VAL ATT400C1   IS  1:  -- 320x200 pal 1 L/Cyan, L/Magenta, White: 1 pg
VAL ATT400C2   IS  2:  -- 320x200 pal 2 Green, Red, Brown: 1 pg
VAL ATT400C3   IS  3:  -- 320x200 pal 3 Cyan, Magenta, L/Gray: 1 pg
VAL ATT400Med  IS  4:  -- 640x200 1 pg
VAL ATT400Hi   IS  5:  -- 640x400 1 pg
VAL VGALo      IS  0:  -- 640x200 16 color 4 pg
VAL VGAMed     IS  1:  -- 640x350 16 color 2 pg
VAL VGAHi      IS  2:  -- 640x480 16 color 1 pg
VAL PC3270Hi   IS  0:  -- 720x350 1 pg

-- Line styles and widths for Get/SetLineStyle
VAL SolidLn    IS  0:
VAL DottedLn   IS  1:
VAL CenterLn   IS  2:
VAL DashedLn   IS  3:
VAL UserBitLn  IS  4:       -- User-defined line style

VAL NormWidth  IS  1:
VAL ThickWidth IS  3:

-- Set/GetTextStyle constants
VAL DefaultFont   IS  0:    -- 8x8 bit mapped font

VAL TriplexFont   IS  1:    -- "Stroked" fonts
VAL SmallFont     IS  2:
VAL SansSerifFont IS  3:
VAL GothicFont    IS  4:

VAL HorizDir   IS  0:       -- left to right
VAL VertDir    IS  1:       -- bottom to top

VAL UserCharSize IS BYTE 0:     -- user-defined char size
```

```
-- Clipping constants
VAL ClipOn  IS TRUE:
VAL ClipOff IS FALSE:

-- Bar3D constants
VAL TopOn  IS TRUE:
VAL TopOff IS FALSE:

-- Fill patterns for Get/SetFillStyle
VAL EmptyFill       IS  0:  -- fills area in background color
VAL SolidFill       IS  1:  -- fills area in solid fill color
VAL LineFill        IS  2:  -- --- fill
VAL LtSlashFill     IS  3:  -- /// fill
VAL SlashFill       IS  4:  -- /// fill with thick lines
VAL BkSlashFill     IS  5:  -- \\\ fill with thick lines
VAL LtBkSlashFill   IS  6:  -- \\\ fill
VAL HatchFill       IS  7:  -- light hatch fill
VAL XHatchFill      IS  8:  -- heavy cross hatch fill
VAL InterleaveFill  IS  9:  -- interleaving line fill
VAL WideDotFill     IS  10: -- Widely spaced dot fill
VAL CloseDotFill    IS  11: -- Closely spaced dot fill
VAL UserFill        IS  12: -- user defined fill

-- BitBlt operators for PutImage
VAL NormalPut     IS  0:    -- MOV
VAL XORPut        IS  1:    -- XOR
VAL OrPut         IS  2:    -- OR
VAL AndPut        IS  3:    -- AND
VAL NotPut        IS  4:    -- NOT

-- Horizontal and vertical justification for SetTextJustify
VAL LeftText   IS  0:
VAL CenterText IS  1:
VAL RightText  IS  2:

VAL BottomText IS  0:
VAL TopText    IS  2:
```

"SENDSTR.OCC"

```
PROC SendString(CHAN OF ANY in, out, VAL []BYTE str)
  BYTE i:
  INT len:
  BOOL cont:
  SEQ
    -- FIND THE LENGTH OF THE STRING
    IF
      IF j = 0 FOR SIZE str
        str[j] = 0 (BYTE)
          len := j
      TRUE
        len := SIZE str

    -- SEND THE STRING
    i := BYTE len
    out ! i                       -- Number of characters
    IF
      (len > 0)
        out ! [str FROM 0 FOR len]
      TRUE
        SKIP
:
```

"TPKEYPRO.INC"

```
-- KEYBOARD PROTOCOL   1 -  50
VAL TPKeypressed          IS BYTE   1:
VAL TPReadKey             IS BYTE   2:
VAL TPReadEchoKey         IS BYTE   3:
VAL TPReadString          IS BYTE   4:
VAL TPReadEchoString      IS BYTE   5:
VAL TPReadNumber          IS BYTE   6:
VAL TPReadEchoNumber      IS BYTE   7:
VAL TPBreakOn             IS BYTE   8:
```

"TPKEYBRD.OCC"

```
#INCLUDE "TPKeyPro.INC"

PROC KeyPressed (CHAN OF ANY in, out, BOOL Pressed)
  BYTE x:
  SEQ
    out ! TPKeypressed    -- Keypressed Function
    in ? x                -- Get result from 8086 server
    IF
      x = 0 (BYTE)                -- Set Function
        Pressed := FALSE
      TRUE
        Pressed :=~TRUE
:

PROC ReadKey (CHAN OF ANY in, out, VAL BOOL echo, BYTE ch)
  SEQ
    IF
      echo = FALSE
        out ! TPReadKey       -- Read Key Function
      TRUE
        out ! TPReadEchoKey   -- Read and echo the key
    in ? ch                   -- Get result from 8086 server
:

PROC ReadString(CHAN OF ANY in, out, VAL BOOL echo, INT len, []BYTE string)
  BYTE length:
  INT i:
  SEQ
    length := BYTE (SIZE string)
    IF
      echo = FALSE
        out ! TPReadString; length     -- Read String Function
      TRUE
        out ! TPReadEchoString; length -- Read Echo String Function

    in ? length                        -- Read the length
    len := INT length
    IF
      (len > 0)
        in ? [string FROM 0 FOR len]  -- Read the string
      TRUE
        SKIP

    IF
      len < (SIZE string)
        string[len] := 0 (BYTE)
      TRUE
        SKIP
:
```

```
PROC ReadNumber(CHAN OF ANY in, out, VAL BOOL echo, INT len, []BYTE string)
  BYTE length:
  INT i:
  SEQ
    length := BYTE (SIZE string)
    IF
      echo = FALSE
        out ! TPReadNumber; length     -- Read String Function
      TRUE
        out ! TPReadEchoNumber; length -- Read Echo String Function

    in ? length                        -- Read the length
    len := INT length
    IF
      (len > 0)
        in ? [string FROM 0 FOR len]   -- Read the string
      TRUE
        SKIP

    IF
      len < (SIZE string)
        string[len] := 0 (BYTE)                    -- Terminate String
      TRUE
        SKIP
:

PROC BreakOn(CHAN OF ANY in,out, VAL BOOL on)
  BYTE b:
  SEQ
    out ! TPBreakOn
    IF
      on
        b := 1 (BYTE)
      TRUE
        b := 0 (BYTE)
    out ! b
:
```

"TPSCRPRO.INC"

```
-- SCREEN PROTOCOL    51 - 100
VAL TPClrScr              IS BYTE  51:
VAL TPClrEOS              IS BYTE  52:
VAL TPClrEOL              IS BYTE  53:
VAL TPGotoXY              IS BYTE  54:
VAL TPNormVideo           IS BYTE  55:
VAL TPHighVideo           IS BYTE  56:
VAL TPLowVideo            IS BYTE  57:
VAL TPSound               IS BYTE  58:
VAL TPNoSound             IS BYTE  59:
VAL TPWhereX              IS BYTE  60:
VAL TPWhereY              IS BYTE  61:
VAL TPTextBackground      IS BYTE  62:
VAL TPTextColor           IS BYTE  63:
VAL TPTextMode            IS BYTE  64:
VAL TPWriteString         IS BYTE  65:
VAL TPWriteLnString       IS BYTE  66:
VAL TPPrintScreen         IS BYTE  80:
```

"TPSCREEN.OCC"

```
#INCLUDE "TPScrPro.INC"

PROC ClrScr (CHAN OF ANY in, out)
  SEQ
    out ! TPClrScr            -- ClrScr Function
:
```

```
PROC ClrEOS (CHAN OF ANY in, out)
  SEQ
    out ! TPClrEOS              -- ClrEOS Function
:

PROC ClrEOL (CHAN OF ANY in, out)
  SEQ
    out ! TPClrEOL              -- ClrEOL Function
:

PROC GotoXY(CHAN OF ANY in, out, VAL INT x,y)
  VAL [4]BYTE bx RETYPES x:
  VAL [4]BYTE by RETYPES y:
  SEQ
    out ! TPGotoXY ; bx[0] ; by[0]     -- GotoXY Function
:

PROC NormVideo (CHAN OF ANY in, out)
  SEQ
    out ! TPNormVideo           -- Normal Video Function
:

PROC HighVideo (CHAN OF ANY in, out)
  SEQ
    out ! TPHighVideo           -- High Video Function
:

PROC LowVideo (CHAN OF ANY in, out)
  SEQ
    out ! TPLowVideo            -- Low Video Function
:

PROC Sound(CHAN OF ANY in, out, VAL INT hertz)
  VAL [4]BYTE small RETYPES hertz:
  SEQ
    out ! TPSound; [small FROM 0 FOR 2]    -- Sound Function and Send the hertz
:

PROC NoSound (CHAN OF ANY in, out)
  SEQ
    out ! TPNoSound             -- NoSound Function
:

PROC WhereX(CHAN OF ANY in, out, INT x)
  BYTE temp:
  SEQ
    out ! TPWhereX              -- WhereX Function
    in ? temp                   -- Get X position
    x := INT temp
:

PROC WhereY(CHAN OF ANY in, out, INT y)
  BYTE temp:
  SEQ
    out ! TPWhereY              -- WhereY Function
    in ? temp                   -- Get Y position
    y := INT temp
:

PROC TextBackground(CHAN OF ANY in, out, VAL INT x)
  VAL [4]BYTE small RETYPES x:
  SEQ
    out ! TPTextBackground ; small[0]    -- Text Background Function
:

PROC TextColor(CHAN OF ANY in, out, VAL INT x)
  VAL [4]BYTE small RETYPES x:
  SEQ
    out ! TPTextColor ; small[0]         -- Text Color Function
:
```

```
PROC TextMode(CHAN OF ANY in, out, VAL INT x)
  VAL [4]BYTE small RETYPES x:
  SEQ
    out ! TPTextMode ; small[0]          -- Text Mode Function
:

PROC WriteString(CHAN OF ANY in, out, VAL []BYTE str)
  #USE "SendStr"
  SEQ
    out ! TPWriteString                  -- Write String Function
    SendString(in, out, str)     -- String to display
:

PROC WriteLnString(CHAN OF ANY in, out, VAL []BYTE str)
  #USE "SendStr"
  SEQ
    out ! TPWriteLnString        -- Writeln String Function
    SendString(in, out, str)     -- String to display
:

PROC PrintScreen(CHAN OF ANY in, out)
  SEQ
    out ! TPPrintScreen                  -- Print Screen Function
:
```

"TPFILPRO.INC"

```
-- FILE PROTOCOL     101 - 150
VAL TPAssignFile          IS BYTE 102:
VAL TPResetFile           IS BYTE 103:
VAL TPRewriteFile         IS BYTE 104:
VAL TPAppendFile          IS BYTE 105:
VAL TPCloseFile           IS BYTE 106:
VAL TPRenameFile          IS BYTE 107:
VAL TPDeleteFile          IS BYTE 108:
VAL TPEndOfFile           IS BYTE 109:
VAL TPFileError           IS BYTE 110:

VAL TPReadTextLine        IS BYTE 111:
VAL TPReadTextNum         IS BYTE 112:
VAL TPReadBlock           IS BYTE 113:
VAL TPReadBool            IS BYTE 114:
VAL TPReadByte            IS BYTE 115:
VAL TPReadInt             IS BYTE 116:
VAL TPReadInt16           IS BYTE 117:
VAL TPReadInt32           IS BYTE 118:
VAL TPReadInt64           IS BYTE 119:
VAL TPReadReal32          IS BYTE 120:
VAL TPReadReal64          IS BYTE 121:

VAL TPWriteTextLine       IS BYTE 131:
VAL TPWriteText           IS BYTE 132:
VAL TPWriteBlock          IS BYTE 133:
VAL TPWriteBool           IS BYTE 134:
VAL TPWriteByte           IS BYTE 135:
VAL TPWriteInt            IS BYTE 136:
VAL TPWriteInt16          IS BYTE 137:
VAL TPWriteInt32          IS BYTE 138:
VAL TPWriteInt64          IS BYTE 139:
VAL TPWriteReal32         IS BYTE 140:
VAL TPWriteReal64         IS BYTE 141:
```

"TPFILE.OCC"

```
#INCLUDE "TPFilPro.INC"

PROC AssignFile(CHAN OF ANY in, out, VAL []BYTE str, BYTE handle)
  #USE "SendStr"
  SEQ
    out ! TPAssignFile                    -- Assign File Function
    SendString(in, out, str)
    in ? handle                           -- Get File Handle
:

PROC ResetFile(CHAN OF ANY in, out, VAL BYTE handle)
  SEQ
    out ! TPResetFile; handle
:

PROC RewriteFile(CHAN OF ANY in, out, VAL BYTE handle)
  SEQ
    out ! TPRewriteFile; handle
:

PROC AppendFile(CHAN OF ANY in, out, VAL BYTE handle)
  SEQ
    out ! TPAppendFile; handle
:

PROC CloseFile(CHAN OF ANY in, out, VAL BYTE handle)
  SEQ
    out ! TPCloseFile; handle
:

PROC RenameFile(CHAN OF ANY in, out, VAL BYTE handle, VAL []BYTE str)
  #USE "SendStr"
  SEQ
    out ! TPRenameFile; handle           -- Rename File Function
    SendString(in, out, str)   -- Write New Name
:

PROC DeleteFile(CHAN OF ANY in, out, VAL BYTE handle)
  SEQ
    out ! TPDeleteFile; handle        -- Erase File Function
:

PROC EndOfFile(CHAN OF ANY in, out, VAL BYTE handle, BOOL eof)
  BYTE temp:
  SEQ
    out ! TPEndOfFile; handle         -- End Of File Function
    in ? temp                         -- Get what it is

    IF
      (temp = 0 (BYTE))
        eof := FALSE
      TRUE
        eof := TRUE
:

PROC FileError(CHAN OF ANY in, out, INT error.code)
  BYTE temp:
  SEQ
    out ! TPFileError
    in ? temp
    error.code := INT temp
:
```

```
PROC ReadTextLine(CHAN OF ANY in, out, VAL BYTE handle, INT len, []BYTE str)
  BYTE length:
  SEQ
    length := BYTE (SIZE str)
    out ! TPReadTextLine; handle; length

    in ? length                         -- Read the length
    len := INT length
    IF
      (len > 0)
        in ? [str FROM 0 FOR len]       -- Read the string
      TRUE
        SKIP

    IF
      len < (SIZE str)
        str[len] := 0 (BYTE)                    -- Terminate String
      TRUE
        SKIP
:

PROC ReadTextNum(CHAN OF ANY in, out, VAL BYTE handle, INT len, []BYTE str)
  BYTE length:
  SEQ
    length := BYTE (SIZE str)
    out ! TPReadTextNum; handle; length

    in ? length                         -- Read the length
    len := INT length
    IF
      (len > 0)
        in ? [str FROM 0 FOR len]       -- Read the string
      TRUE
        SKIP

    IF
      len < (SIZE str)
        str[len] := 0 (BYTE)                    -- Terminate String
      TRUE
        SKIP
:

PROC ReadBlock(CHAN OF ANY in, out, VAL BYTE handle, VAL INT blocksize,
               []BYTE block, INT numread)
  INT16 temp1:
  VAL [4]BYTE sblocksize RETYPES blocksize:
  SEQ
    out ! TPReadBlock; handle; [sblocksize FROM 0 FOR 2] -- Size of the block

    in ? temp1                          -- Actual number read
    numread := INT temp1
    IF
      (numread = 0)
        SKIP
      TRUE
        in ? [block FROM 0 FOR numread]     -- Read The Block
:

PROC ReadBool(CHAN OF ANY in, out, VAL BYTE handle, BOOL val)
  BYTE temp:
  SEQ
    out ! TPReadBool; handle
    in ? temp                           -- Byte Read
    IF
      (temp = 0 (BYTE))
        val := FALSE
      TRUE
        val := TRUE
:
```

```
PROC ReadByte(CHAN OF ANY in, out, VAL BYTE handle, BYTE val)
  SEQ
    out ! TPReadByte; handle
    in ? val                    -- Byte Read
:

PROC ReadInt(CHAN OF ANY in, out, VAL BYTE handle, INT val)
  SEQ
    out ! TPReadInt; handle
    in ? val                    -- Integer Read
:

PROC ReadInt16(CHAN OF ANY in, out, VAL BYTE handle, INT16 val)
  SEQ
    out ! TPReadInt16; handle
    in ? val                        -- Integer Read
:

PROC ReadInt32(CHAN OF ANY in, out, VAL BYTE handle, INT32 val)
  SEQ
    out ! TPReadInt32; handle
    in ? val                          -- Integer Read
:

PROC ReadInt64(CHAN OF ANY in, out, VAL BYTE handle, INT64 val)
  SEQ
    out ! TPReadInt64; handle
    in ? val                       -- Integer Read
:

PROC ReadReal32(CHAN OF ANY in, out, VAL BYTE handle, REAL32 val)
  SEQ
    out ! TPReadReal32; handle
    in ? val                    -- Real Value Read
:

PROC ReadReal64(CHAN OF ANY in, out, VAL BYTE handle, REAL64 val)
  SEQ
    out ! TPReadReal64; handle
    in ? val                   -- Real Value Read
:

PROC WriteTextLine(CHAN OF ANY in, out, VAL BYTE handle, VAL []BYTE str)
  #USE "SendStr"
  SEQ
    out ! TPWriteTextLine; handle
    SendString(in, out, str)
:

PROC WriteText(CHAN OF ANY in, out, VAL BYTE handle, VAL []BYTE str)
  #USE "SendStr"
  SEQ
    out ! TPWriteText; handle
    SendString(in, out, str)
:

PROC WriteBlock(CHAN OF ANY in, out, VAL BYTE handle, VAL INT blocksize,
                VAL []BYTE block, INT numwritten)
  INT16 temp1:
  SEQ
    VAL [4]BYTE sblocksize RETYPES blocksize:
    out ! TPWriteBlock; handle; [sblocksize FROM 0 FOR 2]   -- Size of block

    IF
      (blocksize = 0)
        SKIP
      TRUE
        out ! [block FROM 0 FOR blocksize]   -- Write The Block

    in ? temp1                               -- Actual number written
    numwritten := INT temp1
```

```
:

PROC WriteBool(CHAN OF ANY in, out, VAL BYTE handle, VAL BOOL val)
  SEQ
    out ! TPWriteBool; handle; val      -- Byte Write
:

PROC WriteByte(CHAN OF ANY in, out, VAL BYTE handle,val)
  SEQ
    out ! TPWriteByte; handle; val          -- Write Byte
:

PROC WriteInt(CHAN OF ANY in, out, VAL BYTE handle, VAL INT val)
  SEQ
    out ! TPWriteInt; handle; val         -- Integer Write
:

PROC WriteInt16(CHAN OF ANY in, out, VAL BYTE handle, VAL INT16 val)
  SEQ
    out ! TPWriteInt16; handle; val         -- Integer Write
:

PROC WriteInt32(CHAN OF ANY in, out, VAL BYTE handle, VAL INT32 val)
  SEQ
    out ! TPWriteInt32; handle; val         -- Integer Write
:

PROC WriteInt64(CHAN OF ANY in, out, VAL BYTE handle, VAL INT64 val)
  SEQ
    out ! TPWriteInt64; handle; val         -- Integer Write
:

PROC WriteReal32(CHAN OF ANY in, out, VAL BYTE handle, VAL REAL32 val)
  SEQ
    out ! TPWriteReal32; handle; val         -- Real Value Write
:

PROC WriteReal64(CHAN OF ANY in, out, VAL BYTE handle, VAL REAL64 val)
  SEQ
    out ! TPWriteReal64; handle; val         -- Real Value Write
:
```

"TPDOSPRO.INC"

```
-- DOS PROTOCOL      151 - 200
VAL TPExec               IS BYTE 151:
VAL TPGetDate            IS BYTE 152:
VAL TPSetDate            IS BYTE 153:
VAL TPGetTime            IS BYTE 154:
VAL TPSetTime            IS BYTE 155:
VAL TPDiskFree           IS BYTE 156:
VAL TPDiskSize           IS BYTE 157:
VAL TPGetFAttr           IS BYTE 158:
VAL TPSetFAttr           IS BYTE 159:
VAL TPFileExists         IS BYTE 160:
VAL TPInterrupt          IS BYTE 161:
VAL TPReadPort           IS BYTE 162:
VAL TPWritePort          IS BYTE 163:
VAL TPFileSize           IS BYTE 164:
```

"TPDOS.OCC"

```
#INCLUDE "TPDosPro.INC"

PROC Exec(CHAN OF ANY in, out, VAL []BYTE path, cmd)
  #USE "SendStr"
  SEQ
    out ! TPExec                          -- Execute command
    SendString(in, out, path)   -- Send Path
    SendString(in, out, cmd)    -- Send parameters
:
```

```
PROC GetDate(CHAN OF ANY in, out, INT year, month, day, dayofweek)
  INT16 temp1, temp2, temp3, temp4:
  SEQ
    out ! TPGetDate                        -- Get Date command
    in ? temp1; temp2; temp3; temp4
    year, month, day, dayofweek := INT temp1, INT temp2, INT temp3, INT temp4
:

PROC SetDate(CHAN OF ANY in, out, VAL INT year, month, day)
  VAL [4]BYTE sy RETYPES year:
  VAL [4]BYTE sm RETYPES month:
  VAL [4]BYTE sd RETYPES day:
  SEQ
    -- Set Date command
    out ! TPSetDate; [sy FROM 0 FOR 2]; [sm FROM 0 FOR 2]; [sd FROM 0 FOR 2]
:

PROC GetTime(CHAN OF ANY in, out, INT hour, minute, second, sec100)
  INT16 temp1, temp2, temp3, temp4:
  SEQ
    out ! TPGetTime                        -- Get Time command
    in ? temp1; temp2; temp3; temp4
    hour, minute, second, sec100 := INT temp1, INT temp2, INT temp3, INT temp4
:

PROC SetTime(CHAN OF ANY in, out, VAL INT hour, minute, second, sec100)
  VAL [4]BYTE sh RETYPES hour:
  VAL [4]BYTE sm RETYPES minute:
  VAL [4]BYTE ss RETYPES second:
  VAL [4]BYTE s1 RETYPES sec100:
  SEQ
    -- Set Time Command
    out ! TPSetTime; [sh FROM 0 FOR 2]; [sm FROM 0 FOR 2];
                     [ss FROM 0 FOR 2]; [s1 FROM 0 FOR 2]
:

PROC DiskFree(CHAN OF ANY in, out, VAL INT disk , INT free)
  VAL [4]BYTE sdisk RETYPES disk:
  SEQ
    out ! TPDiskFree; [sdisk FROM 0 FOR 2]
    in ? free
:

PROC DiskSize(CHAN OF ANY in, out, VAL INT disk, INT size)
  VAL [4]BYTE sdisk RETYPES disk:
  SEQ
    out ! TPDiskSize; [sdisk FROM 0 FOR 2]
    in ? size
:

PROC FileSize(CHAN OF ANY in, out, VAL BYTE handle, INT size)
  SEQ
    out ! TPFileSize; handle           -- File Size function
    in ? size
:

PROC GetFAttr(CHAN OF ANY in, out, VAL BYTE handle, INT attr)
  BYTE temp1:
  SEQ
    out ! TPGetFAttr; handle
    in ? temp1      -- Get File Attribute and Error Code
    attr := INT temp1
:

PROC SetFAttr(CHAN OF ANY in, out, VAL BYTE handle, VAL INT attr)
  VAL [4]BYTE sattr RETYPES attr:
  SEQ
    out ! TPSetFAttr; handle; sattr[0]
:
```

```
PROC FileExists(CHAN OF ANY in,out, VAL []BYTE name, BOOL Does)
  #USE "SendStr"
  BYTE byte:
  SEQ
    out ! TPFileExists
    SendString(in,out,name)
    in ? byte
    Does := (byte <> 0 (BYTE))
:

PROC Intr(CHAN OF ANY in,out, VAL BYTE IntNo, INT16 AX, BX, CX, DX, BP,
                                                  SI, DI, DS, ES, Flags)
  SEQ
    out ! TPInterrupt; IntNo; AX; BX; CX; DX; BP; SI; DI; DS; ES; Flags
    in ? AX; BX; CX; DX; BP; SI; DI; DS; ES; Flags
:

PROC ReadPort(CHAN OF ANY in,out, VAL INT Port, BYTE val)
  VAL [4]BYTE sp RETYPES Port:
  SEQ
    out ! TPReadPort; [sp FROM 0 FOR 2]
    in ? val
:

PROC WritePort(CHAN OF ANY in,out, VAL INT Port, VAL BYTE val)
  VAL [4]BYTE sp RETYPES Port:
  SEQ
    out ! TPWritePort; [sp FROM 0 FOR 2]; val
:
```

"TPGRFPRO.INC"

```
-- GRAPH PROTOCOL    201 - 250
VAL TPInitGraph           IS BYTE 201:
VAL TPSetGraphMode        IS BYTE 202:
VAL TPRestoreCrtMode      IS BYTE 203:
VAL TPCloseGraph          IS BYTE 204:
VAL TPSetViewPort         IS BYTE 205:
VAL TPClearViewPort       IS BYTE 206:
VAL TPClearDevice         IS BYTE 207:
VAL TPDetectGraph         IS BYTE 208:
VAL TPSetLineStyle        IS BYTE 209:
VAL TPSetFillPattern      IS BYTE 210:
VAL TPSetFillStyle        IS BYTE 211:
VAL TPArc                 IS BYTE 212:
VAL TPBar                 IS BYTE 213:
VAL TPBar3D               IS BYTE 214:
VAL TPCircle              IS BYTE 215:
VAL TPDrawPoly            IS BYTE 216:
VAL TPLine                IS BYTE 217:
VAL TPLineTo              IS BYTE 218:
VAL TPMoveRel             IS BYTE 219:
VAL TPMoveTo              IS BYTE 220:
VAL TPPutPixel            IS BYTE 221:
VAL TPOutText             IS BYTE 222:
VAL TPOutTextXY           IS BYTE 223:
VAL TPRectangle           IS BYTE 224:
VAL TPSetTextJustify      IS BYTE 225:
VAL TPSetTextStyle        IS BYTE 226:
VAL TPGetAspectRatio      IS BYTE 227:
VAL TPGetMaxX             IS BYTE 228:
VAL TPGetMaxY             IS BYTE 229:
VAL TPGraphResult         IS BYTE 230:
VAL TPSetColor            IS BYTE 231:
VAL TPSetBkColor          IS BYTE 232:
VAL TPGetColor            IS BYTE 233:
VAL TPGetBkColor          IS BYTE 234:
VAL TPEllipse             IS BYTE 240:
VAL TPFillEllipse         IS BYTE 241:
VAL TPFillPoly            IS BYTE 242:
```

```
VAL TPFloodFill           IS BYTE 243:
VAL TPGetArcCoords        IS BYTE 244:
VAL TPLineRel             IS BYTE 245:
VAL TPPieSlice            IS BYTE 246:
VAL TPSector              IS BYTE 247:
```

"TPGRAPH.OCC"

```
#INCLUDE "TPGrfPro.INC"

PROC InitGraph(CHAN OF ANY in, out, INT GDriver, GMode, VAL []BYTE path)
  #USE "SendStr"
  INT16 temp1, temp2:
  SEQ
    --                      Drivers      and     Mode
    [4]BYTE sd RETYPES GDriver:
    [4]BYTE sm RETYPES GMode:
    out ! TPInitGraph; [sd FROM 0 FOR 2]; [sm FROM 0 FOR 2]
    SendString(in, out, path)   -- Send Path To Find Drivers
    in ? temp1; temp2
    GDriver, GMode := INT temp1, INT temp2             -- Return Results
:

PROC SetGraphMode(CHAN OF ANY in, out, VAL INT GMode)
  VAL [4]BYTE s RETYPES GMode:
  SEQ
    out ! TPSetGraphMode; [s FROM 0 FOR 2]
:

PROC RestoreCrtMode (CHAN OF ANY in, out)
  SEQ
    out ! TPRestoreCrtMode
:

PROC CloseGraph (CHAN OF ANY in, out)
  SEQ
    out ! TPCloseGraph
:

PROC SetViewPort(CHAN OF ANY in, out, VAL INT X1, Y1, X2, Y2, VAL BOOL Clip)
  VAL [4]BYTE sx1 RETYPES X1:
  VAL [4]BYTE sy1 RETYPES Y1:
  VAL [4]BYTE sx2 RETYPES X2:
  VAL [4]BYTE sy2 RETYPES Y2:
  SEQ
    out ! TPSetViewPort; [sx1 FROM 0 FOR 2]; [sy1 FROM 0 FOR 2];
                         [sx2 FROM 0 FOR 2]; [sy2 FROM 0 FOR 2]; Clip
:

PROC ClearViewPort (CHAN OF ANY in, out)
  SEQ
    out ! TPClearViewPort
:

PROC ClearDevice (CHAN OF ANY in, out)
  SEQ
    out ! TPClearDevice
:

PROC DetectGraph(CHAN OF ANY in, out, INT GDriver, GMode)
  INT16 temp1, temp2:
  SEQ
    out ! TPDetectGraph
    in ? temp1; temp2
    GDriver, GMode := INT temp1, INT temp2      -- Return Results
:
```

```
PROC SetLineStyle(CHAN OF ANY in, out, VAL INT Style, Pattern, Thickness)
  VAL [4]BYTE ss RETYPES Style:
  VAL [4]BYTE sp RETYPES Pattern:
  VAL [4]BYTE st RETYPES Thickness:
  SEQ
    out ! TPSetLineStyle; [ss FROM 0 FOR 2]; [sp FROM 0 FOR 2];
                          [st FROM 0 FOR 2]
:

PROC SetFillPattern(CHAN OF ANY in, out, VAL [8]BYTE Pattern, VAL INT Color)
  VAL [4]BYTE sc RETYPES Color:
  INT i:
  SEQ
    out ! TPSetFillPattern; Pattern; [sc FROM 0 FOR 2]
:

PROC SetFillStyle(CHAN OF ANY in, out, VAL INT Pattern, Color)
  VAL [4]BYTE sp RETYPES Pattern:
  VAL [4]BYTE sc RETYPES Color:
  SEQ
    out ! TPSetFillStyle; [sp FROM 0 FOR 2]; [sc FROM 0 FOR 2]
:

PROC Arc(CHAN OF ANY in, out, VAL INT X, Y, StAngle, EndAngle, Radius)
  VAL [4]BYTE sx RETYPES X:
  VAL [4]BYTE sy RETYPES Y:
  VAL [4]BYTE ss RETYPES StAngle:
  VAL [4]BYTE se RETYPES EndAngle:
  VAL [4]BYTE sr RETYPES Radius:
  SEQ
    out ! TPArc; [sx FROM 0 FOR 2]; [sy FROM 0 FOR 2];
                 [ss FROM 0 FOR 2]; [se FROM 0 FOR 2]; [sr FROM 0 FOR 2]
:

PROC Bar(CHAN OF ANY in, out, VAL INT X1, Y1, X2, Y2)
  VAL [4]BYTE sX1 RETYPES X1:
  VAL [4]BYTE sY1 RETYPES Y1:
  VAL [4]BYTE sX2 RETYPES X2:
  VAL [4]BYTE sY2 RETYPES Y2:
  SEQ
    out ! TPBar; [sX1 FROM 0 FOR 2]; [sY1 FROM 0 FOR 2];
                 [sX2 FROM 0 FOR 2]; [sY2 FROM 0 FOR 2]
:

PROC Bar3D(CHAN OF ANY in, out, VAL INT X1, Y1, X2, Y2, Depth, VAL BOOL Top)
  VAL [4]BYTE sX1 RETYPES X1:
  VAL [4]BYTE sY1 RETYPES Y1:
  VAL [4]BYTE sX2 RETYPES X2:
  VAL [4]BYTE sY2 RETYPES Y2:
  VAL [4]BYTE sDepth RETYPES Depth:
  SEQ
    out ! TPBar3D; [sX1 FROM 0 FOR 2]; [sY1 FROM 0 FOR 2];
                   [sX2 FROM 0 FOR 2]; [sY2 FROM 0 FOR 2];
                   [sDepth FROM 0 FOR 2]; Top
:

PROC Circle(CHAN OF ANY in, out, VAL INT X, Y, Radius)
  VAL [4]BYTE sX RETYPES X:
  VAL [4]BYTE sY RETYPES Y:
  VAL [4]BYTE sRadius RETYPES Radius:
  SEQ
    out ! TPCircle; [sX FROM 0 FOR 2]; [sY FROM 0 FOR 2];
                    [sRadius FROM 0 FOR 2]
:
```

```
PROC Ellipse(CHAN OF ANY in, out, VAL INT X, Y, StAngle, EndAngle,
                                        XRadius, YRadius)
  VAL [4]BYTE sX        RETYPES X:
  VAL [4]BYTE sY        RETYPES Y:
  VAL [4]BYTE sStAngle  RETYPES StAngle:
  VAL [4]BYTE sEndAngle RETYPES EndAngle:
  VAL [4]BYTE sXRadius  RETYPES XRadius:
  VAL [4]BYTE sYRadius  RETYPES YRadius:
  SEQ
    out ! TPEllipse; [sX FROM 0 FOR 2]; [sY FROM 0 FOR 2];
                     [sStAngle FROM 0 FOR 2]; [sEndAngle FROM 0 FOR 2];
                     [sXRadius FROM 0 FOR 2]; [sYRadius FROM 0 FOR 2]
:

PROC FillEllipse(CHAN OF ANY in, out, VAL INT X, Y, XRadius, YRadius)
  VAL [4]BYTE sX       RETYPES X:
  VAL [4]BYTE sY       RETYPES Y:
  VAL [4]BYTE sXRadius RETYPES XRadius:
  VAL [4]BYTE sYRadius RETYPES YRadius:
  SEQ
    out ! TPFillEllipse; [sX FROM 0 FOR 2]; [sY FROM 0 FOR 2];
                         [sXRadius FROM 0 FOR 2]; [sYRadius FROM 0 FOR 2]
:

PROC PieSlice(CHAN OF ANY in, out, VAL INT X, Y, StAngle, EndAngle, Radius)
  VAL [4]BYTE sX        RETYPES X:
  VAL [4]BYTE sY        RETYPES Y:
  VAL [4]BYTE sStAngle  RETYPES StAngle:
  VAL [4]BYTE sEndAngle RETYPES EndAngle:
  VAL [4]BYTE sRadius   RETYPES Radius:
  SEQ
    out ! TPPieSlice; [sX FROM 0 FOR 2]; [sY FROM 0 FOR 2];
                      [sStAngle FROM 0 FOR 2]; [sEndAngle FROM 0 FOR 2];
                      [sRadius FROM 0 FOR 2]
:

PROC DrawPoly(CHAN OF ANY in, out, VAL INT NumPoints, VAL [][2]INT points)
  INT i:
  VAL [4]BYTE sNumPoints RETYPES NumPoints:
  SEQ
    out ! TPDrawPoly; [sNumPoints FROM 0 FOR 2]
    IF
      NumPoints > 0
        SEQ i = 0 FOR NumPoints
          VAL [4]BYTE sp0 RETYPES points[i][0]:
          VAL [4]BYTE sp1 RETYPES points[i][1]:
          out ! [sp0 FROM 0 FOR 2]; [sp1 FROM 0 FOR 2]
      TRUE
        SKIP

:

PROC FillPoly(CHAN OF ANY in, out, VAL INT NumPoints, VAL [][2]INT points)
  INT i:
  VAL [4]BYTE sNumPoints RETYPES NumPoints:
  SEQ
    out ! TPFillPoly; [sNumPoints FROM 0 FOR 2]
    IF
      NumPoints > 0
        SEQ i = 0 FOR NumPoints
          VAL [4]BYTE sp0 RETYPES points[i][0]:
          VAL [4]BYTE sp1 RETYPES points[i][1]:
          out ! [sp0 FROM 0 FOR 2]; [sp1 FROM 0 FOR 2]
      TRUE
        SKIP

:
```

```
PROC Line(CHAN OF ANY in, out, VAL INT X1, Y1, X2, Y2)
  VAL [4]BYTE sX1 RETYPES X1:
  VAL [4]BYTE sY1 RETYPES Y1:
  VAL [4]BYTE sX2 RETYPES X2:
  VAL [4]BYTE sY2 RETYPES Y2:
  SEQ
    out ! TPLine; [sX1 FROM 0 FOR 2]; [sY1 FROM 0 FOR 2];
                  [sX2 FROM 0 FOR 2]; [sY2 FROM 0 FOR 2]
:

PROC LineTo(CHAN OF ANY in, out, VAL INT X, Y)
  VAL [4]BYTE sX RETYPES X:
  VAL [4]BYTE sY RETYPES Y:
  SEQ
    out ! TPLineTo; [sX FROM 0 FOR 2]; [sY FROM 0 FOR 2]
:

PROC LineRel(CHAN OF ANY in, out, VAL INT X, Y)
  VAL [4]BYTE sX RETYPES X:
  VAL [4]BYTE sY RETYPES Y:
  SEQ
    out ! TPLineRel; [sX FROM 0 FOR 2]; [sY FROM 0 FOR 2]
:

PROC Sector(CHAN OF ANY in, out, VAL INT X, Y, StAngle, EndAngle, XRad, YRad)
  VAL [4]BYTE sX RETYPES X:
  VAL [4]BYTE sY RETYPES Y:
  VAL [4]BYTE sStAngle RETYPES StAngle:
  VAL [4]BYTE sEndAngle RETYPES EndAngle:
  VAL [4]BYTE sXRad RETYPES XRad:
  VAL [4]BYTE sYRad RETYPES YRad:
  SEQ
    out ! TPSector; [sX FROM 0 FOR 2]; [sY FROM 0 FOR 2];
                    [sStAngle FROM 0 FOR 2]; [sEndAngle FROM 0 FOR 2];
                    [sXRad FROM 0 FOR 2]; [sYRad FROM 0 FOR 2]
:

PROC FloodFill(CHAN OF ANY in, out, VAL INT X, Y, Border)
  VAL [4]BYTE sX RETYPES X:
  VAL [4]BYTE sY RETYPES Y:
  VAL [4]BYTE sBorder RETYPES Border:
  SEQ
    out ! TPFloodFill; [sX FROM 0 FOR 2]; [sY FROM 0 FOR 2];
                       [sBorder FROM 0 FOR 2]
:

PROC GetArcCoords(CHAN OF ANY in, out, INT X, Y, StX, StY, EndX, EndY)
  INT16 temp1, temp2, temp3, temp4, temp5, temp6:
  SEQ
    out ! TPGetArcCoords
    in ? temp1; temp2; temp3; temp4; temp5; temp6
    X, Y := INT temp1, INT temp2           -- Centre Point
    StX, StY, EndX, EndY := INT temp3, INT temp4, INT temp5, INT temp6
:

PROC MoveRel(CHAN OF ANY in, out, VAL INT X, Y)
  VAL [4]BYTE sX RETYPES X:
  VAL [4]BYTE sY RETYPES Y:
  SEQ
    out ! TPMoveRel; [sX FROM 0 FOR 2]; [sY FROM 0 FOR 2]
:

PROC MoveTo(CHAN OF ANY in, out, VAL INT X, Y)
  VAL [4]BYTE sX RETYPES X:
  VAL [4]BYTE sY RETYPES Y:
  SEQ
    out ! TPMoveTo; [sX FROM 0 FOR 2]; [sY FROM 0 FOR 2]
:
```

```
PROC PutPixel(CHAN OF ANY in, out, VAL INT X, Y, Pixel)
  VAL [4]BYTE sX RETYPES X:
  VAL [4]BYTE sY RETYPES Y:
  VAL [4]BYTE sPixel RETYPES Pixel:
  SEQ
    out ! TPPutPixel; [sX FROM 0 FOR 2]; [sY FROM 0 FOR 2];
                      [sPixel FROM 0 FOR 2]
:

PROC OutText(CHAN OF ANY in, out, VAL []BYTE str)
  #USE "SendStr"
  SEQ
    out ! TPOutText
    SendString(in, out, str)
:

PROC OutTextXY(CHAN OF ANY in, out, VAL INT X, Y,  VAL []BYTE str)
  #USE "SendStr"
  VAL [4]BYTE sX RETYPES X:
  VAL [4]BYTE sY RETYPES Y:
  SEQ
    out ! TPOutTextXY; [sX FROM 0 FOR 2]; [sY FROM 0 FOR 2]
    SendString(in, out, str)
:

PROC Rectangle(CHAN OF ANY in, out, VAL INT X1, Y1, X2, Y2)
  VAL [4]BYTE sX1 RETYPES X1:
  VAL [4]BYTE sY1 RETYPES Y1:
  VAL [4]BYTE sX2 RETYPES X2:
  VAL [4]BYTE sY2 RETYPES Y2:
  SEQ
    out ! TPRectangle; [sX1 FROM 0 FOR 2]; [sY1 FROM 0 FOR 2];
                       [sX2 FROM 0 FOR 2]; [sY2 FROM 0 FOR 2]
:

PROC SetTextJustify(CHAN OF ANY in, out, VAL INT Horiz, Vert)
  VAL [4]BYTE sHoriz RETYPES Horiz:
  VAL [4]BYTE sVert RETYPES Vert:
  SEQ
    out ! TPSetTextJustify; [sHoriz FROM 0 FOR 2]; [sVert FROM 0 FOR 2]
:

PROC SetTextStyle(CHAN OF ANY in, out, VAL INT Font, Direction,  VAL BYTE Size)
  VAL [4]BYTE sFont      RETYPES Font:
  VAL [4]BYTE sDirection RETYPES Direction:
  SEQ
    out ! TPSetTextStyle; [sFont FROM 0 FOR 2]; [sDirection FROM 0 FOR 2]; Size
:

PROC GetAspectRatio(CHAN OF ANY in, out, INT AspX, AspY)
  INT16 temp1, temp2:
  SEQ
    out ! TPGetAspectRatio
    in ? temp1; temp2
    AspX, AspY := INT temp1, INT temp2          -- Return Results
:

PROC GetMaxX(CHAN OF ANY in, out, INT MaxX)
  INT16 temp1:
  SEQ
    out ! TPGetMaxX
    in ? temp1
    MaxX := INT temp1                -- Return Result

:
```

```
PROC GetMaxY(CHAN OF ANY in, out, INT MaxY)
  INT16 temp1:
  SEQ
    out ! TPGetMaxY
    in ? temp1
    MaxY := INT temp1                -- Return Result
:

PROC GraphResult(CHAN OF ANY in, out, INT Result)
  INT16 temp1:
  SEQ
    out ! TPGraphResult
    in ? temp1
    Result := INT temp1                -- Return Result
:

PROC SetColor(CHAN OF ANY in, out, VAL INT Color)
  VAL [4]BYTE sColor RETYPES Color:
  SEQ
    out ! TPSetColor; [sColor FROM 0 FOR 2]
:

PROC SetBkColor(CHAN OF ANY in, out, VAL INT BkColor)
  VAL [4]BYTE sBkColor RETYPES BkColor:
  SEQ
    out ! TPSetBkColor; [sBkColor FROM 0 FOR 2]
:

PROC GetColor(CHAN OF ANY in, out, INT Color)
  INT16 temp1:
  SEQ
    out ! TPGetColor
    in  ? temp1
    Color := INT temp1
:

PROC GetBkColor(CHAN OF ANY in, out, INT BkColor)
  INT16 temp1:
  SEQ
    out ! TPGetBkColor
    in  ? temp1
    BkColor := INT temp1
:
```

"TPSERPRO.INC"

```
-- SERVER PROTOCOL
VAL TPSystemInfo          IS BYTE 250:
VAL TPBootSubsystem       IS BYTE 251:
VAL TPBootFileSubSys      IS BYTE 252:
VAL TPEndServer           IS BYTE 255:
```

"TPSERVER.OCC"

```
#INCLUDE "TPSerPro.INC"

PROC SystemInfo(CHAN OF ANY in,out, INT memsize)
  SEQ
    out ! TPSystemInfo
    in  ? memsize
:

PROC TerminateServer (CHAN OF ANY in, out)
  SEQ
    out ! TPEndServer          -- End Server Function
:
```

"TPSUBSYS.OCC"

```
PROC reset.subsystem (BOOL err)
  VAL sub.reset   IS ((MOSTNEG INT) >< 0) >> 2:
  VAL sub.error   IS ((MOSTNEG INT) >< 0) >> 2:
  VAL sub.analyse IS ((MOSTNEG INT) >< 4) >> 2:
  PORT OF INT reset, error, analyse:
  PLACE reset   AT sub.reset:
  PLACE error   AT sub.error:
  PLACE analyse AT sub.analyse:

  TIMER clock:
  INT   current, errval:
  SEQ
    analyse ! 0               -- analyse low
    reset   ! 0               -- reset low
    clock ? current
    clock ? AFTER current PLUS 78
    reset   ! 1               -- reset high
    clock ? current
    clock ? AFTER current PLUS 78
    reset   ! 0               -- reset subsystem
    clock ? current
    clock ? AFTER current PLUS 78

    error   ? errval          -- test if error in resetting subsystem
    IF
      errval = 0              -- error is NOTted
        err := TRUE
      TRUE
        err := FALSE
:
PROC poke(CHAN OF ANY to.subsystem, from.subsystem, VAL INT address,value)
  VAL poke.val IS 0 (BYTE):
  SEQ
    to.subsystem ! poke.val          -- request POKE
    to.subsystem ! address           -- send address
    to.subsystem ! value             -- send value to POKE
:
PROC peek(CHAN OF ANY to.subsystem, from.subsystem, VAL INT address, INT16 value)
  VAL peek.val IS 1 (BYTE):
  SEQ
    to.subsystem   ! peek.val          -- request PEEK
    to.subsystem   ! address           -- send address
    from.subsystem ? value             -- receive value PEEKed
:
```

"QUADPUTE.OCC"

```
#INCLUDE "TPSerPro.INC"
#USE "TPSubSys"
#USE "TPScreen"
#USE "SendStr"

PROC boot.quadputer(CHAN OF ANY in, out, bootlink, BOOL error)
  INT size, I:
  BYTE byte:
  BOOL result:

  SEQ
    out ! TPBootSubsystem
    in ? size
    IF
      size > 0
        SEQ
          reset.subsystem(result)
          IF
            result
              SEQ
                SEQ I = 0 FOR size
                  in ? byte
                error := TRUE
```

```
          TRUE
            SEQ
              SEQ I = 0 FOR size
                SEQ
                  in ? byte
                  bootlink ! byte
              error := FALSE
      size = 0                          -- ASSUME QUADPUTER ALREADY RUNNING
        error := FALSE
      TRUE                              -- BOOT FILE NOT FOUND
        error := TRUE
:
PROC boot.file.quadputer(CHAN OF ANY in, out, bootlink, VAL []BYTE name, BOOL error)
  INT size, I:
  BYTE byte:
  BOOL result:

  SEQ
    out ! TPBootFileSubSys
    SendString(in,out,name)
    in ? size
    IF
      size > 0
        SEQ
          reset.subsystem(result)
          IF
            result
              SEQ
                SEQ I = 0 FOR size
                  in ? byte
                error := TRUE
            TRUE
              SEQ
                SEQ I = 0 FOR size
                  SEQ
                    in ? byte
                    bootlink ! byte
                error := FALSE
      size = 0                          -- ASSUME QUADPUTER ALREADY RUNNING
        error := FALSE
      TRUE                              -- BOOT FILE NOT FOUND
        error := TRUE
:
```

Chapter 5 - Airflow Model

Airflow Model - OCCAM Source Code

This chapter contains the source listing of the OCCAM 2 source code of the final implementation of the Airflow Modelling System described in Section 9 of the thesis. It must be noted that the algorithm or techniques used in implementing the theory of Airflow Modelling is not an issue, but rather the technique of implementing the code in parallel. The Airflow Shell therefore consists of the protocols and communication routines used, as well as the method of distributing the work and parallel divisions employed. The worker code listed has been translated as close as possible to the actual Fortran 77 source code, which was provided by the physics consultants, as well as the initial calculations of the system by the master used to set up the model. The actual Fortran 77 code provided by the consultants incorporated in no way any for of parallelism. The consultants merely concentrated on the airflow problem at hand, leaving the parallel implementations entirely to myself. No modifications to the algorithm theory were made, apart from the obvious change of parallelism and the implementations discussed in Sections 9.5 and 9.6 of the thesis.

```
{{{  COMMENT on the format of work distribution
The T414 is the controller and the 4 T800s are the workers W, X, Y and Z.  The procedure START (which
initiates a molecule within a cell) is integrated in the controller with the procedure DOCELL being passed
the cell I and J.  DOCELL then determines the adjoining cell which has been calculated (if any) and passes
across those parameters to the worker.  This adjoining calculated cell is necessary in order to convey
information to the centre of the model.  The adjoining cell closest to the walls should take preference in
order to convey the information as fast as possible.

NOTE: 1 Feb 1989
The model given is operative and fully functional.
}}}  COMMENT on the format of work distribution

{{{  Libraries
{{{  LIB "AIRPRO"    Airflow protocols
{{{  Protocols
PROTOCOL messages
  CASE
    initialise;    INT; INT; INT; REAL32; REAL32; REAL32; REAL32
    initialisepro; INT; INT; INT; INT; REAL32; REAL32; REAL32; REAL32
    point;         INT16; INT16
    results1;      INT; INT; INT
    results2;      [15][15]REAL32; [15][15]REAL32;
                   REAL32; REAL32; REAL32; REAL32;
                   REAL32; REAL32; REAL32; REAL32; REAL32
    work;          INT; INT; [15][15]REAL32; [15][15]REAL32;
                   REAL32; REAL32; REAL32; REAL32; REAL32; REAL32; REAL32
```

```
    toprocessor;   INT; INT; INT; [15][15]REAL32; [15][15]REAL32;
                   REAL32; REAL32; REAL32; REAL32; REAL32; REAL32; REAL32
    lotsofwork;    INT
:
}}} Protocols

{{{ Constants
VAL PI      IS  3.141592654 (REAL32):
VAL ZERO    IS  0.0 (REAL32):
VAL WorkerW IS INT 1:
VAL WorkerX IS INT 2:
VAL WorkerY IS INT 3:
VAL WorkerZ IS INT 4:
}}} Constants
}}} LIB "AIRPRO" Airflow protocols


{{{ LIB "AIRLIB1" -- "Airflow library 1"
{{{F RAND FUNCTION (Returns a random number between 0 and 1)
REAL32, INT FUNCTION RAND(VAL INT SEED)
  #USE snglmath
  INT  newseed:
  REAL32 rand:
  VALOF
    SEQ
      newseed := SEED TIMES 259
      IF
        newseed < 0
          newseed := (newseed PLUS 2147483647) PLUS 1
        newseed = 0
          newseed := 2147483647
        TRUE
          SKIP
      rand := SCALEB(REAL32 ROUND newseed, -31)
    RESULT rand, newseed
:
}}}F RAND FUNCTION (Returns a random number between 0 and 1)
}}} LIB "AIRLIB1" -- "Airflow library 1"

{{{ LIB "AIRLIB2" -- "Airflow library 2"
{{{ Constants
VAL PI     IS  3.141592654 (REAL32):
VAL ZERO   IS  0.0 (REAL32):
}}}

{{{F RSIGN FUNCTION (Returns a random plus or minus one)
REAL32, INT FUNCTION RSIGN(VAL INT SEED)
  #USE Airlib1
  INT  newseed:
  REAL32 ran, result:
  VALOF
    SEQ
      ran, newseed := RAND(SEED)
      IF
        ran > 0.5 (REAL32)
          result := 1.0 (REAL32)
        TRUE
          result := -1.0 (REAL32)
    RESULT result, newseed
:
}}}F RSIGN FUNCTION (Returns a random plus or minus one)

{{{F MIN FUNCTION (Returns the minimum of the two real values)
REAL32 FUNCTION MIN(VAL REAL32 V1,V2)
  REAL32 result:
  VALOF
    SEQ
      IF
        V1 < V2
          result := V1
```

```
        TRUE
          result := V2
    RESULT result
:
}}}F MIN FUNCTION (Returns the minimum of the two real values)

{{{F MAX FUNCTION (Returns the maximum of the two real values)
REAL32 FUNCTION MAX(VAL REAL32 V1,V2)
  REAL32 result:
  VALOF
    SEQ
      IF
        V1 > V2
          result := V1
        TRUE
          result := V2
    RESULT result
:
}}}F MAX FUNCTION (Returns the maximum of the two real values)
}}}  LIB "AIRLIB2" -- "Airflow library 2"

{{{  LIB "WORKER"
{{{F Worker Procedure
#USE AirPro
PROC Worker(VAL INT ProcID, CHAN OF messages fromController, toController)

  #USE snglmath
  #USE Airlib1
  #USE Airlib2


  INT MAXCOL, SEED, LC, I, J, ncolij:
  REAL32 xip, yip, uip, vip, mx0ij, my0ij, c0ij, mx0param, my0param, c0param,
         gij, gxfact, gyfact, H, TEMP:
  [15][15]REAL32 f0ij, f1ij:

  {{{F Start procedure (Starting a new molecule)
  PROC Start()
    #USE Airlib1
    #USE Airlib2
    #USE snglmath

    SEQ
      xip := H * ((REAL32 TRUNC I) + 0.5 (REAL32))
      yip := H * ((REAL32 TRUNC J) + 0.5 (REAL32))
      REAL32 rand, rsign:
      SEQ
        {{{  U := RSIGN*SQRT(-ALOG(RAND))*c0param+mx0param
        rand, SEED := RAND(SEED)
        rsign, SEED := RSIGN(SEED)
        uip := (rsign*(SQRT(-ALOG(rand))*c0param)) + mx0param
        }}}  U := RSIGN*SQRT(-ALOG(RAND))*c0param+mx0param

        {{{  V := RSIGN*SQRT(-ALOG(RAND))*c0param+my0param
        rand, SEED := RAND(SEED)
        rsign, SEED := RSIGN(SEED)
        vip := (rsign*(SQRT(-ALOG(rand))*c0param)) + my0param
        }}}  V := RSIGN*SQRT(-ALOG(RAND))*c0param+my0param
  :
  }}}F Start procedure (Starting a new molecule)
```

```
{{{F Smash procedure
PROC SMASH(INT I, J,  REAL32 US, VS,
           INT SEED,
           [15][15]REAL32 f0ij,
           REAL32 gij, mx0ij, my0ij)
  #USE Airlib1
  #USE Airlib2
  #USE snglmath

  REAL32 R, SUM:
  INT LC, L, M, N:

  VAL REAL32 G1 IS gij:
  VAL REAL32 DUDV IS G1*G1:
  BOOL loop:
  SEQ
    loop := TRUE
    WHILE loop
      SEQ
        REAL32 rand:
        SEQ
          rand, SEED := RAND(SEED)
          R := rand / DUDV
        LC, L, M := 7, 0, 0
        SUM := f0ij[7+L][7+M]
        IF
          SUM > R
            loop := FALSE
          TRUE
            {{{  50 loop
            SEQ
              N := 1
              WHILE (N <= LC) AND loop
                SEQ
                  {{{  10 loop
                  L, M := N, (-N)
                  VAL INT np7          IS 7+N:
                  VAL []REAL32 f0ijn   IS f0ij[np7]:
                  WHILE (M <= N) AND loop
                    SEQ
                      SUM := SUM + f0ijn[7+M]
                      IF
                        SUM > R
                          loop := FALSE
                        TRUE
                          M := M + 1
                  }}}  10 loop
                  IF
                    loop
                      {{{  20, 30, 40 loops
                      SEQ
                        {{{  20 loop
                        L, M := N-1,N
                        VAL INT onemn        IS 1-N:
                        VAL INT np7          IS 7+N:
                        WHILE (L >= onemn) AND loop
                          SEQ
                            SUM := SUM + f0ij[7+L][np7]
                            IF
                              SUM > R
                                loop := FALSE
                              TRUE
                                L := L - 1
                        }}}  20 loop
```

```
                    IF
                      loop
                        {{{  30 and 40 loops
                        SEQ
                          {{{  30 loop  .
                          SEQ
                            L, M := (-N), N
                            VAL []REAL32 fOijn IS fOij[7-N]:
                            WHILE (M >= (-N)) AND loop
                              SEQ
                                SUM := SUM + fOijn[7+M]
                                IF
                                  SUM > R
                                    loop := FALSE
                                  TRUE
                                    M := M - 1
                          }}}  30 loop
                          IF
                            loop
                              SEQ
                                {{{  40 loop
                                L, M := 1-N, (-N)
                                VAL INT sevenmn IS 7-N:
                                VAL INT nm1 IS N-1:
                                WHILE (L <= nm1) AND loop
                                  SEQ
                                    SUM := SUM + fOij[7+L][sevenmn]
                                    IF
                                      SUM > R
                                        loop := FALSE
                                      TRUE
                                        L := L + 1
                                }}}  40 loop
                                {{{  Next N ?
                                IF
                                  loop
                                    N := N + 1
                                  TRUE
                                    SKIP
                                }}}  Next N ?
                            TRUE
                              SKIP
                        }}}  30 and 40 loops
                      TRUE
                        SKIP
                  }}}  20, 30, 40 loops
                TRUE
                  SKIP
          }}}  50 loop

    REAL32 GC, UT, VT:
    SEQ
      UT := ((REAL32 TRUNC L)*G1) + mxOij
      VT := ((REAL32 TRUNC M)*G1) + myOij

      REAL32 rand:
      SEQ
        rand, SEED := RAND(SEED)
        R := 2.0 (REAL32) * (PI * rand)

      VAL REAL32 usmut IS US - UT:
      VAL REAL32 vsmvt IS VS - VT:
      SEQ
        GC := SQRT((usmut*usmut) + (vsmvt*vsmvt))

      US := 0.5 (REAL32) * ((US+UT) + (GC*COS(R)))
      VS := 0.5 (REAL32) * ((VS+VT) + (GC*SIN(R)))
:
}}}F Smash procedure
```

```
SEQ
  fromController ? CASE
    initialise; MAXCOL; SEED; LC; gxfact; gyfact; H; TEMP
      SKIP

  WHILE TRUE
    INT  counter:
    REAL32 CHISQ, C1, MX1, MY1, DUDV:
    SEQ
      -- Get Work
      fromController ? CASE
        work; I; J; f0ij; f1ij; mx0ij; my0ij; c0ij; gij; c0param; mx0param; my0param
          INT nummols:
          SEQ
            nummols := 1
            ncolij, counter := 0, 0
            Start()
            WHILE (ncolij < MAXCOL)
              {{{  Do the points
              INT II, JJ, L, M:
              REAL32 TC:
              REAL32 XP, YP:
              SEQ
                XP := (xip*25.0 (REAL32)) / H
                YP := (yip*25.0 (REAL32)) / H
                IF
                  counter = 10
                    SEQ
                      toController ! point; (INT16 ROUND (XP*gxfact)); (INT16 ROUND (YP*gyfact))
                      counter := 0
                  TRUE
                    counter := counter + 1

                L := INT ROUND ((uip-mx0ij)/gij)
                M := INT ROUND ((vip-my0ij)/gij)
                {{{  Catch Overflows
                IF
                  L < (-LC)
                    L := (-LC)
                  L > LC
                    L := LC
                  TRUE
                    SKIP
                IF
                  M < (-LC)
                    M := (-LC)
                  M > LC
                    M := LC
                  TRUE
                    SKIP
                }}}  Catch Overflows

                -- Time to collision
                REAL32 rand:
                SEQ
                  rand, SEED := RAND(SEED)
                  TC := (-ALOG(rand)) / c0ij

                -- Time to boundary of cell
                REAL32 XSTEP, YSTEP:
                SEQ
                  XSTEP := uip*TC
                  YSTEP := vip*TC
                  xip := xip + XSTEP
                  yip := yip + YSTEP
                  II    := INT TRUNC (xip/H)
                  JJ    := INT TRUNC (yip/H)
```

```
    -- Test for collisional or boundary crossing
    IF
      (II = I) AND (JJ = J)
        SEQ                 -- Collision
          f1ij[7+L][7+M] := f1ij[7+L][7+M] + TC
          ncolij := ncolij + 1
          SMASH(I,J,uip,vip,
                SEED, f0ij, gij, mx0ij, my0ij)
      TRUE
        Start()             -- Start new molecule as boundary crossed
  }}}  Do the points

{{{  Process the results
REAL32 NORM1, C00:
SEQ
  -- 200 Code
  NORM1, MX1, MY1, C1 := ZERO, ZERO, ZERO, ZERO
  DUDV := gij*gij
  SEQ L = (-LC) FOR ((LC+LC)+1)
    []REAL32 f1ijl IS f1ij[7+L]:
    SEQ M = (-LC) FOR ((LC+LC)+1)
      REAL32 f1ijlm IS f1ijl[7+M]:
      SEQ
        MX1 := MX1 + (((gij*(REAL32 TRUNC L)) + mx0ij) * f1ijlm)
        MY1 := MY1 + (((gij*(REAL32 TRUNC M)) + my0ij) * f1ijlm)
        NORM1 := NORM1 + f1ijlm

  -- This                     replaces   this
  MX1 := MX1 / NORM1                    -- NORM1 := NORM1 * DUDV
  MY1 := MY1 / NORM1                    -- MX1 := (MX1*DUDV) / NORM1
                                        -- MY1 := (MY1*DUDV) / NORM1

  SEQ L = (-LC) FOR ((LC+LC)+1)
    []REAL32 f1ijl IS f1ij[7+L]:
    SEQ M = (-LC) FOR ((LC+LC)+1)
      REAL32 f1ijlm IS f1ijl[7+M]:
      REAL32 tmp1, tmp2:
      SEQ
        tmp1 := gij * (REAL32 TRUNC L)
        tmp2 := gij * (REAL32 TRUNC M)
        C1 := C1 + (((tmp1*tmp1) + (tmp2*tmp2))*f1ijlm)
  C1 := SQRT(C1 / NORM1)
  NORM1 := NORM1 * DUDV             -- Final part of (This replaces this)

  c0ij := C1
  C00 := C1*C1
  mx0ij := MX1
  my0ij := MY1

  -- Initialise distribution F0
  gij := 0.3 (REAL32) * c0ij
  DUDV := gij*gij

  REAL32 TOT, DEN1:
  SEQ
    TOT := ZERO
    SEQ L = (-LC) FOR ((LC+LC)+1)
      []REAL32 f0ijl IS f0ij[7+L]:
      SEQ M = (-LC) FOR ((LC+LC)+1)
        REAL32 tmp1, tmp2:
        REAL32 f0ijlm IS f0ijl[7+M]:
        SEQ
          tmp1 := gij * (REAL32 TRUNC L)
          tmp2 := gij * (REAL32 TRUNC M)
          f0ijlm := EXP(-(((tmp1*tmp1) + (tmp2*tmp2))/C00))
          TOT := TOT + f0ijlm

    DEN1 := TOT*DUDV
```

```
            CHISQ := ZERO
            SEQ L = (-LC) FOR ((LC+LC)+1)
              []REAL32 fOijl IS fOij[7+L]:
              []REAL32 f1ijl IS f1ij[7+L]:
              SEQ M = (-LC) FOR ((LC+LC)+1)
                REAL32 fOijlm IS fOijl[7+M]:
                REAL32 f1ijlm IS f1ijl[7+M]:
                REAL32 tmp:
                SEQ
                  fOijlm := fOijlm / DEN1
                  f1ijlm := f1ijlm / NORM1
                  tmp := f1ijlm - fOijlm
                  CHISQ := CHISQ + (DUDV*((tmp*tmp)/fOijlm))
        }}} Process the results

        -- Give Results
        toController ! results1; ProcID; I; J
        toController ! results2; fOij; f1ij; cOij; mxOij; myOij; gij;
                       CHISQ; C1; MX1; MY1; DUDV
:
}}}F Worker Procedure
}}} LIB "WORKER"

}}} Libraries
```

```
{{{  PROGRAM Airflow controller (T414)
#USE AirPro

{{{  SC Airflow Controller for 4 transputers (with stats)
#USE AirPro
PROC T414CONTROLLER(CHAN OF ANY in, out,
                    CHAN OF messages fromW, toW, fromX, toX, fromY, toY)
  {{{  Library Usage
  #USE quadputer
  #USE TPKeyboard
  #USE TPScreen
  #USE TPDos
  #USE TPFile
  #USE TPGraphics
  #USE TPTerminator
  #USE ioconv
  #USE extrio
  #USE snglmath
  }}}  Library Usage

  {{{  Variable declarations
  -- T800 Usage
  [30]BYTE STR, fname, qname:
  INT IM, JM, LM, MM, MAXCOL:

  -- Added Variables
  INT SEED:
  TIMER clock:
  REAL32 xfactor, yfactor:
  INT16 MaxX, MaxY:
  BOOL error:

  INT IP, LC, LOOP, ICELL, NFULL:
  REAL32 C00, C1, G1, H, MX1, MY1, XMAX, YMAX, TEMP, TOT, CHISQ, DUDV:
  [10]REAL32 U, V, X, Y:
  [10][10]INT NCOL:
  [10][10]REAL32 G, MX0, MY0, C0, TCOL, mxtotal, mytotal, mxav, myav:
  [10][10][15][15]REAL32 F0, F1:

  VAL INT numprocs    IS 4:
  [numprocs]BOOL busy:
  INT numbusy:

  VAL INT unconnected IS 1:
  [unconnected]REAL32 c0queue, mx0queue, my0queue:
  [unconnected]INT iqueue, jqueue:
  INT inqueue:

  INT screeniterations, maxiterations, looptotal, restarts:

  BYTE driftfile, ecode:
  BOOL ptrace, piter, paver, ptotal:
  }}}  Variable declarations

  {{{  SC RAND FUNCTION  (Returns a random number between 0 and 1)
  REAL32, INT FUNCTION RAND(VAL INT SEED)
    #USE snglmath
    INT  newseed:
    REAL32 rand:
    VALOF
      SEQ
        newseed := SEED TIMES 259
        IF
          newseed < 0
            newseed := (newseed PLUS 2147483647) PLUS 1
          newseed = 0
            newseed := 2147483647
          TRUE
            SKIP
        rand := SCALEB(REAL32 ROUND newseed, -31)
      RESULT rand, newseed
```

```
:
}}}  SC RAND FUNCTION  (Returns a random number between 0 and 1)

{{{  Procedure to get nearest calculated cell and constants
PROC DoCell(VAL INT i, j)

  VAL icellm1 IS ICELL - 1:
  BYTE ecode:
  {{{  Find Nearest Unprocessed Cell
  PROC FindNearest(VAL INT i, j, INT neari, nearj, neardist)
    VAL icelld2 IS (ICELL+1) / 2:
    INT tempdist, tmpdist:
    SEQ
      IF
        NCOL[i][j] = (-1)
          SEQ
            {{{  Get closest distance to the side
            IF
              i < icelld2
                tmpdist := i
              TRUE
                tmpdist := icellm1 - i
            IF
              j < icelld2
                tempdist := j
              TRUE
                tempdist := icellm1 - j
            IF
              tempdist < tmpdist
                tmpdist := tempdist
              TRUE
                SKIP
            }}}  Get closest distance to the side

            {{{  Set closest distance to the side
            IF
              tmpdist < neardist
                neari, nearj, neardist := i, j, tmpdist
              TRUE
                SKIP
            }}}  Set closest distance to the side
        TRUE
          SKIP
  :
  }}}  Find Nearest Unprocessed Cell

  INT RI, RJ:
  REAL32 c0param, mx0param, my0param:

  {{{  Procedure to empty queue
  PROC emptyqueue(CHAN OF messages downchan)
    INT qi, qj:
    SEQ
      busy[3] := TRUE
      inqueue := inqueue - unconnected
      SEQ q = 0 FOR unconnected
        SEQ
          qi := iqueue[q]
          qj := jqueue[q]
          downchan ! toprocessor; Worker2; qi; qj; F0[qi][qj];
                     F1[qi][qj]; MX0[qi][qj]; MY0[qi][qj];
                     C0[qi][qj]; G[qi][qj];
                     c0queue[q]; mx0queue[q]; my0queue[q]
  :
  }}}  Procedure to empty queue
```

```
{{{  Procedure to give work
PROC givework(CHAN OF messages downchan, VAL INT WorkerNo)
  SEQ
    downchan ! toprocessor; WorkerNo; i; j; FO[i][j]; F1[i][j];
               MXO[i][j]; MYO[i][j]; CO[i][j]; G[i][j];
               c0param; mx0param; my0param
    busy[WorkerNo-1] := TRUE
:
}}}  Procedure to give work

{{{  Procedure to get results if any are available
PROC GetResults()
  BOOL nottimeout:
  TIMER clock:
  INT time:
  SEQ
    nottimeout := TRUE
    WHILE nottimeout OR (numbusy = numprocs)
      INT16 x, y:
      INT RI, RJ, ProcID:
      SEQ
        clock ? time
        PRI ALT
          fromW ? CASE
            point; x; y
              PutPixel(in,out,x,MaxY-y,White)
            results1; ProcID; RI; RJ
              SEQ
                fromW ? CASE
                  results2; FO[RI][RJ]; F1[RI][RJ]; CO[RI][RJ];
                            MXO[RI][RJ]; MYO[RI][RJ]; G[RI][RJ];
                            CHISQ; C1; MX1; MY1; DUDV
                    SEQ
                      busy[ProcID-1] := FALSE
                      numbusy := numbusy - 1
                      NFULL := NFULL + 1
                      NCOL[RI][RJ] := (-1)

                      mxtotal[RI][RJ] := mxtotal[RI][RJ] + MXO[RI][RJ]
                      mytotal[RI][RJ] := mytotal[RI][RJ] + MYO[RI][RJ]

          fromX ? CASE
            point; x; y
              PutPixel(in,out,x,MaxY-y,White)
            results1; ProcID; RI; RJ
              SEQ
                fromX ? CASE
                  results2; FO[RI][RJ]; F1[RI][RJ]; CO[RI][RJ];
                            MXO[RI][RJ]; MYO[RI][RJ]; G[RI][RJ];
                            CHISQ; C1; MX1; MY1; DUDV
                    SEQ
                      busy[ProcID-1] := FALSE
                      numbusy := numbusy - 1
                      NFULL := NFULL + 1
                      NCOL[RI][RJ] := (-1)

                      mxtotal[RI][RJ] := mxtotal[RI][RJ] + MXO[RI][RJ]
                      mytotal[RI][RJ] := mytotal[RI][RJ] + MYO[RI][RJ]

          fromY ? CASE
            point; x; y
              PutPixel(in,out,x,MaxY-y,White)
            results1; ProcID; RI; RJ
              SEQ
                fromY ? CASE
                  results2; FO[RI][RJ]; F1[RI][RJ]; CO[RI][RJ];
                            MXO[RI][RJ]; MYO[RI][RJ]; G[RI][RJ];
                            CHISQ; C1; MX1; MY1; DUDV
                    SEQ
                      busy[ProcID-1] := FALSE
                      numbusy := numbusy - 1
```

```
                    NFULL := NFULL + 1
                    NCOL[RI][RJ] := (-1)

                    mxtotal[RI][RJ] := mxtotal[RI][RJ] + MXO[RI][RJ]
                    mytotal[RI][RJ] := mytotal[RI][RJ] + MYO[RI][RJ]

        clock ? AFTER time
          SEQ
            IF
              ((numbusy <> 0) AND ((numbusy+NFULL) = (ICELL*ICELL))) OR
              ((busy[3] AND busy[2]) AND busy[1])
                SEQ
                  SKIP  -- Must still wait for results
              TRUE
                nottimeout := FALSE   -- Timed out, may give more more, or
                                      -- all finished.
:
}}}  Procedure to get results if any are available

SEQ
  {{{  Set the parameters
  IF
    (((i = 0) OR (i = icellm1)) OR (j = 0)) OR (j = icellm1)
      SEQ                                 -- Border cell
        c0param  := TEMP
        mx0param := 0.0 (REAL32)
        my0param := 0.0 (REAL32)
    TRUE
      INT neari, nearj, neardist:
      SEQ                                 -- Find calculated cell nearest border
        neari, nearj, neardist := (-1), (-1), ICELL
        WHILE neari = (-1)
          SEQ
            FindNearest((i-1), j, neari, nearj, neardist)  -- LEFT
            FindNearest((i+1), j, neari, nearj, neardist)  -- RIGHT
            FindNearest(i, (j-1), neari, nearj, neardist)  -- DOWN
            FindNearest(i, (j+1), neari, nearj, neardist)  -- UP

            IF
              neari = (-1)
                SEQ
                  GetResults()
                  IF
                    numbusy = 0
                      SEQ
                        OutTextXY(in,out,0 (INT16),8  (INT16),"BANG, T414 dead")
                        OutTextXY(in,out,0 (INT16),16 (INT16),"Work assignment ERROR")
                        STOP
                    TRUE
                      SKIP
              TRUE
                SEQ
                  c0param := C0[neari][nearj]
                  mx0param := MX0[neari][nearj]
                  my0param := MY0[neari][nearj]
  }}}  Set the parameters

  -- Send work to idle processor
  NCOL[i][j] := (-2)
  IF
    (NOT busy[0]) AND ((NOT busy[3]) AND (inqueue = unconnected))
      SEQ
        toW ! lotsofwork; unconnected
        emptyqueue(toW)
        givework(toW,WorkerW)
    NOT busy[1]
      SEQ
        givework(toX,WorkerX)
    NOT busy[2]
      SEQ
        givework(toY,WorkerY)
```

```
      (inqueue < unconnected)
        SEQ  -- Place work in queue
          iqueue[inqueue] := i
          jqueue[inqueue] := j
          c0queue[inqueue] := c0param
          mx0queue[inqueue] := mx0param
          my0queue[inqueue] := my0param
          inqueue := inqueue + 1
      TRUE
        SEQ          -- Oops, shouldn't be here
          OutTextXY(in,out,0 (INT16),8 (INT16),"BANG, T414 dead, shouldnt be here")
          OutTextXY(in,out,0 (INT16),16 (INT16),"Work had to go somewhere but couldnt")
          STOP

    numbusy := numbusy + 1

    -- Check all processors for results and if all processors are busy,
    -- wait until one is vacant.
    GetResults()
:
}}}  Procedure to get nearest calculated cell and constants

CHAN OF ANY toquad:
PLACE toquad AT 1:
SEQ
  boot.quadputer(in,out,toquad,error)
  IF
    error
      WriteLnString(in,out,  "Quadputer boot parameter expected")
    TRUE
      INT seedw, seedx, seedy, seedz:
      SEQ
        {{{  Rest of the program
        IM, JM, LM, MM := 10, 10, 10, 10
        LC, ICELL  := 7, 10

        ClrScr(in,out)
        WriteLnString(in,out,"Airflow Modelling  - 10x10 model")
        WriteLnString(in,out," ")
        {{{  Read In Number MAXCOL
        BOOL error:
        SEQ
          error := TRUE
          WriteString(in,out,"Enter the maximum collisions :")
          WHILE error
            INT size:
            BOOL notfound:
            SEQ
              ReadNumber(in,out,TRUE,STR)
              WriteLnString(in,out," ")
              {{{  Get the string size
              size := 0
              notfound := TRUE
              WHILE (size < (SIZE STR)) AND notfound
                SEQ
                  notfound := STR[size] <> 0 (BYTE)
                  IF
                    notfound
                      size := size + 1
                    TRUE
                      SKIP
              }}}  Get the string size
              STRINGTOINT(error,MAXCOL,[STR FROM 0 FOR size])
        }}}  Read In Number MAXCOL


        {{{  Read In Temperature TEMP
        BOOL error:
        SEQ
          error := TRUE
          WriteString(in,out,"Enter the starting wall temperature :")
```

```
  WHILE error
    INT size:
    BOOL notfound:
    SEQ
      ReadNumber(in,out,TRUE,STR)
      WriteLnString(in,out," ")
      {{{  Get the string size
      size := 0
      notfound := TRUE
      WHILE (size < (SIZE STR)) AND notfound
        SEQ
          notfound := STR[size] <> 0 (BYTE)
          IF
            notfound
              size := size + 1
            TRUE
              SKIP
      }}}  Get the string size
      STRINGTOREAL32(error,TEMP,[STR FROM 0 FOR size])
}}}  Read In Temperature TEMP

{{{  Total Iterations
BOOL error:
SEQ
  error := TRUE
  WriteString(in,out,"Enter number of iterations :")
  WHILE error
    INT size:
    BOOL notfound:
    SEQ
      ReadNumber(in,out,TRUE,STR)
      WriteLnString(in,out," ")
      {{{  Get the string size
      size := 0
      notfound := TRUE
      WHILE (size < (SIZE STR)) AND notfound
        SEQ
          notfound := STR[size] <> 0 (BYTE)
          IF
            notfound
              size := size + 1
            TRUE
              SKIP
      }}}  Get the string size
      STRINGTOINT(error,maxiterations,[STR FROM 0 FOR size])
}}}  Total Iterations

{{{  Print screen after ? Iterations
BOOL error:
SEQ
  error := TRUE
  WriteString(in,out,"Enter number of iterations to print screen after :")
  WHILE error
    INT size:
    BOOL notfound:
    SEQ
      ReadNumber(in,out,TRUE,STR)
      WriteLnString(in,out," ")
      {{{  Get the string size
      size := 0
      notfound := TRUE
      WHILE (size < (SIZE STR)) AND notfound
        SEQ
          notfound := STR[size] <> 0 (BYTE)
          IF
            notfound
              size := size + 1
            TRUE
              SKIP
      }}}  Get the string size
      STRINGTOINT(error,screeniterations,[STR FROM 0 FOR size])
```

```
}}}  Print screen after ? Iterations

{{{  Find what to print
BYTE ch:
SEQ
  {{{  Trace
  WriteString(in,out,"Print Trace ? (Y/N)")
  ReadKey(in,out,TRUE,ch)
  IF
    (ch = 'N') OR (ch = 'n')
      ptrace := FALSE
    TRUE
      ptrace := TRUE
  WriteLnString(in,out,"!")
  }}}  Trace

  {{{  Iteration
  WriteString(in,out,"Print Iteration Drift ? (Y/N)")
  ReadKey(in,out,TRUE,ch)
  IF
    (ch = 'N') OR (ch = 'n')
      piter := FALSE
    TRUE
      piter := TRUE
  WriteLnString(in,out,"!")
  }}}  Iteration

  {{{  Average
  WriteString(in,out,"Print Average Drift ? (Y/N)")
  ReadKey(in,out,TRUE,ch)
  IF
    (ch = 'N') OR (ch = 'n')
      paver := FALSE
    TRUE
      paver := TRUE
  WriteLnString(in,out,"!")
  }}}  Average

  {{{  Total
  WriteString(in,out,"Print Total Drift ? (Y/N)")
  ReadKey(in,out,TRUE,ch)
  IF
    (ch = 'N') OR (ch = 'n')
      ptotal := FALSE
    TRUE
      ptotal := TRUE
  WriteLnString(in,out,"!")
  }}}  Total
}}}  Find what to print

{{{  Read In File Name
WriteString(in,out,"Enter the drift data file name :")
INT size:
BOOL notfound:
BYTE ch:
SEQ
  ReadString(in,out,TRUE,fname)
  {{{  Get the string size
  size := 0
  notfound := TRUE
  WHILE (size < (SIZE STR)) AND notfound
    SEQ
      notfound := STR[size] <> 0 (BYTE)
      IF
        notfound
          size := size + 1
        TRUE
          SKIP
  }}}  Get the string size
```

```
    IF
      size = 0
        SEQ
          WriteLnString(in,out,"File DRIFT.DAT used, initialised to 0.0")
          [fname FROM 0 FOR 9] := "DRIFT.DAT"
          fname[9] := 0 (BYTE)
          looptotal := 0
          SEQ I = 0 FOR ICELL
            SEQ J = 0 FOR ICELL
              SEQ
                mxav[I][J] := 0.0 (REAL32)
                myav[I][J] := 0.0 (REAL32)
      TRUE
        INT16 nread:
        SEQ
          AssignFile(in,out,fname,driftfile,ecode)
          ResetFile(in,out,driftfile,ecode)
          ReadInt(in,out,driftfile,looptotal,ecode)
          [400]BYTE totalx RETYPES mxav:
          [400]BYTE totaly RETYPES myav:
          SEQ
            ReadBlock(in,out,driftfile,INT16 (SIZE totalx),totalx,nread,ecode)
            ReadBlock(in,out,driftfile,INT16 (SIZE totaly),totaly,nread,ecode)
          IF
            (nread <> 400 (INT16)) OR (ecode <> 0(BYTE))
              SEQ
                WriteLnString(in,out,"   File not loaded.")
                WriteLnString(in,out,"          ---- Initialised with 0.0")
                looptotal := 0
                SEQ I = 0 FOR ICELL
                  SEQ J = 0 FOR ICELL
                    SEQ
                      mxav[I][J] := 0.0 (REAL32)
                      myav[I][J] := 0.0 (REAL32)
            TRUE
              WriteLnString(in,out,"   File loaded O.K.")
          CloseFile(in,out,driftfile,ecode)
}}}  Read In File Name

{{{  Read In Number of Restarts
BOOL error:
SEQ
  error := TRUE
  WriteString(in,out,"Enter the number of restarts :")
  WHILE error
    INT size:
    BOOL notfound:
    SEQ
      ReadNumber(in,out,TRUE,STR)
      WriteLnString(in,out," ")
      {{{  Get the string size
      size := 0
      notfound := TRUE
      WHILE (size < (SIZE STR)) AND notfound
        SEQ
          notfound := STR[size] <> 0 (BYTE)
          IF
            notfound
              size := size + 1
            TRUE
              SKIP
      }}}  Get the string size
      STRINGTOINT(error,restarts,[STR FROM 0 FOR size])
}}}  Read In Number of Restarts
```

```
{{{  Read In Quadputer file name
IF
  restarts > 0
    SEQ
      WriteString(in,out,"Enter the quadputer boot file name :")
      INT size:
      BOOL notfound:
      BYTE ch:
      SEQ
        ReadString(in,out,TRUE,qname)
        {{{  Get the string size
        size := 0
        notfound := TRUE
        WHILE (size < (SIZE STR)) AND notfound
          SEQ
            notfound := STR[size] <> 0 (BYTE)
            IF
              notfound
                size := size + 1
              TRUE
                SKIP
        }}}  Get the string size
  TRUE
    SKIP
}}}  Read In Quadputer file name

SEQ mainportion = 0 FOR (restarts+1)
  SEQ
    {{{  Main Portion
    {{{  Set Random Number Seed
    INT16 hour, minute, second, sec100:
    SEQ
      GetTime(in,out,hour,minute,second,sec100)
      SEED := (INT hour) PLUS (((INT minute) * 60) PLUS ((INT second) * 3600))
      SEED := SEED PLUS ((INT sec100) * 360000)

    INT16 hour, minute, second, sec100:
    SEQ
      GetTime(in,out,hour,minute,second,sec100)
      seedw := (INT hour) PLUS (((INT minute) * 80) PLUS ((INT second) * 3800))
      seedw := seedw PLUS ((INT sec100) * 380000)

    INT16 hour, minute, second, sec100:
    SEQ
      GetTime(in,out,hour,minute,second,sec100)
      seedx := (INT hour) PLUS (((INT minute) * 100) PLUS ((INT second) * 4000))
      seedx := seedx PLUS ((INT sec100) * 400000)

    INT16 hour, minute, second, sec100:
    SEQ
      GetTime(in,out,hour,minute,second,sec100)
      seedy := (INT hour) PLUS (((INT minute) * 120) PLUS ((INT second) * 4200))
      seedy := seedy PLUS ((INT sec100) * 420000)

    INT16 hour, minute, second, sec100:
    SEQ
      GetTime(in,out,hour,minute,second,sec100)
      seedz := (INT hour) PLUS (((INT minute) * 140) PLUS ((INT second) * 4400))
      seedz := seedz PLUS ((INT sec100) * 440000)
    }}}  Set Random Number Seed

    {{{  Initialise graphics driver
    INT16 GDriver, GMode:
    SEQ
      GDriver := Detect
      InitGraph(in,out,GDriver,GMode,"C:\TP\GRAPHICS")
    }}}  Initialise graphics driver
```

```
{{{  Set graphics world
GetMaxX(in,out,MaxX)
xfactor := (REAL32 ROUND MaxX) / 500.0 (REAL32)
GetMaxY(in,out,MaxY)
yfactor := (REAL32 ROUND MaxY) / 333.0 (REAL32)
}}}  Set graphics world

{{{  Set the text font, style, size, line style and width
SetTextStyle(in,out,DefaultFont,HorizDir,1 (BYTE))
SetTextJustify(in,out,LeftText,TopText)
SetLineStyle(in,out,SolidLn,0 (INT16),NormWidth)
SetBkColor(in,out,Black)
}}}  Set the text font, style, size, line style and width

{{{  Set Fill Style
SetFillStyle(in,out,EmptyFill,Black)
}}}  Set Fill Style

{{{  Draw the top axis
SetColor(in,out,White)
Line(in,out,0 (INT16), MaxY - (INT16 ROUND (330.0 (REAL32) * yfactor)),
           0 (INT16), MaxY - (INT16 ROUND (260.0 (REAL32) * yfactor)) )
Line(in,out,0 (INT16), MaxY - (INT16 ROUND (260.0 (REAL32) * yfactor)),
           (INT16 ROUND (250.0 (REAL32) * xfactor)),
            MaxY - (INT16 ROUND (260.0 (REAL32) * yfactor)) )
}}}  Draw the top axis

BOOL BigLoop:
INT tstart, tend:
SEQ
  clock ? tstart
  H          := 250.0 (REAL32)
  XMAX, YMAX := H*(REAL32 TRUNC ICELL), H*(REAL32 TRUNC ICELL)
  MX1, MY1   := ZERO, ZERO
  C00        := 1.0 (REAL32)
  G1         := 0.3 (REAL32)
  IP         := 0                    -- USED TO BE 1
  C1         := 1.0 (REAL32)

  toW ! initialisepro; WorkerW; MAXCOL; seedw; LC; xfactor; yfactor; H; TEMP
  toX ! initialisepro; WorkerX; MAXCOL; seedx; LC; xfactor; yfactor; H; TEMP
  toY ! initialisepro; WorkerY; MAXCOL; seedy; LC; xfactor; yfactor; H; TEMP
  toW ! initialisepro; WorkerZ; MAXCOL; seedz; LC; xfactor; yfactor; H; TEMP

  {{{  Initialise distribution F0
  SEQ I = 0 FOR ICELL
    []REAL32 gi    IS G[I]:
    []REAL32 mxi   IS MX0[I]:
    []REAL32 myi   IS MY0[I]:
    []REAL32 c0i   IS C0[I]:
    []INT    ncoli IS NCOL[I]:
    SEQ J = 0 FOR ICELL
      SEQ
        gi[J]  := 0.3 (REAL32)
        mxi[J] := ZERO
        myi[J] := ZERO
        c0i[J] := 1.0 (REAL32)
        ncoli[J] := 0

  TOT := ZERO
  SEQ I = 0 FOR ICELL
    SEQ J = 0 FOR ICELL
      [][]REAL32 f0ij IS F0[I][J]:
      SEQ
        MX1, MY1 := MX0[I][J], MY0[I][J]
        C00 := C0[I][J]*C0[I][J]
        G1  := G[I][J]
        SEQ L = (-LC) FOR ((LC+LC)+1)
          SEQ M = (-LC) FOR ((LC+LC)+1)
            REAL32 tmp1, tmp2:
            SEQ
```

```
            tmp1 := (REAL32 TRUNC L)*G1
            tmp2 := (REAL32 TRUNC M)*G1
            fOij[7+L][7+M] := EXP((-((tmp1*tmp1)+(tmp2*tmp2)))/C00) / PI
}}}  Initialise distribution FO

SEQ I = 0 FOR ICELL
  SEQ J = 0 FOR ICELL
    SEQ
      mxtotal[I][J] := 0.0 (REAL32)
      mytotal[I][J] := 0.0 (REAL32)

LOOP := 0
BigLoop := TRUE

clock ? tend
REAL32 time:
INT len:
SEQ
  time := (REAL32 ROUND (tend MINUS tstart)) / 15625.0 (REAL32)
  OutTextXY(in,out,0 (INT16),0 (INT16),"TIME = ")
  REAL32TOSTRING(len,STR,time,5,3)
  STR[len] := 0 (BYTE)
  OutTextXY(in,out,64 (INT16),0 (INT16),STR)

{{{  Draw the right bottom axis
SetColor(in,out,White)
Line(in,out,(INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY,
           (INT16 ROUND (500.0 (REAL32) * xfactor)), MaxY )
Line(in,out,(INT16 ROUND (375.0 (REAL32) * xfactor)), MaxY,
           (INT16 ROUND (375.0 (REAL32) * xfactor)),
            MaxY - (INT16 ROUND (160.0 (REAL32) * yfactor))  )

}}}  Draw the right bottom axis

{{{  Draw the right top axis
SetColor(in,out,White)
Line(in,out,(INT16 ROUND (250.0 (REAL32) * xfactor)),
            MaxY - (INT16 ROUND (166.0 (REAL32) * yfactor)),
           (INT16 ROUND (500.0 (REAL32) * xfactor)),
            MaxY - (INT16 ROUND (166.0 (REAL32) * yfactor)) )
Line(in,out,(INT16 ROUND (375.0 (REAL32) * xfactor)),
            MaxY - (INT16 ROUND (166.0 (REAL32) * yfactor)),
           (INT16 ROUND (375.0 (REAL32) * xfactor)),
            MaxY - (INT16 ROUND (326.0 (REAL32) * yfactor)) )
}}}  Draw the right top axis

WHILE BigLoop
  REAL32 XP, YP:
  [2]BYTE page:
  SEQ
    page[0] := 12 (BYTE)
    page[1] := 0 (BYTE)

    clock ? tstart
    {{{  Draw the white box
    SetColor(in,out,White)
    Rectangle(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
                      (INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY)
    }}}  Draw the white box

    VAL middle IS ICELL / 2:
    SEQ
      LOOP := LOOP + 1
      XP := (REAL32 TRUNC LOOP) * 0.5 (REAL32)
      YP := 260.0 (REAL32) + (5.0 (REAL32) * C0[middle][middle])
      PutPixel(in,out,INT16 ROUND (XP*xfactor),
               MaxY-(INT16 ROUND (YP*yfactor)), White)
      YP := 260.0 (REAL32) + (5.0 (REAL32) * C0[middle-1][middle])
      PutPixel(in,out,INT16 ROUND (XP*xfactor),
               MaxY-(INT16 ROUND (YP*yfactor)), White)
```

```
numbusy, NFULL := 0, 0
inqueue := 0
SEQ p = 0 FOR numprocs
  busy[p] := FALSE

{{{  Zero Distribution F1
SEQ I = 0 FOR ICELL
  SEQ J = 0 FOR ICELL
    SEQ
      NCOL[I][J] := 0
      TCOL[I][J] := ZERO
      [][]REAL32 f1ij IS F1[I][J]:
      SEQ L = (-LC) FOR ((LC+LC)+1)
        SEQ M = (-LC) FOR ((LC+LC)+1)
          f1ij[7+L][7+M] := ZERO
}}}  Zero Distribution F1

{{{  Main worker loop
VAL icellm1 IS ICELL - 1:
VAL icellm2 IS ICELL - 2:

REAL32 r:
INT start, step:
SEQ
  {{{  Get random direction
  r,seedx := RAND(seedx)
  IF
    r > 0.5 (REAL32)
      SEQ
        start := 0
        step := 1
    TRUE
      SEQ
        start := icellm1
        step := (-1)
  }}}  Get random direction
  SEQ I = 0 FOR ICELL
    SEQ
      DoCell(start,0)          -- Bottom line
      start := start + step

  {{{  Get random direction
  r,seedx := RAND(seedx)
  IF
    r > 0.5 (REAL32)
      SEQ
        start := 0
        step := 1
    TRUE
      SEQ
        start := icellm1
        step := (-1)
  }}}  Get random direction
  SEQ I = 0 FOR ICELL
    SEQ
      DoCell(start,9)          -- Top line
      start := start + step

  {{{  Get random direction
  r,seedx := RAND(seedx)
  IF
    r > 0.5 (REAL32)
      SEQ
        start := 1
        step := 1
    TRUE
      SEQ
        start := icellm2
        step := (-1)
  }}}  Get random direction
```

```
                    iadd := 0
                  TRUE
                    SEQ
                      IF
                        jadd = 1
                          iadd := (-1)
                        TRUE
                          iadd := 1
                      J := J - jadd
                      I := I + iadd
                      jadd := 0
            TRUE
              SKIP
      TRUE
        INT len:
        SEQ
          {{{  Inverse direction
          IF
            iadd = 0
              SEQ
                iadd := 0-jadd
                jadd := 0
            TRUE
              SEQ
                jadd := iadd
                iadd := 0
          }}}  Inverse direction

          WHILE NCOL[I][J] >= 0
            SEQ
              DoCell(I,J)
              I := I + iadd
              J := J + jadd
              IF
                NCOL[I][J] = (-1)   -- Processed
                  SEQ
                    IF
                      (iadd <> 0)
                        SEQ
                          IF
                            iadd = 1
                              jadd := (-1)
                            TRUE
                              jadd := 1
                          I := I - iadd
                          J := J + jadd
                          iadd := 0
                      TRUE
                        SEQ
                          IF
                            jadd = 1
                              iadd := 1
                            TRUE
                              iadd := (-1)
                          J := J - jadd
                          I := I + iadd
                          jadd := 0
                TRUE
                  SKIP
    }}}  Random spiral
}}}  Main worker loop

{{{  Display the results
{{{  Clear axis screen
Bar(in,out, (INT16 ROUND (251.0 (REAL32) * xfactor)),
            0 (INT16),
          MaxX, MaxY          )
}}}  Clear axis screen
```

```
{{{  Draw the right bottom axis
SetColor(in,out,White)
Line(in,out,(INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY,
           (INT16 ROUND (500.0 (REAL32) * xfactor)), MaxY )
Line(in,out,(INT16 ROUND (375.0 (REAL32) * xfactor)), MaxY,
           (INT16 ROUND (375.0 (REAL32) * xfactor)),
            MaxY - (INT16 ROUND (160.0 (REAL32) * yfactor))  )

}}}  Draw the right bottom axis

{{{  Draw the right top axis
SetColor(in,out,White)
Line(in,out,(INT16 ROUND (250.0 (REAL32) * xfactor)),
            MaxY - (INT16 ROUND (166.0 (REAL32) * yfactor)),
           (INT16 ROUND (500.0 (REAL32) * xfactor)),
            MaxY - (INT16 ROUND (166.0 (REAL32) * yfactor)) )
Line(in,out,(INT16 ROUND (375.0 (REAL32) * xfactor)),
            MaxY - (INT16 ROUND (166.0 (REAL32) * yfactor)),
           (INT16 ROUND (375.0 (REAL32) * xfactor)),
            MaxY - (INT16 ROUND (326.0 (REAL32) * yfactor)) )
}}}  Draw the right top axis

INT middle:
SEQ
  middle := ICELL / 2
  {{{  Display White (BLUE) (Bottom right) distribution curves
  SetColor(in,out,White)
  SEQ
    {{{  Draw the circles
    SEQ K = (-LC) FOR ((LC+LC)+1)
      SEQ
        XP := 375.0 (REAL32) + (16.0 (REAL32) * (REAL32 TRUNC K))
        YP := 5000.0 (REAL32) * (DUDV*F1[middle][middle][7+K][7])
        Circle(in,out,INT16 ROUND (XP*xfactor),
               MaxY-(INT16 ROUND (YP*yfactor)) , 2 (INT16))
    }}}  Draw the circles

    {{{  Draw the actual distribution curve
    -- Move to start of distribution curve
    XP := 375.0 (REAL32) + (16.0 (REAL32) * (REAL32 TRUNC (-LC)))
    YP := 5000.0 (REAL32) * (DUDV*F0[middle][middle][7-LC][7])
    MoveTo(in,out, INT16 ROUND (XP*xfactor),
           MaxY - (INT16 ROUND (YP*yfactor)))

    -- The actual distribution curve
    SEQ K = (1-LC) FOR (LC+LC)
      SEQ
        XP := 375.0 (REAL32) + (16.0 (REAL32) * (REAL32 TRUNC K))
        YP := 5000.0 (REAL32) * (DUDV*F0[middle][middle][7+K][7])
        LineTo(in,out,(INT16 ROUND (XP*xfactor)), MaxY-(INT16 ROUND (YP*yfactor)))
    }}}  Draw the actual distribution curve
  }}}  Display White (BLUE) (Bottom right) distribution curves

  {{{  Display White (RED) (Top right) distribution curves
  SetColor(in,out,White)
  SEQ
    {{{  Draw the squares
    SEQ K = (-LC) FOR ((LC+LC)+1)
      INT16 x, y:
      SEQ
        XP := 375.0 (REAL32) + (16.0 (REAL32) * (REAL32 TRUNC K))
        YP := 166.0 (REAL32) + (5000.0 (REAL32) * (DUDV*F1[middle][middle][7][7+K]))
        x, y := (INT16 ROUND (XP*xfactor)), (INT16 ROUND (YP*yfactor))
        Rectangle(in,out, x - 2 (INT16), (MaxY - y) - 2 (INT16),
                          x + 2 (INT16), (MaxY - y) + 2 (INT16) )
    }}}  Draw the squares

    {{{  Draw the actual distribution curve
    -- Move to start of distribution curve
    XP := 375.0 (REAL32) + (16.0 (REAL32) * (REAL32 TRUNC (-LC)))
    YP := 166.0 (REAL32) + (5000.0 (REAL32) * (DUDV*F0[middle][middle][7][7-LC]))
```

```
    MoveTo(in,out, INT16 ROUND (XP*xfactor),
            MaxY - (INT16 ROUND (YP*yfactor)))

    -- The actual distribution curve
    SEQ K = (1-LC) FOR (LC+LC)
      SEQ
        XP := 375.0 (REAL32) + (16.0 (REAL32) * (REAL32 TRUNC K))
        YP := 166.0 (REAL32) + (5000.0 (REAL32)*(DUDV*F0[middle][middle][7][7+K]))
        LineTo(in,out,(INT16 ROUND (XP*xfactor)), MaxY-(INT16 ROUND (YP*yfactor)))
    }}}  Draw the actual distribution curve
  }}}  Display White (RED) (Top right) distribution curves

{{{  Display Text
INT len:
INT16 mid:
SEQ
  {{{  Display the loop
  OutTextXY(in,out,(MaxX / 2 (INT16)),0 (INT16),"LOOP = ")
  INTTOSTRING(len,STR,LOOP)
  STR[len] := 0 (BYTE)
  OutTextXY(in,out,64 (INT16) + (MaxX / 2 (INT16)),0 (INT16),STR)
  }}}  Display the loop

  {{{  Display CHI square
  mid := (INT16 ROUND (400.0 (REAL32) * xfactor))
  OutTextXY(in,out,mid,0 (INT16),"CHISQ = ")
  REAL32TOSTRING(len,STR,CHISQ,0,0)
  STR[len] := 0 (BYTE)
  OutTextXY(in,out,MaxX - 80(INT16),0 (INT16),STR)
  }}}  Display CHI square

  {{{  Display C0
  OutTextXY(in,out,mid,16 (INT16),"C1 = ")
  REAL32TOSTRING(len,STR,C1,0,0)
  STR[len] := 0 (BYTE)
  OutTextXY(in,out,MaxX - 80 (INT16),16 (INT16),STR)
  }}}  Display C0

  {{{  Display MY1
  OutTextXY(in,out,mid,MaxY - (INT16 ROUND (300.0 (REAL32) * yfactor)),
                   "MY1 = ")
  REAL32TOSTRING(len,STR,MY1,0,0)
  STR[len] := 0 (BYTE)
  OutTextXY(in,out,MaxX-80 (INT16),MaxY-(INT16 ROUND (300.0 (REAL32)*yfactor)),
                   STR)
  }}}  Display MY1

  {{{  Display MX1
  OutTextXY(in,out,mid,MaxY - (INT16 ROUND (150.0 (REAL32) * yfactor)),
                   "MX1 = ")
  REAL32TOSTRING(len,STR,MX1,0,0)
  STR[len] := 0 (BYTE)
  OutTextXY(in,out,MaxX-80 (INT16),MaxY-(INT16 ROUND (150.0 (REAL32) * yfactor)),
                   STR)
  }}}  Display MX1
}}}  Display Text

{{{  Display Time
clock ? tend

Bar(in,out, 0 (INT16), 0 (INT16), 144 (INT16), 9 (INT16))

REAL32 time:
INT len:
SEQ
  time := (REAL32 ROUND (tend MINUS tstart)) / 15625.0 (REAL32)
  OutTextXY(in,out,0 (INT16),0 (INT16),"TIME = ")
  REAL32TOSTRING(len,STR,time,5,3)
  STR[len] := 0 (BYTE)
  OutTextXY(in,out,64 (INT16),0 (INT16),STR)
}}}  Display Time
```

```
}}}  Display the results

{{{  Check for print screen
IF
  ((LOOP REM screeniterations) = 0) AND ptrace
    BYTE ecode:
    SEQ
      PrintScreen(in,out)
      WriteTextLine(in,out,PRINTER,page,ecode)
  TRUE
    IF
      ((LOOP REM maxiterations) = 0)
        BYTE ch:
        SEQ
          ReadKey(in,out,FALSE,ch)
      TRUE
        SKIP
}}}  Check for print screen

{{{  Display the MX,MY drift for this iteration
SEQ
  Bar(in,out, 0 (INT16), 0 (INT16), 144 (INT16), 9 (INT16))
  OutTextXY(in,out,0 (INT16),0 (INT16),"Iteration drift")

  -- Clear Screen
  Bar(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
              (INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY)

  -- Draw the white box
  SetColor(in,out,White)
  Rectangle(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
                    (INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY)

  -- Draw the average drift for this iteration
  REAL32 size:
  SEQ i = 0 FOR ICELL
    SEQ j = 0 FOR ICELL
      REAL32 x1, x2, y1, y2:
      VAL REAL32 mx0ij IS MX0[i][j]:
      VAL REAL32 my0ij IS MY0[i][j]:
      SEQ
        size := SQRT((mx0ij*mx0ij) + (my0ij*my0ij)) * 2.0 (REAL32)
        x1 := ((H * ((REAL32 TRUNC i) + 0.5 (REAL32))) * 25.0 (REAL32)) / H
        y1 := ((H * ((REAL32 TRUNC j) + 0.5 (REAL32))) * 25.0 (REAL32)) / H
        x2 := x1 + ((MX0[i][j] / size) * 25.0 (REAL32))
        y2 := y1 + ((MY0[i][j] / size) * 25.0 (REAL32))
        Line(in,out,(INT16 ROUND (x1*xfactor)),MaxY-(INT16 ROUND (y1*yfactor)),
                    (INT16 ROUND (x2*xfactor)),MaxY-(INT16 ROUND (y2*yfactor)))
}}}  Display the MX,MY drift for this iteration

{{{  Check for print screen
IF
  ((LOOP REM screeniterations) = 0) AND piter
    BYTE ecode:
    SEQ
      PrintScreen(in,out)
      WriteTextLine(in,out,PRINTER,page,ecode)
  TRUE
    IF
      ((LOOP REM maxiterations) = 0)
        BYTE ch:
        SEQ
          ReadKey(in,out,FALSE,ch)
      TRUE
        SKIP
}}}  Check for print screen
```

```
{{{  Display the numbers for the MX,MY drift
SEQ
  Bar(in,out, 0 (INT16), 0 (INT16), 144 (INT16), 9 (INT16))
  OutTextXY(in,out,0 (INT16),0 (INT16),"MX,MY,CO values")

  SetTextJustify(in,out,CenterText,CenterText)

  -- Clear Screen
  Bar(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
              MaxX, MaxY)

  -- Draw the white box
  SetColor(in,out,White)
  Rectangle(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
                    MaxX, MaxY)

  -- Draw the average drift for this iteration
  REAL32 size:
  SEQ i = 0 FOR ICELL
    SEQ j = 0 FOR ICELL
      REAL32 x1, y1:
      VAL REAL32 mx0ij IS MX0[i][j]:
      VAL REAL32 my0ij IS MY0[i][j]:
      INT len:
      SEQ
        size := SQRT((mx0ij*mx0ij) + (my0ij*my0ij))
        x1 := ((H * ((REAL32 TRUNC i) + 0.5 (REAL32))) * 50.0 (REAL32)) / H
        y1 := ((H * ((REAL32 TRUNC j) + 0.5 (REAL32))) * 25.0 (REAL32)) / H
        {{{  Display MX0
        REAL32TOSTRING(len,STR,MX0[i][j],1,5)
        STR[len] := 0 (BYTE)
        OutTextXY(in,out,(INT16 ROUND (x1*xfactor)),
               (MaxY-(INT16 ROUND (y1*yfactor)))-8(INT16),STR)
        }}}  Display MX0

        {{{  Display MY0
        REAL32TOSTRING(len,STR,MY0[i][j],1,5)
        STR[len] := 0 (BYTE)
        OutTextXY(in,out,(INT16 ROUND (x1*xfactor)),
               (MaxY-(INT16 ROUND (y1*yfactor))),STR)
        }}}  Display MY0

        {{{  Display CO
        REAL32TOSTRING(len,STR,CO[i][j],1,5)
        STR[len] := 0 (BYTE)
        OutTextXY(in,out,(INT16 ROUND (x1*xfactor)),
               (MaxY-(INT16 ROUND (y1*yfactor)))+8(INT16),STR)
        }}}  Display CO

SetTextJustify(in,out,LeftText,TopText)
}}}  Display the numbers for the MX,MY drift

{{{  Check for print screen
IF
  ((LOOP REM screeniterations) = 0) AND piter
    BYTE ecode:
    SEQ
      PrintScreen(in,out)
      WriteTextLine(in,out,PRINTER,page,ecode)
  TRUE
    IF
      ((LOOP REM maxiterations) = 0)
        BYTE ch:
        SEQ
          ReadKey(in,out,FALSE,ch)
      TRUE
        SKIP
}}}  Check for print screen
```

```
{{{  Display the average MX,MY drift
SEQ
  Bar(in,out, 0 (INT16), 0 (INT16), 144 (INT16), 9 (INT16))
  OutTextXY(in,out,0 (INT16),0 (INT16),"Average drift")

  -- Clear Screen
  Bar(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
              (INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY)

  -- Draw the white box
  SetColor(in,out,White)
  Rectangle(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
                    (INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY)

  -- Draw the average drift for this iteration
  REAL32 size:
  SEQ i = 0 FOR ICELL
    SEQ j = 0 FOR ICELL
      REAL32 x1, x2, y1, y2:
      VAL REAL32 mx0ij IS mxtotal[i][j]:
      VAL REAL32 my0ij IS mytotal[i][j]:
      SEQ
        size := SQRT((mx0ij*mx0ij) + (my0ij*my0ij)) * 2.0 (REAL32)
        x1 := ((H * ((REAL32 TRUNC i) + 0.5 (REAL32))) * 25.0 (REAL32)) / H
        y1 := ((H * ((REAL32 TRUNC j) + 0.5 (REAL32))) * 25.0 (REAL32)) / H
        x2 := x1 + ((mxtotal[i][j] / size) * 25.0 (REAL32))
        y2 := y1 + ((mytotal[i][j] / size) * 25.0 (REAL32))
        Line(in,out,(INT16 ROUND (x1*xfactor)),MaxY-(INT16 ROUND (y1*yfactor)),
                    (INT16 ROUND (x2*xfactor)),MaxY-(INT16 ROUND (y2*yfactor)))
}}}  Display the average MX,MY drift

{{{  Check for print screen
IF
  ((LOOP REM screeniterations) = 0) AND paver
    BYTE ecode:
    SEQ
      PrintScreen(in,out)
      WriteTextLine(in,out,PRINTER,page,ecode)
  TRUE
    IF
      ((LOOP REM maxiterations) = 0)
        BYTE ch:
        SEQ
          ReadKey(in,out,FALSE,ch)
      TRUE
        SKIP
}}}  Check for print screen

{{{  Check for loop again
IF
  ((LOOP REM maxiterations) = 0)
    {{{  Get a key, checking for loop again
    BYTE ch:
    SEQ
      ReadKey(in,out,FALSE,ch)
      IF
        (ch = 'x') OR (ch = 'X')
          SEQ
            BigLoop := FALSE                -- Exit loop

            looptotal := looptotal + LOOP
            SEQ i = 0 FOR ICELL
              SEQ j = 0 FOR ICELL
                SEQ
                  mxav[i][j] := mxav[i][j] + mxtotal[i][j]
                  myav[i][j] := myav[i][j] + mytotal[i][j]
```

```
                    {{{  Display the total MX,MY drift
                    SEQ
                      Bar(in,out, 0 (INT16), 0 (INT16), 144 (INT16), 9 (INT16))
                      OutTextXY(in,out,0 (INT16),0 (INT16),"Total drift")

                      -- Clear Screen
                      Bar(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
                                  (INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY)

                      -- Draw the white box
                      SetColor(in,out,White)
                      Rectangle(in,out,0(INT16),(MaxY-(INT16 ROUND (250.0(REAL32)*yfactor))),
                                       (INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY)

                      -- Draw the average drift for this iteration
                      REAL32 size:
                      SEQ i = 0 FOR ICELL
                        SEQ j = 0 FOR ICELL
                          VAL REAL32 mx0ij IS mxav[i][j]:
                          VAL REAL32 my0ij IS myav[i][j]:
                          REAL32 x1, x2, y1, y2:
                          SEQ
                            size := SQRT((mx0ij*mx0ij) + (my0ij*my0ij)) * 2.0 (REAL32)
                            x1 := ((H * ((REAL32 TRUNC i) + 0.5(REAL32))) * 25.0(REAL32)) / H
                            y1 := ((H * ((REAL32 TRUNC j) + 0.5(REAL32))) * 25.0(REAL32)) / H
                            x2 := x1 + ((mxav[i][j] / size) * 25.0 (REAL32))
                            y2 := y1 + ((myav[i][j] / size) * 25.0 (REAL32))
                            Line(in,out,(INT16 ROUND(x1*xfactor)),
                                        MaxY-(INT16 ROUND(y1*yfactor)),
                                       (INT16 ROUND (x2*xfactor)),
                                        MaxY-(INT16 ROUND (y2*yfactor)))
                    }}}  Display the total MX,MY drift

                    {{{  Check for print screen
                    IF
                      ptotal
                        BYTE ecode:
                        SEQ
                          PrintScreen(in,out)
                          WriteTextLine(in,out,PRINTER,page,ecode)
                      TRUE
                        BYTE ch:
                        SEQ
                          ReadKey(in,out,FALSE,ch)
                      Bar(in,out, 0 (INT16), 0 (INT16), 144 (INT16), 9 (INT16))
                      }}}  Check for.print screen
                  TRUE
                    SEQ                              -- Clear Screen
                      Bar(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
                                  (INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY)
            }}}  Get a key, checking for loop again
          TRUE
            SEQ                              -- Clear Screen
              Bar(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
                          (INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY)
        }}}  Check for loop again

CloseGraph(in,out)

-- Check if reboot quadputer
IF
  mainportion < restarts
    BYTE ch:
    BOOL error:
    SEQ
      boot.file.quadputer(in,out,toquad,qname,error)
      IF
        error
          WriteLnString(in,out,"Quadputer not rebooted,  system will hang.")
        TRUE
          WriteLnString(in,out,"Quadputer booted.")
```

```
                ReadKey(in,out,FALSE,ch)
              TRUE
                SKIP
            }}}  Main Portion

          {{{  Write Results to file
          -- Open and start writing
          AssignFile(in,out,fname,driftfile,ecode)
          RewriteFile(in,out,driftfile,ecode)
          WriteInt(in,out,driftfile,looptotal,ecode)
          [400]BYTE totalx RETYPES mxav:
          [400]BYTE totaly RETYPES myav:
          INT16 nwritten:
          SEQ
            WriteBlock(in,out,driftfile,INT16 (SIZE totalx), totalx, nwritten, ecode)
            WriteBlock(in,out,driftfile,INT16 (SIZE totaly), totaly, nwritten, ecode)
            IF
              nwritten <> (INT16 (SIZE totaly))
                WriteLnString(in,out,"ERROR IN WRITING FILE")
              TRUE
                WriteLnString(in,out,"FILE WRITTEN OK")
          CloseFile(in,out,driftfile,ecode)
          }}}  Write Results to file
          }}}  Rest of the program

    TerminateServer(in,out)
:
}}}  SC Airflow Controller for 4 transputers (with stats)

CHAN OF ANY PC2B, B2PC:
CHAN OF messages W2B, B2W, X2B, B2X, Y2B, B2Y:
PROCESSOR 0 T4
  PLACE PC2B AT 4:
  PLACE W2B  AT 5:
  PLACE X2B  AT 6:
  PLACE Y2B  AT 7:
  PLACE B2PC AT 0:
  PLACE B2W  AT 1:
  PLACE B2X  AT 2:
  PLACE B2Y  AT 3:
  T414CONTROLLER(PC2B, B2PC, W2B, B2W, X2B, B2X, Y2B, B2Y)
}}}  PROGRAM Airflow controller (T414)
```

```
{{{  PROGRAM Airflow Worker network
#USE AirPro

{{{  SC WorkerWXY
#USE AirPro
#USE NewWorker2
PROC WorkerWXY(VAL INT workernum,
               CHAN OF messages fromCont, toCont, passRec, passSend)

  CHAN OF messages toWorker, fromWorker:
  PRI PAR
    -- ROUTER
    SEQ
      WHILE TRUE
        INT16 x, y:
        INT maxcol, seed, lc, i, j, procno, amount:
        REAL32 gxfact, gyfact, h, temp, mx0ij, my0ij, c0ij, gij,
               c0param, mx0param, my0param, chisq, c1, mx1, my1, dudv:
        [15][15]REAL32 f0ij, f1ij:
        ALT
          {{{  Messages from controller
          fromCont   ? CASE
            initialisepro; procno; maxcol; seed; lc; gxfact; gyfact; h; temp
              {{{  Pass initialise message on or give to this worker
              IF
                procno = workernum
                  -- Send work to this worker
                  toWorker ! initialise; maxcol; seed; lc;
                             gxfact; gyfact; h; temp
                TRUE
                  -- Pass work onto the next processor
                  passSend ! initialisepro; procno; maxcol; seed; lc;
                             gxfact; gyfact; h; temp
              }}}  Pass initialise message on or give to this worker

            lotsofwork; amount
              SEQ I = 0 FOR (amount+1)
                fromCont ? CASE
                  toprocessor; procno; i; j; f0ij; f1ij; mx0ij; my0ij; c0ij; gij;
                               c0param; mx0param; my0param
                    {{{  Pass work on or give to this worker
                    IF
                      procno = workernum
                        -- Send work to this worker
                        toWorker ! work; i; j; f0ij; f1ij; mx0ij; my0ij; c0ij; gij;
                                   c0param; mx0param; my0param
                      TRUE
                        -- Pass work onto the next processor
                        passSend ! toprocessor; procno; i; j; f0ij; f1ij; mx0ij; my0ij; c0ij; gij;
                                   c0param; mx0param; my0param
                    }}}  Pass work on or give to this worker

            toprocessor; procno; i; j; f0ij; f1ij; mx0ij; my0ij; c0ij; gij;
                         c0param; mx0param; my0param
              {{{  Pass work on or give to this worker
              IF
                procno = workernum
                  -- Send work to this worker
                  toWorker ! work; i; j; f0ij; f1ij; mx0ij; my0ij; c0ij; gij;
                             c0param; mx0param; my0param
                TRUE
                  -- Pass work onto the next processor
                  passSend ! toprocessor; procno; i; j; f0ij; f1ij; mx0ij; my0ij; c0ij; gij;
                             c0param; mx0param; my0param
              }}}  Pass work on or give to this worker
          }}}  Messages from controller
```

```
        {{{  Messages from other processor
        passRec    ? CASE
          point; x; y
            toCont ! point; x; y
          results1; procno; i; j
            passRec ? CASE
              results2; f0ij; f1ij; c0ij; mx0ij; my0ij; gij;
                        chisq; c1; mx1; my1; dudv
                SEQ
                  toCont ! results1; procno; i; j
                  toCont ! results2; f0ij; f1ij; c0ij; mx0ij; my0ij; gij;
                            chisq; c1; mx1; my1; dudv
        }}}  Messages from other processor

        {{{  Messages from this worker
        fromWorker ? CASE
          point; x; y
            toCont ! point; x; y
          results1; procno; i; j
            fromWorker ? CASE
              results2; f0ij; f1ij; c0ij; mx0ij; my0ij; gij;
                        chisq; c1; mx1; my1; dudv
                SEQ
                  toCont ! results1; procno; i; j
                  toCont ! results2; f0ij; f1ij; c0ij; mx0ij; my0ij; gij;
                            chisq; c1; mx1; my1; dudv
        }}}  Messages from this worker

    -- WORKER
    Worker(workernum, toWorker, fromWorker)
:
}}}  SC WorkerWXY

{{{  SC WorkerZ
#USE AirPro
#USE NewWorker2
PROC WorkerZ(VAL INT workernum,  CHAN OF messages from1, to1)
  CHAN OF messages toWorker, fromWorker:
  PRI PAR
    -- ROUTER
    SEQ
      WHILE TRUE
        INT16 x, y:
        INT maxcol, seed, lc, i, j, procno:
        REAL32 gxfact, gyfact, h, temp, mx0ij, my0ij, c0ij, gij,
               c0param, mx0param, my0param, chisq, c1, mx1, my1, dudv:
        [15][15]REAL32 f0ij, f1ij:
        REAL32 totx, toty:
        SEQ
          ALT
            {{{  Messages from 1
            from1   ? CASE
              initialisepro; procno; maxcol; seed; lc; gxfact; gyfact; h; temp
                {{{  Give to this worker
                IF
                  procno = workernum
                    -- Send work to this worker
                    toWorker ! initialise; maxcol; seed; lc;
                               gxfact; gyfact; h; temp
                  TRUE
                    STOP   -- Error, last processor in list so message MUST be destined here
                }}}  Give to this worker

              toprocessor; procno; i; j; f0ij; f1ij; mx0ij; my0ij; c0ij; gij;
                           c0param; mx0param; my0param
                {{{  Give to this worker
                IF
                  procno = workernum
                    -- Send work to this worker
                    toWorker ! work; i; j; f0ij; f1ij; mx0ij; my0ij; c0ij; gij;
                               c0param; mx0param; my0param
```

```
                TRUE
                  STOP   -- Error, last processor in list so message MUST be destined here
              }}}  Give to this worker
          }}}  Messages from 1

          {{{  Messages from this worker
          fromWorker ? CASE
            point; x; y
              to1 ! point; x; y
            results1; procno; i; j
              fromWorker ? CASE
                results2; f0ij; f1ij; c0ij; mx0ij; my0ij; gij;
                          chisq; c1; mx1; my1; dudv
                  SEQ
                    to1 ! results1; procno; i; j
                    to1 ! results2; f0ij; f1ij; c0ij; mx0ij; my0ij; gij;
                          chisq; c1; mx1; my1; dudv
          }}}  Messages from this worker

    -- WORKER
    Worker(workernum, toWorker, fromWorker)
:
}}}  SC WorkerZ

CHAN OF messages B2W, W2B, B2X, X2B, B2Y, Y2B,
                 W2Z, Z2W, X2Z, Z2X, Y2Z, Z2Y:
PLACED PAR
  PROCESSOR 1 T8
    PLACE W2B AT 0:
    PLACE W2Z AT 3:
    PLACE B2W AT 4:
    PLACE Z2W AT 7:
    WorkerWXY(1, B2W, W2B, Z2W, W2Z)

  PROCESSOR 2 T8
    PLACE X2B AT 1:
    PLACE X2Z AT 3:
    PLACE B2X AT 5:
    PLACE Z2X AT 7:
    WorkerWXY(2, B2X, X2B, Z2X, X2Z)

  PROCESSOR 3 T8
    PLACE Y2B AT 2:
    PLACE Y2Z AT 3:
    PLACE B2Y AT 6:
    PLACE Z2Y AT 7:
    WorkerWXY(3, B2Y, Y2B, Z2Y, Y2Z)

  PROCESSOR 4 T8
    PLACE Z2W AT 0:
    PLACE Z2X AT 1:
    PLACE Z2Y AT 2:
    PLACE W2Z AT 4:
    PLACE X2Z AT 5:
    PLACE Y2Z AT 6:
    WorkerZ(4, W2Z, Z2W)
}}}  PROGRAM Airflow Worker network
```

```
{{{  PROGRAM Airflow results display program
{{{  SC Airflow results display program
#USE AirPro
PROC T414RESULTS(CHAN OF ANY in,out)
  #USE TPKeyboard
  #USE TPScreen
  #USE TPDos
  #USE TPFile
  #USE TPGraphics
  #USE TPTerminator
  #USE ioconv
  #USE extrio
  #USE snglmath

  -- T800 Usage
  [30]BYTE STR, fname, qname:

  -- Added Variables
  REAL32 xfactor, yfactor:
  INT16 MaxX, MaxY:
  BOOL error:

  [10][10][2]REAL32 total2:
  BYTE driftfile, ecode:
  INT looptotal:
  BOOL ptotal:

  SEQ
    ClrScr(in,out)
    WriteLnString(in,out,"Airflow Modelling  - 10x10 model Results display program")
    WriteLnString(in,out," ")
    {{{  Find what to print
    BYTE ch:
    SEQ
      {{{  Total
      WriteString(in,out,"Print Screen Total Drift ? (Y/N)")
      ReadKey(in,out,TRUE,ch)
      IF
        (ch = 'N') OR (ch = 'n')
          ptotal := FALSE
        TRUE
          ptotal := TRUE
      WriteLnString(in,out,"!")
      }}}  Total
    }}}  Find what to print

    {{{  Read In File Name
    INT size:
    BOOL notfound:
    BYTE ch:
    SEQ
      notfound := TRUE
      WHILE notfound
        SEQ
          WriteString(in,out,"Enter the file name :")
          ReadString(in,out,TRUE,fname)
          {{{  Get the string size
          size := 0
          notfound := TRUE
          WHILE (size < (SIZE STR)) AND notfound
            SEQ
              notfound := STR[size] <> 0 (BYTE)
              IF
                notfound
                  size := size + 1
                TRUE
                  SKIP
          }}}  Get the string size
          IF
            size = 0
              WriteLnString(in,out,"File name required, retry !")
```

```
        TRUE
          INT16 nread:
          SEQ
            AssignFile(in,out,fname,driftfile,ecode)
            ResetFile(in,out,driftfile,ecode)
            IF
              ecode = 0 (BYTE)
                SEQ
                  ReadInt(in,out,driftfile,looptotal,ecode)
                  [800]BYTE totalB RETYPES total2:
                  SEQ
                    ReadBlock(in,out,driftfile,INT16 (SIZE totalB),totalB,nread,ecode)
                  IF
                    (nread <> 800 (INT16)) OR (ecode <> 0(BYTE))
                      WriteLnString(in,out,"Error in reading file. Retry !")
                    TRUE
                      SEQ
                        WriteLnString(in,out,"   File loaded O.K.")
                        notfound := FALSE
              TRUE
                WriteLnString(in,out,"File not found. Retry !")
            CloseFile(in,out,driftfile,ecode)
}}} Read In File Name

{{{ Initialise graphics driver
INT16 GDriver, GMode:
SEQ
  GDriver := Detect
  InitGraph(in,out,GDriver,GMode,"C:\TP\GRAPHICS")
}}} Initialise graphics driver

{{{ Set graphics world
GetMaxX(in,out,MaxX)
xfactor := (REAL32 ROUND MaxX) / 500.0 (REAL32)
GetMaxY(in,out,MaxY)
yfactor := (REAL32 ROUND MaxY) / 333.0 (REAL32)
}}} Set graphics world

{{{ Set the text font, style, size, line style and width
SetTextStyle(in,out,DefaultFont,HorizDir,1 (BYTE))
SetTextJustify(in,out,LeftText,TopText)
SetLineStyle(in,out,SolidLn,0 (INT16),NormWidth)
SetBkColor(in,out,Black)
}}} Set the text font, style, size, line style and width

{{{ Set Fill Style
SetFillStyle(in,out,EmptyFill,Black)
}}} Set Fill Style

{{{ Draw the white box
SetColor(in,out,White)
Rectangle(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
                  (INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY)
}}} Draw the white box

{{{ Display the average drift
SEQ
  Bar(in,out, 0 (INT16), 0 (INT16), 144 (INT16), 9 (INT16))
  OutTextXY(in,out,0 (INT16),0 (INT16),"Total drift")

  -- Clear Screen
  Bar(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
              (INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY)

  -- Draw the white box
  SetColor(in,out,White)
  Rectangle(in,out, 0 (INT16),(MaxY - (INT16 ROUND (250.0 (REAL32) * yfactor))),
                    (INT16 ROUND (250.0 (REAL32) * xfactor)), MaxY)
```

```
  -- Draw the average drift
  SEQ i = 0 FOR 10
    SEQ j = 0 FOR 10
      VAL REAL32 H IS 250.0 (REAL32):
      REAL32 x1, x2, y1, y2:
      SEQ
        x1 := ((H * ((REAL32 TRUNC i) + 0.5 (REAL32))) * 25.0 (REAL32)) / H
        y1 := ((H * ((REAL32 TRUNC j) + 0.5 (REAL32))) * 25.0 (REAL32)) / H
        x2 := ((total2[i][j][0]/(REAL32 ROUND looptotal)) * 25.0 (REAL32)) / H
        y2 := ((total2[i][j][1]/(REAL32 ROUND looptotal)) * 25.0 (REAL32)) / H
        Line(in,out,(INT16 ROUND (x1*xfactor)),MaxY-(INT16 ROUND (y1*yfactor)),
                    (INT16 ROUND (x2*xfactor)),MaxY-(INT16 ROUND (y2*yfactor)))
}}}  Display the average drift

{{{  Check for print screen
IF
  ptotal
    BYTE ecode:
    VAL PRINTER IS 255 (BYTE):
    SEQ
      PrintScreen(in,out)
      WriteTextLine(in,out,PRINTER," ",ecode)
      WriteTextLine(in,out,PRINTER," ",ecode)
      WriteTextLine(in,out,PRINTER," ",ecode)
      WriteTextLine(in,out,PRINTER," ",ecode)
      WriteTextLine(in,out,PRINTER," ",ecode)
      WriteTextLine(in,out,PRINTER," ",ecode)
      WriteTextLine(in,out,PRINTER," ",ecode)
      WriteTextLine(in,out,PRINTER," ",ecode)
      WriteTextLine(in,out,PRINTER," ",ecode)
  TRUE
    BYTE ch:
    SEQ
      ReadKey(in,out,FALSE,ch)
}}}  Check for print screen

CloseGraph(in,out)

{{{  Check for print screen and print numbers for the drift
REAL32 xav, yav:
INT len:
IF
  ptotal
    BYTE ecode:
    VAL PRINTER IS 255 (BYTE):
    VAL H IS 250.0 (REAL32):
    SEQ
      WriteTextLine(in,out,PRINTER,"DRIFT FOR CELLS - X, then Y",ecode)
      WriteTextLine(in,out,PRINTER,"---------------",ecode)
      WriteTextLine(in,out,PRINTER,"",ecode)

      xav, yav := 0.0 (REAL32), 0.0 (REAL32)
      SEQ i = 0 FOR 10
        SEQ
          -- Display X
          SEQ j = 0 FOR 10
            REAL32 mid, num:
            SEQ
              mid := (H * ((REAL32 TRUNC j) + 0.5 (REAL32)))
              num := (total2[j][9-i][0] / (REAL32 ROUND looptotal)) - mid
              xav := xav + num
              REAL32TOSTRING(len,STR,num,2,3)
              STR[len] := 0 (BYTE)
              WriteText(in,out,PRINTER,STR,ecode)
              WriteText(in,out,PRINTER," ",ecode)
          WriteTextLine(in,out,PRINTER,"",ecode)
```

```
        -- Display Y
        SEQ j = 0 FOR 10
          INT len:
          REAL32 mid, num:
          SEQ
            mid := (H * ((REAL32 TRUNC (9-i)) + 0.5 (REAL32)))
            num := (total2[j][9-i][1] / (REAL32 ROUND looptotal)) - mid
            yav := yav + num
            REAL32TOSTRING(len,STR,num,2,3)
            STR[len] := 0 (BYTE)
            WriteText(in,out,PRINTER,STR,ecode)
            WriteText(in,out,PRINTER," ",ecode)
        WriteTextLine(in,out,PRINTER,"",ecode)
        WriteTextLine(in,out,PRINTER,"",ecode)

    xav := xav / 100.0 (REAL32)
    yav := yav / 100.0 (REAL32)
    WriteTextLine(in,out,PRINTER,"",ecode)
    WriteTextLine(in,out,PRINTER,"",ecode)
    WriteTextLine(in,out,PRINTER,"Average X/Y drift for all cells",ecode)
    REAL32TOSTRING(len,STR,xav,3,3)
    STR[len] := 0 (BYTE)
    WriteText(in,out,PRINTER,STR,ecode)
    WriteText(in,out,PRINTER,"       ",ecode)
    REAL32TOSTRING(len,STR,yav,3,3)
    STR[len] := 0 (BYTE)
    WriteTextLine(in,out,PRINTER,STR,ecode)
    WriteTextLine(in,out,PRINTER,"",ecode)
    WriteTextLine(in,out,PRINTER,"",ecode)

    WriteTextLine(in,out,PRINTER,"DRIFT SIZE FOR CELLS",ecode)
    WriteTextLine(in,out,PRINTER,"----------------",ecode)
    WriteTextLine(in,out,PRINTER,"",ecode)

    SEQ i = 0 FOR 10
      SEQ
        SEQ j = 0 FOR 10
          INT len:
          REAL32 num1, num2, mid1, mid2:
          SEQ
            mid1 := (H * ((REAL32 TRUNC j) + 0.5 (REAL32)))
            mid2 := (H * ((REAL32 TRUNC (9-i)) + 0.5 (REAL32)))
            num1 := (total2[j][9-i][0] / (REAL32 ROUND looptotal)) - mid1
            num2 := (total2[j][9-i][1] / (REAL32 ROUND looptotal)) - mid2
            num1 := SQRT((num1*num1) + (num2*num2))
            REAL32TOSTRING(len,STR,num1,2,3)
            STR[len] := 0 (BYTE)
            WriteText(in,out,PRINTER,STR,ecode)
            WriteText(in,out,PRINTER," ",ecode)
        WriteTextLine(in,out,PRINTER,"",ecode)

    xav := SQRT((xav*xav) + (yav*yav))
    WriteTextLine(in,out,PRINTER,"",ecode)
    WriteTextLine(in,out,PRINTER,"",ecode)
    WriteTextLine(in,out,PRINTER,"Average drift size for all cells = ",ecode)
    REAL32TOSTRING(len,STR,xav,3,3)
    STR[len] := 0 (BYTE)
    WriteTextLine(in,out,PRINTER,STR,ecode)
    WriteTextLine(in,out,PRINTER,"",ecode)
TRUE
  VAL H IS 250.0 (REAL32):
  BYTE ch:
  SEQ
    ClrScr(in,out)
    WriteLnString(in,out,"DRIFT FOR CELLS - X, then Y")
    WriteLnString(in,out,"---------------")

    xav, yav := 0.0 (REAL32), 0.0 (REAL32)
```

```
SEQ i = 0 FOR 10
  SEQ
    -- Display X
    SEQ j = 0 FOR 10
      INT len:
      REAL32 mid, num:
      SEQ
        mid := (H * ((REAL32 TRUNC j) + 0.5 (REAL32)))
        num := (total2[j][9-i][0] / (REAL32 ROUND looptotal)) - mid
        xav := xav + num
        REAL32TOSTRING(len,STR,num,2,3)
        STR[len] := 0 (BYTE)
        WriteString(in,out,STR)
        WriteString(in,out," ")

    -- Display Y
    SEQ j = 0 FOR 10
      INT len:
      REAL32 mid, num:
      SEQ
        mid := (H * ((REAL32 TRUNC (9-i)) + 0.5 (REAL32)))
        num := (total2[j][9-i][1] / (REAL32 ROUND looptotal)) - mid
        yav := yav + num
        REAL32TOSTRING(len,STR,num,2,3)
        STR[len] := 0 (BYTE)
        WriteString(in,out,STR)
        WriteString(in,out," ")
    WriteLnString(in,out,"")
    ReadKey(in,out,FALSE,ch)

xav := xav / 100.0 (REAL32)
yav := yav / 100.0 (REAL32)
WriteLnString(in,out,"")
WriteLnString(in,out,"Average X/Y drift for all cells")
REAL32TOSTRING(len,STR,xav,3,3)
STR[len] := 0 (BYTE)
WriteString(in,out,STR)
WriteString(in,out,"       ")
REAL32TOSTRING(len,STR,yav,3,3)
STR[len] := 0 (BYTE)
WriteLnString(in,out,STR)
ReadKey(in,out,FALSE,ch)

ClrScr(in,out)
WriteLnString(in,out,"DRIFT SIZE FOR CELLS")
WriteLnString(in,out,"--------------------")
WriteLnString(in,out," ")

SEQ i = 0 FOR 10
  SEQ
    SEQ j = 0 FOR 10
      INT len:
      REAL32 num1, num2, mid1, mid2:
      SEQ
        mid1 := (H * ((REAL32 TRUNC j) + 0.5 (REAL32)))
        mid2 := (H * ((REAL32 TRUNC (9-i)) + 0.5 (REAL32)))
        num1 := (total2[j][9-i][0] / (REAL32 ROUND looptotal)) - mid1
        num2 := (total2[j][9-i][1] / (REAL32 ROUND looptotal)) - mid2
        num1 := SQRT((num1*num1) + (num2*num2))
        REAL32TOSTRING(len,STR,num1,2,3)
        STR[len] := 0 (BYTE)
        WriteString(in,out,STR)
        WriteString(in,out," ")

xav := SQRT((xav*xav) + (yav*yav))
WriteLnString(in,out,"")
WriteLnString(in,out,"")
WriteString(in,out,"Average drift size for all cells = ")
REAL32TOSTRING(len,STR,xav,3,3)
STR[len] := 0 (BYTE)
WriteLnString(in,out,STR)
```

```
        ReadKey(in,out,FALSE,ch)
    }}}  Check for print screen and print numbers for the drift

    TerminateServer(in,out)
:
```

}}} SC Airflow results display program

```
CHAN OF ANY PC2B, B2PC:
PROCESSOR 0 T4
  PLACE PC2B AT 4:
  PLACE B2PC AT 0:
  T414RESULTS(PC2B, B2PC)
```

}}} **PROGRAM Airflow results display program**

Chapter 6 - Depth First Worm

Depth First Worm Source

This chapter lists the source code of the *Head* and *Tail* of the depth first worm. The worm *Tail* uses the Turbo Pascal Server. The boot code for *Head* must be contained in the file **WORMHEAD.TCD**. The server must use the *Tail* boot code to boot the root transputer which will then commence the search as described in Section 8.4 of the thesis.

The Head - Listing (i)

```
PROC WormHead()
  #USE reinit

  -- Absolute integer value function
  INT FUNCTION abs(VAL INT val)
    INT result:
    VALOF
      IF
        val < 0
          result := (-val)
        TRUE
          result := val
      RESULT result
  :

  -- Hardware link channel placement
  [4]CHAN OF ANY linkin, linkout:
  PLACE linkout AT 0:
  PLACE linkin  AT 4:

  -- Protocols
  VAL endid IS 0:
  VAL BYTE peekbyte     IS 1 (BYTE):
  VAL BYTE bootrequest  IS 4 (BYTE):
  VAL BYTE sendlinkdata IS 8 (BYTE):
  VAL BYTE token        IS 16(BYTE):
  VAL [3]BYTE T2Peek    IS [peekbyte, 128(BYTE), 0(BYTE)]:
  VAL [5]BYTE FirstPeek IS [peekbyte, 128(BYTE), 0(BYTE), 0(BYTE), 0(BYTE)]:
  VAL [2]BYTE RestPeek  IS [0(BYTE), 0(BYTE)]:

  -- Identification constants
  VAL TestConst IS (MOSTNEG INT):
  VAL HostConst IS (TestConst+1):
  VAL T212Const IS (TestConst+2):
  VAL NothConst IS (TestConst+3):
  VAL failConst IS (TestConst+4):

  -- Identification variables for this transputer and its links
  [4][2]INT linkID:
  [4]BOOL linkbooted:
  INT myID, myMemSize, myType, nextID, parentLink:

  -- Run time variables
  BYTE command:

  -- The clock variable and timeout constants
  TIMER clock:
  VAL timeout IS 1563:
```

```
-- The Worm HEAD
SEQ
  -- Initialise all links as unidentified
  SEQ I = 0 FOR 4
    SEQ
      linkID[I][0] := TestConst
      linkbooted[I] := FALSE

  -- Monitor all links for the CID token, accompanied by other identification info.
  ALT I = 0 FOR 4
    linkin[I] ? myID; myMemSize; linkID[I]
      -- Set this link as the link to the parent
      SEQ
        parentLink := I       -- Parent Link number
        linkout[I] ! I        -- Link number to which the parent is connected

  -- The parent link has been identified
  linkID[parentLink][0] := (-linkID[parentLink][0])

  -- Determine what type of transputer this is (T414 or T800) using memory tests.
  INT onchip, maybeoff, offchip:
  PLACE onchip   AT #70:
  PLACE offchip  AT #400:
  PLACE maybeoff AT #300:
  INT temp, start, end, ontime, offtime:
  TIMER clock:
  SEQ
    -- On-Chip time
    clock ? start
    SEQ I = 0 FOR 100000
      temp := onchip
    clock ? end
    ontime := end MINUS start

    -- Off-Chip time
    clock ? start
    SEQ I = 0 FOR 100000
      temp := offchip
    clock ? end
    offtime := end MINUS start

    -- Border time, determing whether this is on or off-chip
    clock ? start
    SEQ I = 0 FOR 100000
      temp := maybeoff
    clock ? end
    temp := end MINUS start

    -- This transputer type is determined by the size of the on-chip RAM.
    start := abs(ontime - temp)
    end   := abs(offtime - temp)
    IF
      start < end
        -- T800
        myType := 8
      TRUE
        -- T414
        myType := 4

  -- Advance the CID token
  nextID := myID PLUS 1

  -- Explore all the unidentified links
  SEQ I = 0 FOR 4
    IF
      linkID[I][0] = TestConst       -- Link must be explored
        INT time:
        BOOL fail:
        SEQ
```

```
-- Attempt 16 bit peek
clock ? time
time := time PLUS timeout
OutputOrFail.t(linkout[I],T2Peek,clock,time,fail)
IF
  fail
    linkID[I][0] := NothConst    -- Transmission failed, nothing there
  TRUE
    [2]BYTE t2val:               -- Transmission succeeded, continue ...
    SEQ
      -- Check if 16 bit transputer exists
      clock ? time
      time := time PLUS timeout
      InputOrFail.t(linkin[I],t2val,clock,time,fail)
      IF
        fail
          -- Not a 16 bit transputer
          SEQ
            -- Complete a 32 bit peek command
            clock ? time
            time := time PLUS timeout
            OutputOrFail.t(linkout[I],RestPeek,clock,time,fail)
            IF
              fail
                -- Failed, nothing there
                linkID[I][0] := failConst
              TRUE
                -- Succeeded, get results of peek
                BOOL boot:
                INT val:
                SEQ
                  linkin[I] ? val   -- Read results
                  boot := FALSE
                  IF
                    val = 123456
                      -- Peek again to confirm already booted
                      SEQ
                        linkout[I] ! peekbyte; TestConst
                        linkin[I] ? val
                        IF
                          val = 654321
                            -- T/P already booted, PROTOCOL matches
                            SKIP
                          TRUE
                            -- Coincidental value, T/P not booted
                            boot := TRUE
                    TRUE
                      -- Not 123456 so T/P not booted
                      boot := TRUE.
                  IF
                    boot
                      -- Determine T/P memsize and boot
                      SEQ
                        INT itsMemSize, size:
                        dadin       IS linkin[parentLink]:
                        dadout      IS linkout[parentLink]:
                        daughterin  IS linkin[I]:
                        daughterout IS linkout[I]:
                        VAL BYTE pokeval IS 0 (BYTE):
                        VAL BYTE peekval IS 1 (BYTE):
                        VAL minint IS MOSTNEG INT:
                        INT last.mem, this.mem, dummy:
                        BOOL loop, error:
                        SEQ
                          -- Determine this T/Ps memory size
                          last.mem, this.mem, loop := 0, 4, TRUE
                          daughterout ! pokeval; (minint+last.mem); last.mem
```

```
  WHILE loop
    INT val:
    SEQ
      daughterout ! pokeval; (minint+this.mem); this.mem
      daughterout ! peekval; (minint+last.mem)
      daughterin ? val
      IF
        val = last.mem
          -- Lap around, continue search
          SEQ
            last.mem := this.mem
            this.mem := this.mem << 1
        TRUE
          -- Lap Around did not occur
          loop := FALSE
  itsMemSize := last.mem

  -- Request boot file from parent
  linkbooted[I] := TRUE
  dadout ! bootrequest

  -- Boot the T/P with boot file
  dadin ? size
  SEQ J = 0 FOR size
    BYTE bootbyte:
    SEQ
      dadin ? bootbyte
      daughterout ! bootbyte

  -- Pass CID to newly booted T/P and
  -- exchange identification info.
  daughterout ! nextID; itsMemSize; myID; I
  linkID[I][0] := nextID
  daughterin ? linkID[I][1]

-- Enter Monitoring Phase
BOOL loop:
BYTE control:
SEQ
  loop := TRUE
  WHILE loop
    ALT J = 0 FOR 4
      linkin[J] ? control
        IF
          control = peekbyte
            -- Emulate the Peek
            INT address:
            SEQ
              -- First peek request
              linkin[J] ? address
              linkout[J] ! 123456

              -- Second Peek request must follow
              linkin[J]  ? control; address
              linkout[J] ! 654321

              -- Exchange identfication info.
              linkin[J] ? linkID[J]
              linkout[J] ! myID; J
          control = bootrequest
            -- Boot file requested, request
            -- from parent and pass on.
            INT size:
            BYTE bootdata:
            dadin  IS linkin[parentLink]:
            dadout IS linkout[parentLink]:
            sonout IS linkout[J]:
            SEQ
              -- Request boot file from parent
              dadout ! bootrequest
```

```
                                        -- Pass boot file to sibling
                                        dadin ? size
                                        sonout ! size
                                        SEQ K = 0 FOR size
                                          SEQ
                                            dadin ? bootdata
                                            sonout ! bootdata
                                      control = token
                                        -- CID returned, terminate monitoring
                                        SEQ  -- (Note: I = J)
                                          loop := FALSE
                                          linkin[J] ? nextID
                            TRUE
                              -- T/P already booted, exchange identification info
                              SEQ
                                linkout[I] ! myID; I
                                linkin[I] ? linkID[I]
                  TRUE
                    -- 16 bit peek succeeded, must be T212
                    linkID[I][0] := T212Const
      TRUE
        -- Link already identified.
        SKIP

-- Return CID token to parent as search is complete here
linkout[parentLink] ! token; nextID

-- Wait for request from parent to pass back the identification info.
linkin[parentLink] ? command
IF
  command = sendlinkdata
    -- Pass back identification info.
    dadout IS linkout[parentLink]:
    SEQ
      -- Send this link data
      dadout ! myID
      SEQ I = 0 FOR 4
        dadout ! linkID[I]
      dadout ! myType; myMemSize

      -- Request data from links which this T/P booted, also forwarding data
      SEQ I = 0 FOR 4
        IF
          linkbooted[I]
            sonout IS linkout[I]:
            sonin  IS linkin[I]:
            BOOL loop:
            INT idthis:
            [2]INT idattached:
            SEQ
              -- Request sibling link identification data
              sonout ! sendlinkdata

              -- Pass back to parent
              loop := TRUE
              WHILE loop
                SEQ
                  sonin ? idthis
                  IF
                    idthis = endid
                      -- No more identification data
                      loop := FALSE
                    TRUE
                      -- More identification data
                      SEQ
                        dadout ! idthis
                        SEQ J = 0 FOR 5
                          SEQ
                            sonin ? idattached
                            dadout ! idattached
```

```
            TRUE
              -- This T/P did not boot this link, so do nothing
              SKIP

        -- Signal parent that identification information is complete
        dadout ! endid

    TRUE          -- Error
      GUY
        SETERR
:

PROCESSOR 1 T4
  WormHead()
```

The Tail - Listing (ii)

```
PROC WormTail()
  #USE reinit
  #USE TPScreen
  #USE TPKeyboard
  #USE TPFile
  #USE subsystem          -- Insert when doing subsystem
  #USE TPServer

  -- Procedure to display an integer value in decmimal on the screen within a field
  PROC WriteInt(CHAN OF ANY in, out, VAL INT int, field)
    #USE ioconv
    #USE TPScreen
    [30]BYTE string:
    INT len, size:
    SEQ
      -- Convert to s string
      INTTOSTRING(len, string, int)
      IF
        (field <= 0) OR (len >= field)
          -- Cut the size short
          size := len
        TRUE
          -- Pad with spaces
          [30]BYTE newstr:
          VAL INT numspaces IS field - len:
          SEQ
            [newstr FROM numspaces FOR len] := [string FROM 0 FOR len]
            string := newstr
            SEQ I = 0 FOR numspaces
              string[I] := ' '
            size := field

      -- Display the string
      WriteString(in,out, [string FROM 0 FOR size])
  :

  -- Procedure to display an integer value in hex on the screen within a field
  PROC WriteHexInt(CHAN OF ANY in, out, VAL INT int, field)
    #USE ioconv
    #USE TPScreen
    [30]BYTE string:
    INT len:
    SEQ
      HEXTOSTRING(len, string, int)
      WriteString(in,out, [string FROM 0 FOR len])
  :

  -- Integer absolute function
  INT FUNCTION abs(VAL INT val)
    INT result:
    VALOF
      IF
        val < 0
          result := (-val)
        TRUE
          result := val
      RESULT result
  :

  -- Hardware link channel placement
  [4]CHAN OF ANY linkin, linkout:
  PLACE linkout AT 0:
  PLACE linkin  AT 4:
```

```
-- Identification constants
VAL endid IS 0:
VAL HostLink IS 0:
VAL TestConst IS (MOSTNEG INT):
VAL HostConst IS (TestConst+1):
VAL T212Const IS (TestConst+2):
VAL NothConst IS (TestConst+3):
VAL failConst IS (TestConst+4):

-- Protocols
VAL BYTE peekbyte     IS 1 (BYTE):
VAL BYTE bootrequest  IS 4 (BYTE):
VAL BYTE sendlinkdata IS 8 (BYTE):
VAL BYTE token        IS 16(BYTE):
VAL [3]BYTE T2Peek IS [peekbyte, 128(BYTE), 0(BYTE)]:
VAL [5]BYTE FirstPeek IS [peekbyte, 128(BYTE), 0(BYTE), 0(BYTE), 0(BYTE)]:
VAL [2]BYTE RestPeek  IS [0(BYTE), 0(BYTE)]:

-- Identification variables and table storage, as well as system variables
INT bootsize, error:
BYTE wormfile, command:
[256][5][2]INT data:
[4][2]INT linkID:
[4]BOOL linkbooted:
INT myID, myType, nextID, numtrans:

-- Timer channel and timeout constant
TIMER clock:
VAL timeout IS 1563:

-- Start of the root code
SEQ
  -- Display the header information
  ClrScr(linkin[HostLink], linkout[HostLink])
  reset.subsystem(linkbooted[0])        -- required when doing subsystem

  WriteString(linkin[HostLink],linkout[HostLink], "TRANSPUTER NETWORK WORM PROGRAM   ")
  WriteLnString(linkin[HostLink],linkout[HostLink], "Written by A.M.Schuilenburg")
  WriteString(linkin[HostLink],linkout[HostLink], "---------------------------------")
  WriteLnString(linkin[HostLink],linkout[HostLink], "--------------------------")
  WriteLnString(linkin[HostLink],linkout[HostLink], "")

  -- Open the HEAD file for reading
  AssignFile(linkin[HostLink],linkout[HostLink], "WormHead.TCD", wormfile)
  ResetFile(linkin[HostLink],linkout[HostLink],wormfile)
  FileError(linkin[HostLink],linkout[HostLink],error)
  IF
    error <> 0
      -- HEAD file not found.  Abort !!!
      SEQ
        WriteLnString(linkin[HostLink],linkout[HostLink], "FILE WORMHEAD.TCD NOT FOUND")
        STOP
    TRUE
      SKIP

  -- Initialise all links as unidentified
  SEQ I = 0 FOR 4
    SEQ
      linkID[I][0] := TestConst
      linkbooted[I] := FALSE

  -- Set Host system to be connected to link 0 (assumption !!)
  linkID[HostLink][0] := HostConst

  -- Notify user that the search has been started
  WriteLnString(linkin[HostLink],linkout[HostLink], "")
  WriteString(linkin[HostLink],linkout[HostLink], "Please wait while wriggling ")

  -- Set the ID, CID and number of transputer variables and tokens.
  myID, nextID, numtrans := 1, 2, 1
```

```
-- Determine the transputer type of this transputer
INT onchip, maybeoff, offchip:
PLACE onchip   AT #70:
PLACE offchip  AT #400:
PLACE maybeoff AT #300:
INT temp, start, end, ontime, offtime:
TIMER clock:
SEQ
  -- Get the on-chip time
  clock ? start
  SEQ I = 0 FOR 100000
    temp := onchip
  clock ? end
  ontime := end MINUS start

  -- Get the off-chip time
  clock ? start
  SEQ I = 0 FOR 100000
    temp := offchip
  clock ? end
  offtime := end MINUS start

  -- Get the border time (either off or on chip)
  clock ? start
  SEQ I = 0 FOR 100000
    temp := maybeoff
  clock ? end
  temp := end MINUS start

  -- This transputer type is determined by the size of on-chip RAM
  start := abs(ontime - temp)
  end   := abs(offtime - temp)
  IF
    start < end
      myType := 8
    TRUE
      myType := 4

-- Indicate to the user that this transputer has been determined
WriteString(linkin[HostLink],linkout[HostLink], ".")

-- Search all unidentified links
SEQ I = 0 FOR 4
  IF
    linkID[I][0] = TestConst  -- Link unidentified, must be explored
      INT time:
      BOOL fail:
      SEQ
        -- Attempt 16 bit peek
        clock ? time
        time := time PLUS timeout
        OutputOrFail.t(linkout[I],T2Peek,clock,time,fail)
        IF
          fail
            linkID[I][0] := NothConst   -- Transmission failed, nothing there
          TRUE
            [2]BYTE t2val:              -- Transmission succeeded, continue ...
            SEQ
              -- Check if 16 bit transputer exists
              clock ? time
              time := time PLUS timeout
              InputOrFail.t(linkin[I],t2val,clock,time,fail)
              IF
                fail
                  -- Not a 16 bit transputer
                  SEQ
                    -- Complete 32 bit peek command
                    clock ? time
                    time := time PLUS timeout
                    OutputOrFail.t(linkout[I],RestPeek,clock,time,fail)
```

```
IF
  fail
    -- Failed, nothing there
    linkID[I][0] := failConst
  TRUE
    -- Succeeded, get results of peek
    BOOL boot:
    SEQ
      INT val:
      SEQ
        linkin[I] ? val     -- Read results
        boot := FALSE
        IF
          val = 123456
            -- Peek again to confirm already booted
            SEQ
              linkout[I] ! peekbyte; TestConst     -- Peek again
              linkin[I] ? val
              IF
                val = 654321
                -- T/P already booted, PROTOCOL matches
                  SKIP
                TRUE
                  -- Coincidental value, T/P not booted
                  boot := TRUE
          TRUE
            -- Not 123456 so T/P not booted
            boot := TRUE
      IF
        boot
          -- Determine T/P memsize and boot
          SEQ
            INT itsMemSize, size:
            dadin       IS linkin[HostLink]:
            dadout      IS linkout[HostLink]:
            daughterin  IS linkin[I]:
            daughterout IS linkout[I]:
            BYTE bootbyte:
            BOOL loop:
            VAL BYTE pokeval IS 0 (BYTE):
            VAL BYTE peekval IS 1 (BYTE):
            VAL minint IS MOSTNEG INT:
            INT last.mem, this.mem, dummy:
            BOOL loop, error:
            SEQ
              -- Determine this T/Ps memory size
              last.mem, this.mem, loop := 0, 4, TRUE
              daughterout ! pokeval; (minint+last.mem); last.mem
              WHILE loop
                INT val:
                SEQ
                  daughterout ! pokeval; (minint+this.mem); this.mem
                  daughterout ! peekval; (minint+last.mem)
                  daughterin ? val
                  IF
                    val = last.mem
                      SEQ
                        last.mem := this.mem
                        this.mem := this.mem << 1
                    TRUE
                      loop := FALSE
              itsMemSize := last.mem
```

```
  -- Read the boot file from the HOST and
  -- use it to boot this T/P.
  bootsize, linkbooted[I] := 0, TRUE
  loop := TRUE
  WHILE loop
    SEQ
      ReadByte(dadin,dadout,wormfile,bootbyte)
      daughterout ! bootbyte
      bootsize := bootsize + 1
      EndOfFile(dadin,dadout,wormfile,loop)
      loop := NOT loop

  -- Pass CID to newly booted T/P and
  -- exchange identification info.
  daughterout ! nextID; itsMemSize; myID; I
  linkID[I][0] := nextID
  daughterin ? linkID[I][1]            -- Booted T/Ps boot link

  -- Another T/P discovered
  numtrans := numtrans + 1
  WriteString(dadin,dadout, ".")

  -- Reset HEAD file for re-use
  CloseFile(dadin,dadout,wormfile)
  AssignFile(dadin,dadout, "WormHead.TCD", wormfile)
  ResetFile(dadin,dadout,wormfile)
  FileError(dadin,dadout,error)
  IF
    error <> 0
      INT t:
      SEQ
        t := INT error
        WriteInt(dadin, dadout, t, 0)
        WriteString(dadin,dadout, " TERMINAL RESET ERROR --FILE")
        WriteLnString(dadin,dadout, " WORMHEAD.TCD NOT FOUND")
        STOP
    TRUE
      SKIP

-- Enter Monitoring Phase
BOOL loop:
BYTE control:
SEQ
  loop := TRUE
  WHILE loop
    ALT J = 0 FOR 4
      linkin[J] ? control
        IF
          control = peekbyte
            -- Emulate the peek
            INT address:
            SEQ
              -- First peek request
              linkin[J] ? address
              linkout[J] ! 123456

              -- Second Peek must follow
              linkin[J]  ? control; address
              linkout[J] ! 654321

              -- Exchange identification info.
              linkin[J] ? linkID[J]
              linkout[J] ! myID; J
```

```
                                    control = bootrequest
                                      -- Boot file requested, read from
                                      -- PC host and pass on.
                                      BYTE bootdata:
                                      dadin  IS linkin[HostLink]:
                                      dadout IS linkout[HostLink]:
                                      sonout IS linkout[J]:
                                      SEQ
                                        sonout ! bootsize
                                        SEQ M = 0 FOR bootsize
                                          SEQ
                                            ReadByte(dadin,dadout,wormfile,bootdata)
                                            sonout ! bootdata

                                        -- Another T/P booted
                                        numtrans := numtrans + 1
                                        WriteString(dadin,dadout, ".")

                                        -- Open HEAD file for re-use
                                        CloseFile(dadin,dadout,wormfile)
                                        AssignFile(dadin,dadout, "WormHead.TCD", wormfile)
                                        ResetFile(dadin,dadout,wormfile)
                                        FileError(dadin,dadout,error)
                                        IF
                                          error <> 0
                                            INT t:
                                            SEQ
                                              t := INT error
                                              WriteInt(dadin, dadout, t, 0)
                                              WriteString(dadin,dadout, " TERMINAL RESET ERROR")
                                              WriteString(dadin,dadout, "  --FILE WORMHEAD.TCD")
                                              WriteLnString(dadin,dadout, "  NOT FOUND")
                                              STOP
                                          TRUE
                                            SKIP
                                    control = token
                                      -- CID returned, terminate monitoring
                                      SEQ  -- (Note: I = J)
                                        loop := FALSE
                                        linkin[J] ? nextID
                          TRUE
                            -- T/P already booted, exchange identification info.
                            SEQ
                              linkout[I] ! myID; I
                              linkin[I] ? linkID[I]
                TRUE
                  -- 16 bit peek succeeded; must be T212
                  linkID[I][0] := T212Const   -- T212
      TRUE
        -- Link already identified.
        SKIP

-- All links are now identified,  Search is complete
in  IS linkin[HostLink]:
out IS linkout[HostLink]:
SEQ
  -- File no longer required
  CloseFile(in,out,wormfile)

  -- Display header information
  GotoXY(in,out, 1, 4)
  ClrEOS(in,out)
  WriteInt(in,out, numtrans, 0)
  WriteString(in,out, " Transputers found.")
  GotoXY(in,out, 1, 6)
  WriteLnString(in,out, "Link configuration is                ID-Link#        ** = Booted by")
  WriteLnString(in,out, "T/P ID    Type       Memory        Link 0   Link 1   Link 2   Link 3")
```

```
SEQ
  -- Store this link data
  [data[0] FROM 0 FOR 4] := linkID
  data[0][4][0] := myType
  SystemInfo(in,out, data[0][4][1])

  -- Request data from links which this T/P booted and also forward data
  SEQ I = 0 FOR 4
    IF
      linkbooted[I]
        sonout IS linkout[I]:
        sonin  IS linkin[I]:
        BOOL loop:
        INT idthis:
        SEQ
          sonout ! sendlinkdata   -- Request link data
          loop := TRUE
          WHILE loop
            SEQ
              sonin ? idthis      -- Data ?
              IF
                idthis = endid    -- Data complete from this link
                  loop := FALSE
                TRUE              -- More data from this link
                  adddata IS data[idthis-myID]:
                  SEQ J = 0 FOR 5
                    sonin ? adddata[J]
      TRUE
        -- Not a parent to this link
        SKIP

-- Display the results
SEQ I = myID FOR numtrans
  thisdata IS data[I-myID]:
  SEQ
    -- Transputer Number
    WriteInt(in,out, I, 3)
    WriteString(in,out, ")      ")

    -- Transputer Type
    IF
      thisdata[4][0] = 4
        WriteString(in,out,"T414   ")
      thisdata[4][0] = 8
        WriteString(in,out,"T800   ")
      TRUE
        WriteString(in,out,"????   ")

    -- Memory size
    IF
      thisdata[4][1] = 0
        WriteString(in,out, "    Unknown       ")
      TRUE
        SEQ
          WriteString(in,out, "   #")
          WriteHexInt(in,out, thisdata[4][1], 8)
          WriteString(in,out, "     ")

    -- Link connections
    SEQ J = 0 FOR 4
      SEQ
        IF
          thisdata[J][0] = HostConst
            WriteString(in,out, "  HOST**")
          thisdata[J][0] = T212Const
            WriteString(in,out, "  T212 ")
          thisdata[J][0] = NothConst
            WriteString(in,out, "  ---- ")
          thisdata[J][0] = failConst
            WriteString(in,out, "  fail ")
```

```
            TRUE
              SEQ
                WriteInt(in,out, abs(thisdata[J][0]),4)
                WriteString(in,out, "-")
                WriteInt(in,out, thisdata[J][1],1)
                IF
                  thisdata[J][0] < 0
                    -- Was booted by this link
                    WriteString(in,out,"***")
                  TRUE
                    WriteString(in,out," ")
          WriteString(in,out, "  ")
      WriteLnString(in,out, "")

    -- File the data ?
    BYTE ch:
    SEQ
      WriteLnString(in,out,"*C*N*C*N")
      WriteString(in,out,"File Data Y/N (Default N) ? ")
      ReadKey(in,out,TRUE,ch)
      IF
        (ch = 'Y') OR (ch = 'y')
          INT len:
          [30]BYTE fname:
          SEQ
            WriteString(in,out,"*C*NEnter the file name ?")
            ReadString(in,out,TRUE,len,fname)
            IF
              len = 0
                SKIP
              TRUE
                BYTE file:
                INT ecode, written:
                []BYTE byteblock RETYPES [data FROM 0 FOR numtrans]:
                SEQ
                  AssignFile(in,out,[fname FROM 0 FOR len],file)
                  RewriteFile(in,out,file)
                  -- WriteTextLine(in,out,PRINTER,[fname FROM 0 FOR len])
                  WriteBlock(in,out,file,SIZE byteblock,byteblock,written)
                  CloseFile(in,out,file)
                  FileError(in,out,ecode)
                  IF
                    (ecode <> 0) OR (written <> (SIZE byteblock))
                      WriteLnString(in,out,"*C*NError in writing file")
                    TRUE
                      WriteLnString(in,out,"*C*NFile written O.K.")
        TRUE
          SKIP

  -- Allow Pascal Server to quit to dos
  TerminateServer(linkin[HostLink],linkout[HostLink])

:
PROCESSOR 1 T4
  WormTail()
```

Chapter 7 - Breadth First Worm

Breadth First Worm Source

This chapter lists the source code of the *Mouth* and *Foot* of the breadth first worm. The worm *Foot* uses the Turbo Pascal Server. The boot code for *Head* must be contained in the file **MOUTH.BTL**. The server must use the *Foot* boot code to boot the root transputer which will then commence the search as described in Section 8.5 of the thesis. Furthermore, the *Foot* boot file must be appended with the 32 bit integer token which is used as the *CID* token. This allows the *Foot* to determine the actual link connected to the Host PC, instead of assuming the default as the Depth First Worm does. The default is to use a *CID* of 0. The source of the link identification prcedure, common to both processes, is given seperately in Listing 7(i), with the source of *Mouth* and *Foot* provided in Listings 7(ii) and 7(iii) respectively.

Identify Link Procedure - Listing (i)

```
PROC identify.link(CHAN OF ANY linkout, linkin,
                   VAL INT myID, myLink,
                   INT type.or.id, mem.or.link)
  #USE "XLINK.LIB"
  #INCLUDE "CONSTS.INC"

  -- Constants
  VAL pokeval           IS 0 (BYTE):
  VAL peekval           IS 1 (BYTE):
  VAL [5]BYTE Peek IS [peekval, 128(BYTE), 0(BYTE), 0(BYTE), 0(BYTE)]:

  -- Variables
  INT time:
  BOOL fail, booted:
  TIMER clock:

  SEQ                          -- Search down this link
    booted := FALSE    -- Initialise

    -- Attempt 16 bit peek
    clock ? time
    time := time PLUS timeout
    OutputOrFail.t(linkout,[Peek FROM 0 FOR 3],clock,time,fail)
    IF
      fail
        SKIP   -- Nothing there !
      TRUE
        SEQ    -- Something there !
          [2]BYTE t2val:
          SEQ
            -- Attempt to get results of 16 bit peek
            clock ? time
            time := time PLUS timeout
            InputOrFail.t(linkin,t2val,clock,time,fail)
            IF
              fail
                SEQ              -- Check for 32 bit transputer
                  -- Complete peek to be a 32 bit peek
                  clock ? time
                  time := time PLUS timeout
```

```
              OutputOrFail.t(linkout,[Peek FROM 3 FOR 2],clock,time,fail)
              IF
                fail
                  type.or.id := failed     -- Something there (Unknown)
                TRUE
                  [4]BYTE valb1, valb2:
                  SEQ                      -- Check for 32 bit
                    -- Attempt to get results of peek
                    clock ? time
                    time := time PLUS timeout
                    InputOrFail.t(linkin,valb1,clock,time,fail)
                    INT val1 RETYPES valb1:
                    IF
                      fail
                        type.or.id := failed   -- Something there (Unknown)
                      val1 = testval1
                        SEQ  -- Peek again, ensure is really booted
                          -- 2nd Peek Request
                          clock ? time
                          time := time PLUS timeout
                          OutputOrFail.t(linkout,Peek,clock,time,fail)

                          -- Get results of peek
                          clock ? time
                          time := time PLUS timeout
                          InputOrFail.t(linkin,valb2,clock,time,fail)
                          type.or.id := Something
                          VAL INT val2 RETYPES valb2:
                          IF
                            val2 = testval2 -- T/P already booted, identify
                              booted := TRUE
                            TRUE           -- Coincidental, T/P not booted
                              SKIP
                      TRUE
                        -- 32 bit Transputer Found
                        type.or.id := Something
          TRUE
            -- 16 bit peek succeeded, T212 found.
            type.or.id := T.212

IF
  booted
    --  Get the booted transputers id, and its link number
    SEQ
      linkout ! req.id; myID; myLink
      linkin ? type.or.id; mem.or.link
  (type.or.id = Something)
    --  Get the transputer memory size.
    VAL minint IS MOSTNEG INT:
    BOOL found:
    INT mem:
    SEQ
      -- Check special memory location to see if already found
      found, mem := TRUE, #70
      WHILE found AND (mem < #80)
        INT val:
        SEQ
          linkout ! peekval; (minint + (mem << 2))
          linkin ? val
          IF
            val=mem
              mem := mem + 1
            TRUE
              found := FALSE
```

```
          IF
            -- Already been found, will be booted by another T/P
            found
              type.or.id := AlreadyFound
            -- Not yet found, must get details and boot
            TRUE
              -- Get the memory size
              INT this.mem, val:
              BOOL loop:
              SEQ
                -- Determine the memory size
                mem.or.link, this.mem, loop := 0, 4, TRUE
                linkout ! pokeval; (minint+mem.or.link); mem.or.link
                WHILE loop
                  SEQ
                    linkout ! pokeval; (minint+this.mem); this.mem
                    linkout ! peekval; (minint+mem.or.link)
                    linkin ? val
                    IF
                      val = mem.or.link
                        -- Lap around, continue trying
                        SEQ
                          mem.or.link := this.mem
                          this.mem := this.mem << 1
                      TRUE
                        -- No lap around, size is found
                        loop := FALSE

                -- Mark as being found
                SEQ I = #70 FOR #10
                  linkout ! pokeval; (minint + (I << 2)); I
      TRUE
        SKIP
:
```

The Mouth - Listing (ii)

```
PROC Mouth()
  #USE "idlink"
  #INCLUDE "CONSTS.INC"

  -- Absolute Integer function
  INT FUNCTION abs(VAL INT value)
    INT result:
    VALOF
      IF
        (value < 0)
          result := (-value)
        TRUE
          result := value
      RESULT result
  :

  -- Hardware link placement
  [4]CHAN OF ANY linkin, linkout:
  PLACE linkout AT 0:
  PLACE linkin  AT 4:

  -- Identification and sustem variables
  [4][2]INT linkID:
  [4]BOOL linkBoot, getInfo, bootSize:

  INT myID, myType, myMem, bootLink:
  BOOL needboot:

  -- Procedure to monitor all the links
  PROC Monitor.Links(BYTE command, BOOL needboot)
    INT address:
    BOOL repeat:
    SEQ
      repeat := TRUE
      WHILE repeat
        ALT I = 0 FOR 4
          linkin[I] ? command
            IF
              -- Respond as a booted transputer using the booted protocol
              (command = peekval)
                SEQ  -- Respond as a booted transputer
                  -- Return the firs special value (123456)
                  linkin[I] ? address
                  linkout[I] ! testval1

                  -- Return the second special value (654321)
                  linkin[I] ? command; address
                  linkout[I] ! testval2
              -- Give the ID and link number connection, exchanging identification info.
              (command = req.id)
                SEQ
                  linkin[I] ? linkID[I][0]; linkID[I][1]
                  linkout[I] ! myID; I
              -- Cease monitoring and give id data
              (command = req.link.data)
                repeat := FALSE
              -- The explore token was returned
              (command = explore.return)
                SEQ
                  linkin[I] ? needboot
                  repeat := FALSE
              -- Cease monitoring and continue exploration
              (command = explore)
                repeat := FALSE
  :
```

```
SEQ
  -- Initialise all channels as unidentified
  SEQ I = 0 FOR 4
    SEQ
      linkID[I] := [Nothing,Nothing]
      linkBoot[I] := FALSE
      getInfo[I] := FALSE
      bootSize[I] := FALSE
  needboot := FALSE

  -- Monitor all the links for the ID
  ALT I = 0 FOR 4
    linkin[I] ? myID
      bootLink := I

  -- Request my memory size, returning CID token
  linkout[bootLink] ! req.mem; (myID + 1)
  linkin[bootLink] ? myMem

  -- Set my type
  INT read, onchip, maybeoff, offchip, start, timeon, timeoff, testtime:
  PLACE onchip   AT #70:
  PLACE maybeoff AT #300:
  PLACE offchip  AT #400:
  VAL INT loop IS 10000:
  TIMER clock:
  SEQ
    -- Get on-chip time
    clock ? start
    SEQ I = 0 FOR loop
      read := onchip
    clock ? timeon
    timeon := abs(timeon - start)        --  Time for on-chip access

    -- Get off-chip time
    clock ? start
    SEQ I = 0 FOR loop
      read := offchip
    clock ? timeoff
    timeoff := abs(timeoff - start)      --  Time for off-chip access

    -- Get border time (Either on or off chip)
    clock ? start
    SEQ I = 0 FOR loop
      read := maybeoff
    clock ? testtime
    testtime := abs(testtime - start)    --  Test time access

    -- The type depends on whether 4k (T800) or 2k (T414) onchip RAM
    IF
      abs(timeon - testtime) < abs(timeoff - testtime)
        myType := T.800
      TRUE
        myType := T.414

  -- Monitor all links, transferring ID's
  BYTE command:
  BOOL dummy:
  SEQ
    Monitor.Links(command,dummy)
    -- command must be explore from bootLink
```

```
--  Identify remaining links
SEQ I = 0 FOR 4
  IF
    (linkID[I][0] = Nothing)
      SEQ
        identify.link(linkout[I],linkin[I],myID,I,
                      linkID[I][0],linkID[I][1]   )
        IF
          (linkID[I][0] >= Something) AND (linkID[I][0] <> T.212)
            SEQ                       -- Transputer is there and needs booting
              linkBoot[I] := TRUE     -- Boot this link
              getInfo[I] := TRUE      -- Get information once complete
              needboot := TRUE        -- Boot file is needed from bootlink
              bootSize[I] := FALSE    -- Do not give the boot file size
          TRUE
            SKIP
    TRUE
      SKIP

-- Return the token as the boot file request
linkout[bootLink] ! explore.return; needboot

-- Now for the worm body
WHILE needboot
  SEQ
    -- Pass down the boot file to T/Ps to be booted
    INT bootsize:
    BYTE bootbyte:
    SEQ
      -- Get the size and forward to necessary links
      linkin[bootLink] ? bootsize
      SEQ I=0 FOR 4
        IF
          linkBoot[I] AND bootSize[I]
            linkout[I] ! bootsize
          TRUE
            bootSize[I] := TRUE

      -- Now actually transmit the worm mouth to necessary links
      SEQ I=0 FOR bootsize
        SEQ
          linkin[bootLink] ? bootbyte
          SEQ J=0 FOR 4
            IF
              linkBoot[J]
                linkout[J] ! bootbyte
              TRUE
                SKIP

    -- Get CID token and forward down each links
    INT current.id:
    SEQ
      linkin[bootLink] ? current.id        -- Get the current ID
      SEQ I = 0 FOR 4
        IF
          linkBoot[I]
            BYTE command:
            SEQ
              linkout[I] ! current.id          -- Forward CID token
              linkin[I] ? command; current.id  -- Wait for CID return
              IF
                (command = id.return)
                  SKIP
                (command = req.mem)
                  SEQ  -- Give booted transputer its memory size
                    linkout[I] ! linkID[I][1]

                    -- Request booted T/P id and link info
                    linkout[I] ! req.id; myID; I
                    linkin[I] ? linkID[I][0]; linkID[I][1]
```

```
          TRUE
            SKIP
        linkout[bootLink] ! id.return; current.id -- Send back the current ID

      -- Wait for command from from boot link, but monitor in the mean time
      BYTE command:
      BOOL dummy:
      SEQ
        Monitor.Links(command,dummy)    -- command must be explore from bootLink

      -- Got token from boot link, continue exploration
      needboot := FALSE
      SEQ I = 0 FOR 4
        IF
          linkBoot[I]
            BYTE command:
            SEQ
              linkout[I] ! explore                -- Pass on token
              Monitor.Links(command,linkBoot[I])  -- Get back token
              IF    -- command must be explore.return from I
                linkBoot[I]
                  needboot := TRUE
                TRUE
                  SKIP
          TRUE
            SKIP
      linkout[bootLink] ! explore.return; needboot -- Return token to boot link

  -- Monitor links again, this time wait for the req.link.data command
  BYTE command:
  BOOL dummy:
  SEQ
    Monitor.Links(command,dummy)
    -- Command must be req.link.data from bootLink
    SEQ I=0 FOR 4          -- Forward to other users
      IF
        getInfo[I]
          linkout[I] ! req.link.data
        TRUE
          SKIP

  -- Continually pass link information from other transputers down
  needboot := TRUE
  INT tmpID, tmpType, tmpMem, tmpBL:
  [4][2]INT tmplinkID:
  BYTE command:
  WHILE needboot
    SEQ
      needboot := FALSE                    -- Reset, will be set again
      SEQ I = 0 FOR 4                      -- if more info is to be sent.
        IF
          getInfo[I]
            SEQ
              -- Get info and forward
              linkin[I] ? command; tmpID; tmpType; tmpMem; tmpBL; tmplinkID
              linkout[bootLink] ! tp.info; tmpID; tmpType; tmpMem; tmpBL; tmplinkID

              -- Check if it is the last
              IF
                (command = tp.info)
                  needboot := TRUE         -- More info to come
                (command = last.tp.info)
                  getInfo[I] := FALSE      -- No more info from this link
          TRUE
            SKIP

  -- Pass back this T/P information down boot link
  linkout[bootLink] ! last.tp.info; myID; myType; myMem; bootLink; linkID

  -- WORM WIGGLING COMPLETE FOR THIS T/P
:
```

The Foot - Listing (iii)

```
PROC WormFoot()
  #USE "xlink.lib"
  #USE "convert.lib"
  #USE "idlink"
  #USE "TPScreen"
  #USE "TPDos"
  #USE "TPKeybrd"
  #USE "TPFile"
  #USE "TPServer"
  #USE "TPSubSys"

  #INCLUDE "CONSTS.INC"

  -- Procedure to display an integer value in decmimal on the screen within a field
  PROC WriteInt(CHAN OF ANY in, out, VAL INT int, field)
    [30]BYTE string:
    INT len, size:
    SEQ
      -- Get the string
      INTTOSTRING(len, string, int)
      IF
        (field <= 0) OR (len >= field)
          String too long, cut off
          size := len
        TRUE
          -- Pad string with spaces
          [30]BYTE newstr:
          VAL INT numspaces IS field - len:
          SEQ
            [newstr FROM numspaces FOR len] := [string FROM 0 FOR len]
            string := newstr
            SEQ I = 0 FOR numspaces
              string[I] := ' '
            size := field

      -- Display the string
      WriteString(in,out, [string FROM 0 FOR size])
  :

  -- Procedure to display an integer value in hex on the screen within a field
  PROC WriteHexInt(CHAN OF ANY in, out, VAL INT int, field)
    [30]BYTE string:
    INT len:
    SEQ
      HEXTOSTRING(len, string, int)
      WriteString(in,out, [string FROM 0 FOR len])
  :

  -- This procedure will display the information relative to a transputer
  -- on a single line.
  PROC Display.Info(CHAN OF ANY in, out, VAL INT id, type, mem, BL,
                                         VAL [4][2]INT links    )
    SEQ
      -- The transputer ID
      WriteInt(in,out, id, 3)
      WriteString(in,out, ")      ")

      -- The Transputer Type
      IF
        type = T.414
          WriteString(in,out,"T414   ")
        type = T.800
          WriteString(in,out,"T800   ")
        type = T.212
          WriteString(in,out,"T212   ")
        TRUE
          WriteString(in,out,"????   ")
```

```
    -- The memory Size
    WriteString(in,out, "   #")
    WriteHexInt(in,out, mem, 8)
    WriteString(in,out, "     ")

    -- The link connections
    SEQ J = 0 FOR 4
      SEQ
        CASE links[J][0]
          Host
            WriteString(in,out, "  HOST")
          T.212
            WriteString(in,out, "  T212")
          Nothing
            WriteString(in,out, "  ----")
          Something
            WriteString(in,out, "  ????")
          failed
            WriteString(in,out, "  fail")
          ELSE
            SEQ
              WriteInt(in,out,links[J][0],4)
              WriteString(in,out, "-")
              WriteInt(in,out,links[J][1],1)

        -- Display ** if transputer was booted from this link
        IF
          J=BL
            WriteString(in,out, "**  ")
          TRUE
            WriteString(in,out, "   ")
    WriteLnString(in,out, "")
:

-- Integer abslute function
INT FUNCTION abs(VAL INT val)
  INT result:
  VALOF
    IF
      val < 0
        result := (-val)
      TRUE
        result := val
    RESULT result
:

-- Hardware link placement
[4]CHAN OF ANY linkin, linkout:
PLACE linkout AT 0:
PLACE linkin  AT 4:

-- Identification and system variables
INT myID, myType, myMem, current.id, BL, error:
BOOL needboot:
BYTE wormfile:

[4][2]INT linkID:
[4]BOOL linkBoot, getInfo, bootSize:
```

```
-- Procedure to monitor all links
PROC Monitor.Links(BYTE command, BOOL needboot)
  INT address:
  BOOL repeat:
  SEQ
    repeat := TRUE
    WHILE repeat
      ALT I = 0 FOR 4
        linkin[I] ? command
          IF
            -- Respond as a booted transputer using the booted protocol
            (command = peekval)
              SEQ  -- Respond as a booted transputer
                -- Return the first special value (123456)
                linkin[I] ? address
                linkout[I] ! testval1

                -- The second peek, return (654321)
                linkin[I] ? command; address
                linkout[I] ! testval2
            -- Give the ID and link number connection, exchanging identification info.
            (command = req.id)
              SEQ
                linkin[I] ? linkID[I][0]; linkID[I][1]
                linkout[I] ! myID; I
            -- Cease monitoring and give id data
            (command = req.link.data)
              repeat := FALSE
            -- The explore token was returned
            (command = explore.return)
              SEQ
                -- Get indicator whether this link requires boot file
                linkin[I] ? needboot
                repeat := FALSE      -- Cease monitoring
            -- Cease monitoring and commence explore
            (command = explore)
              repeat := FALSE
:

TIMER clock:
SEQ
  -- Initialise all the channels as unidentified
  SEQ I = 0 FOR 4
    SEQ
      linkID[I] := [Nothing,Nothing]
      linkBoot[I] := FALSE
      getInfo[I] := FALSE
      bootSize[I] := FALSE

  -- Initialise identification variables
  needboot := FALSE
  myType, myMem := Nothing, 0

  -- Monitor all links to get response from Host along link to Host
  -- Boot file appended with four byte 0 values as ID.
  ALT I = 0 FOR 4
    linkin[I] ? myID
      BL := I       -- BL is Boot Link

  -- Set the boot link ID
  linkID[BL] := [Host,0]

  -- Advance CID token
  current.id := myID + 1

  -- Initialise the screen and do the headings
  ClrScr(linkin[BL], linkout[BL])
  BOOL flag:
  reset.subsystem(flag)                  -- required when doing subsystem

  WriteString(linkin[BL],linkout[BL], "TRANSPUTER NETWORK WORM PROGRAM V2.0")
```

```
WriteLnString(linkin[BL],linkout[BL], "   Written by A.M.Schuilenburg")
WriteString(linkin[BL],linkout[BL], "-----------------------------------")
WriteLnString(linkin[BL],linkout[BL], "-----------------------------")
WriteLnString(linkin[BL],linkout[BL], "")

-- Ready the boot file for reading
AssignFile(linkin[BL],linkout[BL], "MOUTH.BTL", wormfile)
ResetFile(linkin[BL],linkout[BL],wormfile)
FileError(linkin[BL],linkout[BL],error)
IF
  error <> 0
    SEQ
      WriteLnString(linkin[BL],linkout[BL], "FILE MOUTH.BTL NOT FOUND")
      TerminateServer(linkin[BL],linkout[BL])
  TRUE
    CloseFile(linkin[BL],linkout[BL],wormfile)

-- Notify discovery process
WriteString(linkin[BL],linkout[BL],"Exploring ")

-- Set my type
INT read, onchip, maybeoff, offchip, start, timeon, timeoff, testtime:
PLACE onchip   AT #70:
PLACE maybeoff AT #300:
PLACE offchip  AT #400:
VAL INT loop IS 10000:
TIMER clock:
SEQ
  -- Get on-chip time
  clock ? start
  SEQ I = 0 FOR loop
    read := onchip
  clock ? timeon
  timeon := abs(timeon - start)        --  Time for on-chip access

  -- Get off-chip time
  clock ? start
  SEQ I = 0 FOR loop
    read := offchip
  clock ? timeoff
  timeoff := abs(timeoff - start)      --  Time for off-chip access

  -- Get border time (Is either on or off-chip)
  clock ? start
  SEQ I = 0 FOR loop
    read := maybeoff
  clock ? testtime
  testtime := abs(testtime - start)    --  Test time access

  -- The type depends on whether 4k (T800) or 2k (T414) onchip RAM
  IF
    abs(timeon - testtime) < abs(timeoff - testtime)
      myType := T.800
    TRUE
      myType := T.414

-- Request my memory size, determined by HOST
SystemInfo(linkin[BL],linkout[BL], myMem)

--  The tail boots the 1st T/P and has the initial token,
--  therefore implying that no link monitoring is required.
--  Now identify remaining links
SEQ I = 0 FOR 4
  IF
    (linkID[I][0] = Nothing)
      -- Link not yet determined, explore ....
      SEQ
        WriteString(linkin[BL],linkout[BL],"?")
        identify.link(linkout[I],linkin[I],myID,I,
                      linkID[I][0],linkID[I][1]   )
        IF
```

```
          (linkID[I][0] >= Something) AND (linkID[I][0] <> T.212)
            SEQ                     -- Transputer is there and needs booting
              linkBoot[I] := TRUE   -- Boot this link
              getInfo[I] := TRUE    -- Get information once complete
              needboot := TRUE      -- Boot file is needed from bootlink
              bootSize[I] := FALSE -- Do not send the boot size
          TRUE
            SKIP
    TRUE
      SKIP

-- Now for the sending of the worm body
WHILE needboot
  SEQ
    -- Pass down the boot file to T/Ps to be booted
    INT bootsize:
    BYTE bootbyte:
    SEQ
      WriteString(linkin[BL],linkout[BL],".")

      -- Send the boot file down each link requiring it
      AssignFile(linkin[BL],linkout[BL], "MOUTH.BTL", wormfile)
      ResetFile(linkin[BL],linkout[BL],wormfile)
      FileSize(linkin[BL],linkout[BL],wormfile,bootsize)
      SEQ I=0 FOR 4
        IF
          linkBoot[I] AND bootSize[I]
            linkout[I] ! bootsize
          TRUE
            bootSize[I] := TRUE

      SEQ I=0 FOR bootsize
        SEQ
          ReadByte(linkin[BL],linkout[BL], wormfile, bootbyte)
          SEQ J=0 FOR 4
            IF
              linkBoot[J]
                linkout[J] ! bootbyte
              TRUE
                SKIP
      CloseFile(linkin[BL],linkout[BL],wormfile)
    WriteString(linkin[BL],linkout[BL],"**")    -- Notify send complete

    -- Forward CID token down each link
    SEQ I = 0 FOR 4
      IF
        linkBoot[I]
          BYTE command:
          SEQ
            linkout[I] ! current.id  -- Forward CID token
            linkin[I] ? command; current.id  -- Wait for return of CID
            IF
              (command = id.return)
                SKIP
              (command = req.mem)
                SEQ  -- Give booted transputer its memory size
                  linkout[I] ! linkID[I][1]

                  -- Request booted T/P id and link info
                  linkout[I] ! req.id; myID; I
                  linkin[I] ? linkID[I][0]; linkID[I][1]
            -- Give number T/Ps found
            WriteString(linkin[BL],linkout[BL],"#")
            WriteInt(linkin[BL],linkout[BL],current.id-1,0)
        TRUE
          SKIP
```

```
      -- Send token down each link
      BYTE command:
      SEQ
        needboot := FALSE
        SEQ I = 0 FOR 4
          IF
            linkBoot[I]
              BYTE command:
              SEQ
                WriteString(linkin[BL],linkout[BL],"?")
                linkout[I] ! explore              -- Pass on token
                Monitor.Links(command,linkBoot[I]) -- Get back token
                IF                -- command must be explore.return
                  linkBoot[I]
                    needboot := TRUE
                  TRUE
                    SKIP
            TRUE
              SKIP

    -- Request the link information
    SEQ I=0 FOR 4
      IF
        getInfo[I]
          linkout[I] ! req.link.data
        TRUE
          SKIP

    -- Display the heading
    GotoXY(linkin[BL],linkout[BL], 1, 5)
    ClrEOS(linkin[BL],linkout[BL])
    WriteInt(linkin[BL],linkout[BL], current.id, 0)
    WriteString(linkin[BL],linkout[BL], " Transputers found.")
    GotoXY(linkin[BL],linkout[BL], 1, 6)
    WriteString(linkin[BL],linkout[BL], "Link configuration is")
    WriteLnString(linkin[BL],linkout[BL], "                ID-Link#        ** = Booted by")
    WriteString(linkin[BL],linkout[BL], "T/P ID    Type       Memory        Link 0")
    WriteLnString(linkin[BL],linkout[BL], "   Link 1   Link 2   Link 3")

    -- Display this transputer type
    Display.Info(linkin[BL],linkout[BL],myID,myType,myMem,BL,linkID)

    -- Continually pass link information from other transputers down
    needboot := TRUE
    INT tmpID, tmpType, tmpMem, tmpBL:
    [4][2]INT tmplinkID:
    BYTE command:
    WHILE needboot
      SEQ
        needboot := FALSE                  -- Reset, will be set again
        SEQ I = 0 FOR 4                    -- if more info is to be sent.
          IF
            getInfo[I]
              SEQ
                -- Get info and display
                linkin[I] ? command; tmpID; tmpType; tmpMem; tmpBL; tmplinkID
                Display.Info(linkin[BL],linkout[BL],
                             tmpID,tmpType,tmpMem,tmpBL,tmplinkID)

                -- Check if it is the last
                IF
                  (command = tp.info)
                    needboot := TRUE       -- More info to come
                  (command = last.tp.info)
                    getInfo[I] := FALSE    -- No more info from this link
            TRUE
              SKIP

    -- WORM WIGGLING COMPLETE FOR THIS T/P
    TerminateServer(linkin[BL],linkout[BL])
:
```

Chapter 8 - SAPF Source Listing

(i) SAPF Source

"SAPF.PAS"

```
{$UNDEF DoXRef}
{$DEFINE DoXRef}

{$DEFINE Debug}
{$UNDEF Debug}

{------------------------------------------------------------------------}
{ This is the first version of my Semi-Automatic Parallelisation of Fortran. }
{ Date   : 9 September 1989                                              }
{ Author : A.M.Schuilenburg                                              }
{                                                                        }
Program SAPF;

Uses Crt, Dos, SAPFConstants, SAPFAnalyse, SAPFXRef;

Var InName, OutName, Strng          : BigString;
    Path                            : DirStr;
    Dummy1                          : NameStr;
    Dummy2                          : ExtStr;
    NameFile                        : Text;
    NameStack, NS                   : NamePtr;
    Error, I, P                     : Word;
    List, ListP                     : ListPtr;
    Abort                           : Boolean;
    MemS                            : LongInt;
    Dummy                           : String[4];

{Variables added for output}
Var OutFile                                          : Text;
    CBRec                                            : IntPtr;
    ComTmp, ComRoot, ComNew, ComLast, ComCom         : CBPtr;
    CrTmp                                            : CallPtr;
    VTmp                                             : Variable;
    CTmp                                             : CPtr;
    LTmp, LLast                                      : ListPtr;
    TmpDim                                           : DimPtr;
    TmpRef                                           : RefPtr;
    OutBuf                                           : Pointer;

{------------------------------------------------------------------------}
{ This procedure outputs the usage flags.                                }
{                                                                        }
Procedure Display_Usage(Usage : Word);
Begin
  if ((usage AND refbit) <> 0) then write(outfile,'R')
                               else write(outfile,'r');
  if ((usage AND assbit) <> 0) then write(outfile,'A')
                               else write(outfile,'a');
  if ((usage AND parbit) <> 0) then write(outfile,'P')
                               else write(outfile,'p');
  if ((usage AND conbit) <> 0) then write(outfile,'N')
                               else write(outfile,'n');
  if ((usage AND extbit) <> 0) then write(outfile,'E')
                               else write(outfile,'e');
  if ((usage AND combit) <> 0) then write(outfile,'M')
                               else write(outfile,'m');
  if ((usage AND callbit) <> 0) then write(outfile,'C')
                                else write(outfile,'c');
  if ((usage AND inbit) <> 0)   then write(outfile,'I')
                                else write(outfile,'i');
  if ((usage AND outbit) <> 0)  then write(outfile,'O')
                                else write(outfile,'o');
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure sorts the list of functions in alphabetic order.         }
{                                                                         }
Procedure SortProcs(Var List : ListPtr);
Var Arr           : Array[1..1000] Of ListPtr;
    N, I, NVars   : Word;
    NoSwap        : Boolean;
    Temp          : ListPtr;
Begin
  If (List <> NIL) AND (List^.Next <> NIL) Then Begin
    {PLACE VARAIABLES IN ARRAY}
    NVars := 0;
    While (List <> NIL) Do Begin
      inc(nvars);
      arr[nvars] := list;
      list := list^.next;
    End;

    {SORT THE ARRAY}
    N := Pred(NVars);
    Repeat
      NoSwap := TRUE;
      For I := 1 To N Do
       if (Arr[I]^.Name > Arr[Succ(I)]^.Name) Then begin
         Temp := Arr[I];
         Arr[I] := Arr[Succ(I)];
         Arr[Succ(I)] := Temp;
         NoSwap := FALSE;
       end;
      Dec(N);
    Until NoSwap;

    {PLACE THE ARRAY BACK INTO A LINKED LIST}
    List := Arr[1];
    For I := 1 To Pred(NVars) Do  Arr[I]^.Next := Arr[Succ(I)];
    Arr[NVars]^.Next := NIL;
  End;
End;

{-------------------------------------------------------------------------}
{ This procedure sorts the list of common blocks in alphabetic order.     }
{                                                                         }
Procedure SortCommon(Var List : CBPtr);
Var Arr           : Array[1..1000] Of CBPtr;
    N, I, NVars   : Word;
    NoSwap        : Boolean;
    Temp          : CBPtr;
Begin
  If (List <> NIL) AND (List^.Next <> NIL) Then Begin
    {PLACE VARAIABLES IN ARRAY}
    NVars := 0;
    While (List <> NIL) Do Begin
      inc(nvars);
      arr[nvars] := list;
      list := list^.next;
    End;

    {SORT THE ARRAY}
    N := Pred(NVars);
    Repeat
      NoSwap := TRUE;
      For I := 1 To N Do
       if (Arr[I]^.Name > Arr[Succ(I)]^.Name) Then begin
         Temp := Arr[I];
         Arr[I] := Arr[Succ(I)];
         Arr[Succ(I)] := Temp;
         NoSwap := FALSE;
       end;
      Dec(N);
    Until NoSwap;
```

```
    {PLACE THE ARRAY BACK INTO A LINKED LIST}
    List := Arr[1];
    For I := 1 To Pred(NVars) Do  Arr[I]^.Next := Arr[Succ(I)];
    Arr[NVars]^.Next := NIL;
  End;
End;

{-------------------------------------------------------------------------}
{ This procedure will output the variable type.                           }
{                                                                         }
Procedure Display_Type(Usage,Tipe : Word);
Begin
  if (usage = callbit) then write(outfile,' ') else
  case (tipe MOD impltype) of
    0 : write(outfile,' U');
    1 : write(outfile,'I');
    2 : write(outfile,'R');
    3 : write(outfile,'C');
    4 : write(outfile,'B');
    5 : write(outfile,'L');
    6 : write(outfile,'i');
    7 : write(outfile,'r');
  end;
End;
{=========================================================================}

Begin
  {Debug, memory information}
  MemS := MemAvail;

  InName := '';   OutName := '';   LogFlag := FALSE;
  AssignCrt(Out);  Rewrite(Out);   NOLFlag := FALSE;

  {Check for parameter/s passed}
  For P := 1 To ParamCount Do Begin
    Strng := ParamStr(P);
    For I := 1 To Length(Strng) Do Strng[I] := UpCase(Strng[I]);

    If (Strng[1] = '/') Then Begin
      {Process option}
      delete(Strng,1,1);

      if (Strng = 'LOG') then begin
        Close(Out);
        Assign(Out,'');  Rewrite(Out);
        LogFlag := TRUE;
      end else if (Strng = 'NOL') then begin
        NOLFlag := TRUE;
      end else begin
        Writeln('Invalid option "/',Strng,'"');
      end;
    End Else Begin
      If InName = '' Then InName := Strng Else
      If OutName = '' Then OutName := Strng Else Begin
        writeln('Too many parameters.  Only 2 expected.');
        halt(1);
      End;
    End;
  End;

  If (InName = '') Then Begin
    ClrScr;
    Writeln('SAPF (Semi-Automatic Parallelisation of FORTRAN');
    Writeln('============================================');
    Write('INPUT FILE    :');
    Readln(InName);
    If InName = '' Then Halt(1);
    Write('OUTPUT FILE   :');
    Readln(OutName);
  End;
```

```
If (OutName = '') Then Begin
  I := Pos('.',InName);
  If (I = 0) Then I := Length(InName) Else Dec(I);
  OutName := Copy(InName,1,I) + '.SAP';
End;

{INITIALISE STACK OF FILES TO PROCESS}
I := Pos('@',InName);
If (I = 0) Then Begin
  {SIMPLY PLACE THIS NAME ON TOP}
  New(NameStack);
  NameStack^.Name := FExpand(InName);
  NameStack^.Next := NIL;
End Else Begin
  InName := Copy(InName,Succ(I),255);
  Assign(NameFile,InName);
  {$I-}  Reset(NameFile);  {$I+}
  If (IOResult <> 0) Then Begin
    Writeln('File name list "',InName,'" not found.');
    Halt(1);
  End;

  FSplit(FExpand(InName),Path,Dummy1,Dummy2);
  NameStack := NIL;
  While Not EOF(NameFile) Do
   begin
     readln(namefile,Strng);
     if Strng <> '' then begin
       If (pos('/',Strng) = 0) AND (pos(':',Strng) = 0) then
         Strng := path+Strng;
       New(NS);
       NS^.Next := NameStack;
       NS^.Name := Strng;
       NameStack := NS;
     end;
   end;
End;

List := NIL;              Abort := FALSE;

{PROCESS ALL THE FILES}
While (NameStack <> NIL) Do Begin
  {Pop the stack}
  ns := namestack;
  namestack := ns^.next;

  {Analyse the file}
  analyse(ns^.name,list,error);

  if (error <> 0) then begin
    abort := TRUE;
    writeln;
    writeln('ERROR: ',Error,'     ANALYSIS ABORTED');
    writeln;
  end;

  {Kill name}
  dispose(ns);
End;

{DETERMINE IF ANY BLOCK DATA}
ListP := List;   I := 0;
While (ListP <> NIL) Do Begin
  if listp^.tipe = blcktype Then Begin
    Inc(I);
    Str(I:3,Dummy1);
    ListP^.Name := 'BLOCK DATA' + Dummy1;
  End;
  listp := listp^.next;
End;
```

```
  {ADD REFERENCES TO BLOCK DATA}
  If (I > 0) Then Begin
    listp := list;
    while (listp <> nil) do begin
      If (ListP^.Tipe <> BlckType) Then Begin
        {ADD CALL TO BLOCKS}
        for p := 1 to i do begin
          New(CrTmp);
          Str(P:3,Dummy1);
          CrTmp^.Name := 'BLOCK DATA' + Dummy1;
          CrTmp^.Line := 0;
          CrTmp^.Refs := NIL;
          CrTmp^.Next := ListP^.Calls;
          ListP^.Calls := CrTmp;
        end;
      End;
      ListP := ListP^.Next;
    end;
  End;

{$IFDEF DoXRef}
  {Determine if O.K. to continue}
  If Not Abort Then Begin
    CrossReference(List);
  End;
{$ENDIF}

  Writeln(Out);
  SortProcs(List);

  {Sort the common blocks, and collect unique common block list}
  LTmp := List;
  New(ComRoot);   {Dummy 1st}
  ComRoot^.Next := NIL;
  While (LTmp <> NIL) Do Begin
    sortcommon(ltmp^.common);

    {Merge the main and the sorted list}
    comtmp := comroot^.next;  comlast := comroot;
    comnew := ltmp^.common;
    while (comnew <> NIL) Do begin
      If (ComTmp = NIL) OR (ComTmp^.Name > ComNew^.Name) Then Begin
        {ITEM DOES NOT APPEAR, SO INSERT A COPY}
        new(comcom);
        comlast^.next := comcom;
        comcom^.name := comnew^.name;
        comcom^.refs := comnew^.refs;
        comcom^.next := comtmp;
        comlast := comcom;
        comnew := comnew^.next;
      End Else If (ComTmp^.Name < ComNew^.Name) Then Begin
        {NOT IN 2nd LIST}
        comlast := comtmp;
        comtmp := comtmp^.next
      End Else Begin
        {ITEM EXISTS, SKIP TO NEXT}
        comlast := comtmp;
        comtmp := comtmp^.next;
        comnew := comnew^.next;
      End;
    end;

    ltmp := ltmp^.next;
  End;

  {Remove dummy 1st}
  ComTmp := ComRoot;
  ComRoot := ComRoot^.Next;
  Dispose(ComTmp);
```

```
  {Output for the moment}
  If Outname = '' then outname := 'TEST';
  Assign(OutFile,OutName);
  GetMem(OutBuf,20000);
  SetTextBuf(OutFile,OutBuf^,20000);
  Rewrite(OutFile);

  {reverse list}
{  llast := nil;
  while list <> nil do begin
    ltmp := list^.next;
    list^.next := llast;
    llast := list;
    list := ltmp;
  end;
  list := llast;
}
  {DISPLAY THE COMMON BLOCKS}
  Writeln(OutFile,'SYSTEM COMMON BLOCKS');
  Writeln(OutFile,'====================');
  While (ComRoot <> NIL) Do Begin
    writeln(outfile,comroot^.name);
    cbrec := comroot^.refs;
    while (cbrec <> NIL) do begin
      Writeln(OutFile,'  *',CbRec^.V^.Name);
      cbrec := cbrec^.next;
    end;
    writeln(outfile);

    comtmp := comroot;
    comroot := comroot^.next;
    dispose(comtmp);
  End;

  {DISPLAY THE BODY}
  Writeln(OutFile,'SYSTEM PROCEDURES, FUNCTIONS AND PROGRAM');
  Writeln(OutFile,'========================================');
  While (List <> NIL) Do Begin
    with list^ do begin
     {OUTPUT THE DATA}
     {THE MAIN/SUBR/FUNC NAME AND PARAMETERS}
     Writeln(OutFile,Name,' ':31-Length(Name),Tipe);
     While (Params <> NIL) Do
      begin
        cbrec := params;
        params := params^.next;
        with cbrec^.v^ do begin
          Write(Outfile,'  => ',Name,' ':31-Length(Name));
          If ((Usage AND RefBit) = RefBit)
           Then Write(Outfile,'R')
           Else Write(Outfile,'r');
          If ((Usage AND AssBit) = AssBit)
           Then Writeln(Outfile,'A')
           Else Writeln(Outfile,'a');
        end;
        dispose(cbrec);
      end;
     Writeln(Outfile);

     Writeln(Outfile,'BLOCKS=',NumB); I := 1;
     While (Common <> NIL) Do Begin
       writeln(outfile,I:3,') ',Common^.Name);
       while (common^.refs <> NIL) do begin
         cbrec := common^.refs;
         common^.refs := cbrec^.next;
         with cbrec^.v^ do begin
           Write(Outfile,'     *',Name,' ':31-Length(Name));
           Display_Type(Usage,Tipe);
           Write(Outfile,'  ');
           Display_Usage(Usage);
           Writeln(Outfile,' (',Tipe:4,',',Usage:4,')');
```

```
          end;
          dispose(cbrec);
        end;

        comtmp := common;
        common := common^.next;
        dispose(comtmp);
        inc(i);
      End;
      Writeln(Outfile);

      Writeln(Outfile,'PROCEDURE AND FUNCTION CALLS');
{$IFDEF DoXRef}
      While (CList <> NIL) Do Begin
        if (clist^.c^.tipe <> blcktype) then
          writeln(outfile,' > ',CList^.C^.Name);
        ctmp := clist;
        clist := ctmp^.next;
        dispose(ctmp);
      End;
{$ENDIF}

      While (Calls <> NIL) Do Begin
         with calls^ do begin
          If (Pos('BLOCK DATA',Name) = 0) Then
           Writeln(Outfile,' > ',Name,' ':31-Length(Name),'Line : ',Line);
          While (Refs <> NIL) Do Begin
            if refs^.v <> NIL
             then writeln(outfile,'    *',refs^.v^.name)
             else writeln(outfile,'    *');
            tmpref := refs;
            refs := tmpref^.next;
            dispose(tmpref);
          End;
        end;

        {NEXT REFERENCE}
        crtmp := calls;
        calls := crtmp^.next;
        dispose(crtmp);
      End;

      Writeln(Outfile);

      Writeln(Outfile,'VARIABLES=',NumV:4,'   . NAME          Tp  Cm  Usage');
      While (Vars <> NIL) Do Begin
        With Vars^ Do Begin
          write(outfile,Name,' ':31-Length(Name),'  ');

          {DISPLAY EXPLICIT OR IMPLICIT TYPE}
          if (tipe >= impltype) then
          if (usage = callbit) then write(outfile,' ') else write(outfile,'+')
          else write(outfile,' ');

          display_type(usage,tipe);

          if (common <> 0) then write(outfile,' ',common:3,'  ')
                           else write(outfile,'      ');

          display_usage(usage);
          write(outfile,' ',usage:3);

          {DISPLAY THE DIMENSIONS (IF ANY)}
          if dimension = NIL then writeln(outfile) else begin
            Write(Outfile,'  Dimensions=');
            Repeat
              if (dimension^.d = minint)
                then write(outfile,'*')
                else write(outfile,dimension^.d);
              tmpdim := dimension;
              dimension := dimension^.next;
```

```
            dispose(tmpdim);
            if (dimension <> NIL) then write(outfile,',');
          Until (Dimension = NIL);
          Writeln(Outfile);
        end;
      End;

      VTmp := Vars;
      Vars := Vars^.Next;
      Dispose(VTMp);
    End;

    Writeln(OutFile,'================================================================================');
  end;

  ltmp := list;
  list := list^.next;
  dispose(ltmp);
End;
Close(OutFile);
FreeMem(OutBuf,20000);
Close(Out);

{Debug, memory information}
Writeln('START, Memavail = ',MemS);
Writeln('DONE , Memavail = ',MemAvail);
End.
```

"SAPFCONS.PAS"

```
Unit SAPFConstants;

Interface

Const Digits     = ['0'..'9'];
      SkipChars  = ['*','+','-','/',''''','"'];
      ValidChars = ['A'..'Z','0'..'9','_'];
      MaxDigits  = [' ','*','+','-','/','(',')','.',
                    ',',''''','"','=','0'..'9'];

      MinInt  : LongInt = Pred(-2147483647);

      NoBit   : Word = 0;
      RefBit  : Word = 1;
      AssBit  : Word = 2;
      ParBit  : Word = 4;
      ConBit  : Word = 16;
      ExtBit  : Word = 32;
      ComBit  : Word = 64;
      CallBit : Word = 128;
      OutBit  : Word = 256;
      InBit   : Word = 512;

      UnKnType : Byte = 0;
      IntType  : Byte = 1;
      RealType : Byte = 2;
      CompType : Byte = 3;
      CharType : Byte = 4;
      LogType  : Byte = 5;
      DblIType : Byte = 6;
      DblRType : Byte = 7;
      AnyType  : Byte = 98;
      ComnType : Byte = 99;
      ImplType : Byte = 128;
      BlckType : Byte = 252;
      FuncType : Byte = 253;
      SubrType : Byte = 254;
      ProgType : Byte = 255;
```

```
Type BigString = String[255];
     Strings = String[30];

     DimPtr = ^DimRec;
     DimRec = Record
       D    : LongInt;
       Next : DimPtr;
     End;

     Variable = ^VarRec;
     VarRec = Record
       Name      : Strings;
       Tipe      : Byte;        {0=UnKnown, 1=Integer, 2=Real, 3=Complex, 4=Byte}
       Usage     : Word;        {Bits: 0=Referenced, 1=Assigned, 2=Parameter}
       Common    : Word;        {Array reference to common block, or parameter}
                               {number if the variable is a parameter.}
       Dimension : DimPtr;
       Next      : Variable;
     End;

     IntPtr = ^IntRec;
     IntRec = Record
       V    : Variable;
       Next : IntPtr;
     End;

     RefPtr = ^RefRec;
     RefRec = Record
       V     : Variable;
       APosn : Boolean;          {ARRAY POSITION}
       Next  : RefPtr;
     End;

     CBPtr = ^CBType;
     CBType = Record
       Name : Strings;
       Refs : IntPtr;
       Next : CBPtr;
     End;

     CallPtr = ^CallRec;
     CallRec = Record
       Name : Strings;
       Line : Word;
       Refs : RefPtr;
       Next : CallPtr;
     End;

     Sets = Set Of 'A'..'Z';

     NamePtr = ^NameRec;
     NameRec = Record
       Name : BigString;
       Next : NamePtr;
     End;

     CPtr = ^CRec;
     ListPtr = ^ListRec;
     ListRec = Record
       Name   : Strings;
       Fyl    : BigString;
       Tipe   : Byte;
       Params : IntPtr;
       Common : CBPtr;
       Vars   : Variable;
       Calls  : CallPtr;
       CList  : CPtr;
       NumV,
       NumB   : Word;
       Next   : ListPtr;
     End;
```

```
    CRec = Record
      C    : ListPtr;
      Next : CPtr;
    End;

    LinePtr = ^LineRec;
    LineRec = Record
      Line : BigString;
      Next : LinePtr;
    End;

Var IntrinsicV     : Array[1..100] Of String[6];
    NIntrin        : Word;
    LogFlag, NOLFlag : Boolean;
    Out            : Text;

Implementation
Begin
  NIntrin := 0;
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'ABS';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'ACOS';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'AIMAG';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'AINT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'ALOG';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'ALOG10';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'AMAX0';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'AMAX1';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'AMIN0';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'AMIN1';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'AMOD';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'AND';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'ANINT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'ASIN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'ATAN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'ATAN2';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'BTEST';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'CABS';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'CCOS';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'CEXP';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'CHAR';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'CLOG';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'CMPLX';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'CONJG';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'COS';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'COSH';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'CSIN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'CSQRT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'CTAN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DABS';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DACOS';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DASIN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DATAN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DATAN2';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DBLE';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DCOS';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DCOSH';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DDIM';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DEXP';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DIM';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DINT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DLOG';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DLOG10';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DMAX1';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DMIN1';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DMOD';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DNINT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DPROD';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DSIGN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DSIN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DSINH';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DSQRT';
```

```
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DTAN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'DTANH';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'EXP';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'FLOAT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'IABS';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'IAND';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'IBCLR';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'IBITS';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'IBSET';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'ICHAR';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'IDIM';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'IDINT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'IDNINT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'IEOR';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'IFIX';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'INDEX';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'INT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'IOR';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'ISHFT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'ISHFTC';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'ISIGN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'LEN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'LGE';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'LGT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'LLE';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'LLT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'LOG';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'LOG10';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'MAX';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'MAX0';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'MAX1';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'MIN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'MIN0';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'MIN1';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'MOD';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'NINT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'NOT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'OR';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'REAL';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'SIGN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'SIN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'SINH';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'SNGL';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'SQRT';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'TAN';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'TANH';
  Inc(NIntrin);     IntrinsicV[NIntrin] := 'XOR';
End.
```

"SAPFANAL.PAS"

```
{$DEFINE Debug}
{$UNDEF Debug}

{------------------------------------------------------------------------}
{ This is the first version of my Semi-Automatic Parallelisation of Fortran. }
{ Date   : 25 September 1989                                             }
{ Author : A.M.Schuilenburg                                              }
{                                                                        }
Unit SAPFANALYSE;

Interface

Uses Crt, SAPFConstants;
Procedure Analyse(InName          : BigString;
                  Var Root        : ListPtr;
                  Var ENo         : Word);

Procedure RefVariable(RefToken : Strings; Ref : Word; Var Call : Boolean);
```

```
Implementation

Type ConstantPtr = ^ConstantRec;
     ConstantRec = Record
       V     : Variable;
       Val   : Real;
       Next  : ConstantPtr;
     End;

Var
  InFile          : Text;                    {FORTRAN input file}
  InBuf           : Pointer;

  Variables       : Array[0..1000] Of VarRec;  {List of variable definitions}
  CBlock          : Array[1..100] Of CBType;   {List of common block names}

  TypeSets                        : Array[1..6] Of Sets;
  AddSet                          : Sets;
  NVars, NBlock, LineNo, Error    : Word;        {Counters for array locations}
  BracketCount, LineNum           : Word;        {Counter for handling IF}
  DoLine, NextLine, ContLine      : LinePtr;
  Token, MainName                 : Strings;
  MainType, B                     : Byte;
  MainPars                        : IntPtr;
  I,J,K                           : Integer;
  Done                            : Boolean;     {Subroutine/Function end}
  CBRec                           : IntPtr;
  CalledRefs                      : CallPtr;
  Constants                       : ConstantPtr;

{-------------------------------------------------------------------------}
Procedure Show(T : Strings; Doll : LinePtr);
Begin
  write(T,' "');
  while doll <> NIL do begin
    write(doll^.line);
    doll := doll^.next;
  end;
  writeln('"');
End;

{-------------------------------------------------------------------------}
{ This routine replaces the token on the stack.                           }
{                                                                         }
Procedure Replace_Token(St : Strings);
Var Temp : LinePtr;
Begin
  If (St <> '') Then Begin
    If (DoLine <> NIL) Then Begin
      If (Length(St) + Length(DoLine^.Line)) < 255
       Then DoLine^.Line := St + ' ' + DoLine^.Line
       Else Begin
         New(Temp);
         Temp^.Line := St;
         Temp^.Next := DoLine;
         DoLine := Temp;
       End;
    End Else Begin
      New(DoLine);
      DoLine^.Line := Token;
      DoLine^.Next := NIL;
    End;
  End;
End;

{-------------------------------------------------------------------------}
{ This routine removes the leading spaces from LINE.                      }
{                                                                         }
Procedure Remove_Spaces(Var Doll : LinePtr);
Var L, I : Word;
    Cont : Boolean;
```

```
    Tmp  : LinePtr;
Begin
  Cont := TRUE;
  While Cont AND (Doll <> NIL) Do Begin
    with doll^ do begin
      L := Length(Line);   I := 1;
      While (I <= L) AND (Line[I] = ' ') Do Inc(I);
      Delete(Line,1,Pred(I));
    end;

    if doll^.line = '' then begin
      tmp := doll;
      doll := doll^.next;
      dispose(tmp);
    end else begin
      cont := FALSE;
      if (doll^.next <> NIL) then
      if (length(doll^.line) + length(doll^.next^.line)) < 255 then begin
        tmp := doll^.next;
        doll^.next := tmp^.next;
        doll^.line := doll^.line + tmp^.line;
        dispose(tmp);
      end;
    end;
  End;
End;

{------------------------------------------------------------------------}
{ This procedure will get a parameter from the list of parameters left in  }
{ LINE.                                                                    }
{                                                                          }
Procedure FixParameter(Var BracketCount : Word);
Var I, J      : Word;
    Temp, Tmp : LinePtr;
    TempStr   : BigString;
Begin
  Temp := DoLine;

{$IFDEF Debug}
writeln('IN FIXPARAMETER');
writeln('Line "',Temp^.Line,'"');
{$ENDIF}

  {DO A Bracket Count}
  Repeat
    I := 1;  J := Length(Temp^.Line);
    Repeat
      If (Temp^.Line[I] = '(') Then Inc(BracketCount)
      Else If (Temp^.Line[I] = ')') Then Dec(BracketCount);

      Inc(I);
    Until (I > J) OR (BracketCount=0);
    If (BracketCount <> 0) Then Temp := Temp^.Next;
  Until (BracketCount = 0) OR (Temp = NIL);

  If (BracketCount = 0) Then Begin
    TempStr := Copy(Temp^.Line,I,255);
    Delete(Temp^.Line,I,255);

    {Remove spaces from nextline}
    J := Length(TempStr);   I := 1;
    While (I <= J) AND (TempStr[I] = ' ') Do Inc(I);
    Delete(TempStr,1,Pred(I));

    {Remove rest of line, adding onto CONTLINE}
    ContLine := Temp^.Next;
    Temp^.Next := NIL;

    {Add to beginning of the rest}
    If (TempStr <> '') Then Begin
      if tempstr[1] in Digits then tempstr := 'GOTO ' + tempstr;
```

```
      new(temp);
      temp^.line := tempstr;
      temp^.next := contline;
      contline := temp;
      remove_spaces(contline);
    End;
  End;

{$IFDEF Debug}
writeln('BRACKETCOUNT = ',bracketcount);
show('Line',DoLine);
show('Cont',ContLine);
{$ENDIF}
End;

{------------------------------------------------------------------------}
{ This procedure will remove the boolean operations from the LINE.       }
{                                                                        }
Procedure Remove_Boolean_Ops;
Var I, J : Word;
    OP   : Strings;
    Doll : LinePtr;
Begin

{$IFDEF Debug}
writeln('IN IF, REMOVING BOOLEAN OPS');
show('Line',doline);
{$ENDIF}

  {Remove the boolean operations}
  Doll := DoLine;
  While (Doll <> NIL) Do Begin
    I := Pos('.',Doll^.Line);
    While (I > 0) Do Begin
      j := pos('.',copy(doll^.line,succ(i),length(doll^.line)));
      if j > 0 then begin
        OP := Copy(Doll^.Line,Succ(I),Pred(J));
        If (OP = 'LT') OR (OP = 'LE') OR (OP = 'EQ') OR
           (OP = 'GE') OR (OP = 'GT') OR (OP = 'OR') OR
           (OP = 'AND') OR (OP = 'NOT') OR (OP = 'NE') OR
           (OP = 'FALSE') OR (OP = 'TRUE') Then Begin
          Delete(Doll^.Line,I,J);     {REMOVE BOOLEAN ".OP"}
          Doll^.Line[I] := ' ';       {PUT " " OVER "."}
          Doll^.Line := Copy(Doll^.Line,1,I) + '0 ' + Copy(Doll^.Line,Succ(I),255);
          I := Pos('.',Doll^.Line);
        End Else I := J + I;
      End Else I := 0;
    End;

    Doll := Doll^.Next;
  End;
End;

{------------------------------------------------------------------------}
{ This is the procedure that reads LINE from the input file.             }
{                                                                        }
Procedure ReadLine;
Var Rd         : Boolean;
    I          : Word;
    St         : Strings;
    Ln         : BigString;
    Last, This : LinePtr;
Begin
  {TRANSFER LINE READ AHEAD TO CURRENT LINE}
  If (ContLine <> NIL) Then Begin
    DoLine := ContLine;
    ContLine := NIL;
  End Else Begin
    DoLine := NextLine;
    NextLine := NIL;
    LineNo := LineNum;
```

```
    If DoLine = NIL Then Exit;
    Last := DoLine;  While (Last^.Next <> NIL) Do Last := Last^.Next;

    {READ IN NEXT LINE (LINE AHEAD)}
    Repeat
      {GET A VALID LINE}
      Rd := FALSE;
      Repeat
        ReadLn(InFile,Ln);
        Inc(LineNum);
        If (Ln <> '') AND (UpCase(Ln[1]) <> 'C') Then Rd := TRUE;
      Until Rd OR EOF(InFile);

      {REMOVE TABS}
      I := Pos(#9,Ln);
      While (I <> 0) Do Begin
        Ln := Copy(Ln,1,Pred(I)) + '       ' + Copy(Ln,Succ(I),255);
        I := Pos(#9,Ln);
      End;

      {MAKE UPPER CASE}
      For I := 1 To Length(Ln) Do Ln[I] := Upcase(Ln[I]);

      {CHECK IF LINE IF CONTINUATION}
      Rd := (Copy(Ln,6,1) <> ' ');
      Delete(Ln,1,6);

      {REMOVE BOOLEAN VALUES}
      I := Pos('.',Ln);
      While (I <> 0) Do Begin
        j := pos('.',copy(ln,succ(i),255));
        if (j <> 0) then begin
          {POSSIBLE BOOLEAN}
          St := Copy(Ln,Succ(I),Pred(J));
          If (St = 'FALSE') OR (St = 'TRUE') Then Begin
            {REPLACE WITH DIGIT}
            Delete(Ln,I,J);
            Ln[I] := '0';
          End;
        end else i := 0;
        i := i+j;
      End;

      {REMOVE STUFF BETWEEN QUOTES}
      I := Pos('''',Ln);
      While (I <> 0) Do Begin
        j := pos('''',copy(ln,succ(i),255));
        if (j <> 0) then begin
          Delete(Ln,I,Succ(J));
        end else i := 0;
        i := i+j;
      End;

      {ADD WHERE APPROPRIATE}
      If (Ln <> '') Then Begin
        new(this);
        this^.line := ln;
        this^.next := NIL;
        if rd then begin  {CONTINUATION OF CURRENT LINE}
          last^.next := this;
          last := this;
        end else {NEXT LINE} nextline := this;
      End;
    Until (NextLine <> NIL) OR EOF(InFile);

    Remove_Spaces(DoLine);
  End;
{$IFDEF Debug}
writeln('************************READLINE*********************************');
show('Line',doline);
writeln;
```

```
show('Next',nextline);
writeln('*************************************************************************');
{$ENDIF}

End;

{-------------------------------------------------------------------------}
{ This routine searches for a variable in the main variable list.  If the  }
{ variable is not found it it returns a 0, otherwise the position in the   }
{ array is returned.                                                       }
{                                                                          }
Function FindVariable(Token : Strings) : Word;
Var Posn : Word;
    Go   : Boolean;
Begin
  Posn := 0;   Go := TRUE;
  While (Posn < NVars) AND Go Do
   begin
     inc(posn);
     go := (Variables[Posn].Name <> Token);
   end;
  If Go Then FindVariable := 0
        Else FindVariable := Posn;
End;

{-------------------------------------------------------------------------}
{ This routine is used to sort the variables (through a bubble sort).     }
{                                                                          }
Procedure Sort_Variables;
Var Temp   : VarRec;
    N, I   : Word;
    NoSwap : Boolean;
Begin
  If (NVars > 1) Then Begin
    N := Pred(NVars);
    Repeat
      NoSwap := TRUE;
      For I := 1 To N Do
       if (Variables[I].Name > Variables[Succ(I)].Name) Then begin
         Temp := Variables[I];
         Variables[I] := Variables[Succ(I)];
         Variables[Succ(I)] := Temp;
         NoSwap := FALSE;
       end;
      Dec(N);
    Until NoSwap;
  End;
End;

{-------------------------------------------------------------------------}
{ This routine will remove a number from LINE.                             }
{                                                                          }
Procedure Remove_Number(Var Line : BigString; Var Num : Strings);
Var L,I : Word;
Begin
  L := Length(Line);   I := 1;

  {REMOVE LEADING DIGITS}
  While (I <= L) AND (Line[I] IN Digits) Do Inc(I);

  If (I <= L) Then Begin
    If Line[I] = '.' Then Inc(I);

    {REMOVE TRAILING DIGITS}
    While (I <= L) AND (Line[I] IN Digits) Do Inc(I);

    If (I <= L) AND (Line[I] IN ['E','F','D']) Then Begin {REMOVE MANTISSA}
      I := I + 2;    {Kill mantissa and possible sign}
      While (I <= L) AND (Line[I] IN Digits) Do Inc(I);
    End;
  End;
```

```
  Dec(I);
  Num := Copy(Line,1,I);
  Delete(Line,1,I);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to extract a the TOKEN from LINE.                }
{                                                                         }
Procedure Extract_Token(Var Line : BigString; Var Result : Strings);
Var I, J, L : Word;
Begin
  I := 1;  J := 0;  L := Length(Line);
  While (I <= L) AND (Line[I] IN ValidChars) Do Begin
    Inc(J);
    Result[J] := Line[I];
    Inc(I);
  End;
  Delete(Line,I-J,J);
  Result[0] := Char(J);

  {REMOVE LEADING SPACES FROM LINE}
  I := 1;  L := Length(Line);
  While (I <= L) AND (Line[I] = ' ') Do Inc(I);
  Delete(Line,1,Pred(I));
End;

{-------------------------------------------------------------------------}
{ This function is used to evaluate the arithmetic expression passed.     }
{                                                                         }
Function Evaluate(Var Error : Boolean) : Real;
Type ValPtr = ^ValRec;
     ValRec = Record
       Val     : Real;
       Next    : ValPtr;
     End;

     OpPtr = ^OpRec;
     OpRec = Record
       Op   : Char;
       Next : OpPtr;
     End;

Var Ops  : OpPtr;
    Vals : ValPtr;

    Procedure PushVal(V : Real);
    Var Tmp : ValPtr;
    Begin
      New(Tmp);
      With Tmp^ Do Begin
        val     := v;
        next    := vals;
      End;
      Vals := Tmp;
    End;

    Procedure PopVal(Var V : Real);
    Var Tmp : ValPtr;
    Begin
      If (Vals = NIL) Then Begin
        V := 0.0;
      End Else Begin
        Tmp := Vals;
        Vals := Vals^.Next;
        V := Tmp^.Val;
        Dispose(Tmp);
      End;
    End;

    Procedure PushOp(O : Char);
    Var Tmp : OpPtr;
```

```
    Begin
      New(Tmp);
      With Tmp^ Do Begin
        op   := o;
        next := ops;
      End;
      Ops := Tmp;
    End;

    Procedure PopOp(Var O : Char);
    Var Tmp : OpPtr;
    Begin
      If (Ops = NIL) Then Begin
        Error := TRUE;
        O := '(';
      End Else Begin
        Tmp := Ops;
        Ops := Ops^.Next;
        O := Tmp^.Op;
        Dispose(Tmp);
      End;
    End;

    Function Order(Ch : Char) : Integer;
    Begin
      Case Ch Of
       '('     : Order := 0;
       '-','+' : Order := 1;
       '*','/' : Order := 2;
       '^'     : Order := 3;
      End;
    End;

    Function Compute(V1 : Real; Ch : Char; V2 : Real) : Real;
    Begin
      Case Ch Of
        '^' : Compute := Exp(Ln(V1)*V2);
        '*' : Compute := V1 * V2;
        '/' : Compute := V1 / V2;
        '+' : Compute := V1 + V2;
        '-' : Compute := V1 - V2;
        Else Compute := 0.0;
      End;
    End;

Var St         : Strings;
    V1, V2     : Real;
    Ch, Ahead  : Char;
    Tmp        : ConstantPtr;
    IsVariable : Boolean;
    I          : Word;
Begin
  {INITIALISE THE STACKS}
  Ops  := NIL;
  Vals := NIL;
  Error := FALSE;
  IsVariable := FALSE;

  {Remove any spaces from the line}
  Remove_Spaces(DoLine);

  {Do the evaluation}
  While (DoLine <> NIL) AND (DoLine^.Line <> '') AND
        (DoLine^.Line[1] <> ',') AND (DoLine^.Line[1] <> ':') Do Begin
    If DoLine^.Line[1] In Digits Then Begin
      {Extract the number}
      remove_number(doline^.line,st);
      remove_spaces(doline);

      {Check if valid}
      Val(St,V1,I);
```

```
    If (I = 0) Then PushVal(V1) Else Begin
      Writeln(Out,'ERROR:  Invalid numeric "',st,'"');
      Writeln(Out,'ERROR:  Substituting 1 and attempting to continue.');
      Error := TRUE;
      PushVal(1.0);
    End;

  End Else If DoLine^.Line[1] IN ['A'..'Z'] Then Begin
    {EXTRACT THE TOKEN}
    Extract_Token(DoLine^.Line,St);
    Remove_Spaces(DoLine);

    {SEARCH FOR THE TOKEN}
    Tmp := Constants;
    While (Tmp <> NIL) AND (Tmp^.V^.Name <> St) Do Tmp := Tmp^.Next;

    {TOKEN FOUND, ADD VALUE}
    If (Tmp <> NIL) Then PushVal(Tmp^.Val) Else Begin
      {Put 0 on the stack}
      PushVal(0.0);

      {Token not valid, check if variable}
      I := FindVariable(St);
      If (I=0) Then Begin
        {Token invalid}
        Writeln(Out,'ERROR: Invalid constant "',st,'" found at line ',LineNo);
        Writeln(Out,'ERROR: Setting Constant Value to 0 and attempting to continue.');
        Error := TRUE;
      End Else IsVariable := TRUE;     {VARIABLE IN CONSTANT}
    End;
  End Else Begin
    {Get OP}
    Ch := DoLine^.Line[1];
    Delete(DoLine^.Line,1,1);
    Remove_Spaces(DoLine);

    {Check for power}
    If (Ch = '*') AND (DoLine <> NIL) AND (Copy(DoLine^.Line,1,1) = '*') Then Begin
      Ch := '^';
      Delete(DoLine^.Line,1,1);
      Remove_Spaces(DoLine);
    End;

    {Ensure operand}
    If Ch In ['+','-','*','/','^','(',')'] Then Begin
      {Check for brackets}
      If (Ch = '(') Then Begin
        PushOp(Ch);
        If (DoLine <> NIL) AND (Copy(DoLine^.Line,1,1) = '-')
          Then PushVal(0.0);
      End Else If (Ch = ')') Then Begin
        {Evaluate until a bracket is encountered}
        PopOp(Ch);
        While (Ch <> '(') Do Begin
          If (Vals = NIL) Then Begin
            {ILLEGAL ")" ENCOUNTERED}
            Writeln(Out,'ERROR: Illegal ")" encountered, ignoring');
            Ch := '(';
          End Else Begin
            PopVal(V2);
            PopVal(V1);
            PushVal(Compute(V1,Ch,V2));
            PopOp(Ch);
          End;
        End;
      End Else Begin
        If (Ch = '*') AND (Vals = NIL) Then PushVal(MinInt) Else
        If (Ops = NIL) Then PushOp(Ch) Else
        If Order(Ops^.Op) >= Order(Ch) Then Begin
          {Do evaluation, or push on stack}
          PopVal(V2);
```

```
            PopVal(V1);
            PopOp(Ahead);
            PushVal(Compute(V1,Ahead,V2));
            PushOp(Ch);
          End Else PushOp(Ch);
        End;
      End Else Begin
        Writeln(Out,'ERROR Illegal operand "',ch,'" in expression.');
        Writeln(Out,'ERROR Attempting to continue.');
        Error := TRUE;
      End;
    End;
  End;

  While (Ops <> NIL) Do Begin
    PopVal(V2);
    PopVal(V1);
    PopOp(Ahead);
    PushVal(Compute(V1,Ahead,V2));
  End;

  If Vals = NIL Then Begin
    evaluate := 0;
    error := TRUE;
  End Else Begin
    popval(v1);
    if isvariable then v1 := minint;
    if (vals <> NIL) Then Begin
      repeat popval(v2) until vals = NIL;
      Writeln(Out,'ERROR: Expression cannot be evaluated');
      Error := TRUE;
    End;
    evaluate := v1;
  End;
End;

{------------------------------------------------------------------------}
{ This function returns the next token being read from the input file.   }
{                                                                        }
Function GetString  : Strings;
Var Result, Dummy   : Strings;
    Done, LineRead  : Boolean;
    I, J, Zero, LB  : Word;
    BC              : Integer;
    TmpLine         : LinePtr;
    Ch              : Char;
    BCSet           : Boolean;
Begin
  LineRead := FALSE;
  Result := '';

  {Pre-process line}
  While (DoLine <> NIL) AND (DoLine^.Line[1] in MaxDigits) Do Begin
    If DoLine^.Line[1] IN Digits Then Remove_Number(DoLine^.Line,Dummy)
    Else Delete(DoLine^.Line,1,1);
    If (DoLine^.Line = '') Then Begin
      tmpline := doline;
      doline := doline^.next;
      dispose(tmpline);
    End;
  End;

  {ENSURE DATA IS THERE}
  If (DoLine = NIL) Then ReadLine Else Begin
    {Find the token}
    Extract_Token(DoLine^.Line,Result);

    {Remove any spaces}
    Remove_Spaces(DoLine);

    {Check for assignment}
```

```
    If (DoLine <> NIL) Then Begin
      I := Pos('=',DoLine^.Line);
      If (I <> 0) Then Begin
        Ch := DoLine^.Line[1];
        If (I <> 1) Then
        If (CH = '(') Then Begin
          {Check for array reference}
          J := 1;   bc := 0;  BCSet := FALSE;  zero := 0;
          While (J < I) Do Begin
            if doline^.line[j] = '(' then begin
              inc(bc);
              bcset := TRUE
            end else if doline^.line[j] = ')' then begin
              dec(bc);
              if (bc=0) then inc(zero);
              bcset := TRUE;
              lb := j;
            end;
            inc(j);
          End;

          If (BC=0) AND BCSet AND (Zero <= 1) Then Begin
            {Test for actual variable, spaces between last bracket and =}
            ch := '=';
            inc(lb);
            while (lb < i) do begin
              if (doline^.line[lb] <> ' ') then ch := ' ';
              inc(lb);
            end;
          End Else Ch := ' ';
        End Else Ch := ' ';

        If (Ch = '=') Then Begin
          Result := Result + '=';
          {ArrayRef := (Pos('(',Copy(DoLine^.Line,1,I)) > 0);}
          DoLine^.Line[I] := ' ';          {MASK OUT ASSIGNMENT}
        End;
      End;
    End;
  End;

{$IFDEF Debug}
writeln('GETSTRING="',result,'"');
show('GETSTR LN=',doline);
{$ENDIF}
  GetString := Result;
End;

{-------------------------------------------------------------------------}
{ This function returns the set of letters used by an IMPLICIT statement.  }
{                                                                          }
Procedure GetSet(Var Result : Sets);
Var Start, Stop, Sep : Char;
    TmpLine          : LinePtr;
Begin
  Result := [];
  If (Copy(DoLine^.Line,1,1) <> '(') Then Begin
    writeln(out,'ERROR: Error in implicit statement');
  End Else Delete(DoLine^.Line,1,1);
  Remove_Spaces(DoLine);

  While (DoLine <> NIL) AND (Copy(DoLine^.Line,1,1) <> ')') Do
   begin
     if (doline^.line = '') then begin
       TmpLine := DoLine;
       DoLine := TmpLine^.Next;
       Dispose(TmpLine);
     end else begin
       start := doline^.line[1];
       delete(doline^.line,1,1);   remove_spaces(doline);
       sep := doline^.line[1];
```

```
      if (sep = ',') then begin
        Result := Result + [Start];
        Delete(DoLine^.Line,1,1);
      end else if (sep = '-') then begin
        Delete(DoLine^.Line,1,1);  Remove_Spaces(DoLine);
        Stop := DoLine^.Line[1];
        Result := Result + [Start..Stop];
        Delete(DoLine^.Line,1,1);
      end else if (sep = ')') then begin
        Result := Result + [Start];
      end;
      remove_spaces(doline);
    end;
  end;

  {REMOVE THE BRACKET}
  If (DoLine <> NIL) Then Begin
    Delete(DoLine^.Line,1,1);
    Remove_Spaces(DoLine);
  End;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to fix the set types in an IMPLICIT.             }
{                                                                         }
Procedure FixSets(No : Byte);
Var  AddSet : Sets;
     I      : Word;
Begin
  I := Pos('(',DoLine^.Line);
  Delete(DoLine^.Line,1,Pred(I));
  GetSet(AddSet);
  For I := 1 To 6 Do
    if (i = no) then typesets[i] := typesets[i] + addset
                else typesets[i] := typesets[i] - addset
End;

{-------------------------------------------------------------------------}
{ This function will return TRUE if the name passed is an intrinsic function }
{ and FALSE otherwise.                                                    }
{                                                                         }
Function Intrinsic(Token : Strings) : Boolean;
Var Top, Bottom, Look : Word;
    Result            : Boolean;
Begin
  Top := NIntrin;
  Bottom := 1;

  Result := FALSE;
  Repeat
    look := (top + bottom) div 2;
    if (intrinsicv[look] = token) then Result := TRUE else begin
      If IntrinsicV[Look] > Token
        Then Top := Pred(Look)
        Else Bottom := Succ(Look);
    end;
  Until Result OR (Top < Bottom);

  Intrinsic := Result;
End;

{-------------------------------------------------------------------------}
{ This routine simply adds a variable to the main variable list.          }
{                                                                         }
Procedure AddVariable(Token : Strings;
                      T     : Byte;
                      U     : Word;
                      C     : Word);
Var Add : Boolean;
    I, L : Word;
```

```
Begin
  Add := TRUE;  L := Length(Token);
  If (DoLine^.Line[1] = '(') Then Begin  {CHECK NOT INTRINSIC FUNCTION}
    Add := Not Intrinsic(Token);
  End;

  If Add Then Begin
{$IFDEF Debug}
writeln('Adding "',token,'"');
show('Line',doline);
{$ENDIF}

    Inc(NVars);
    With Variables[NVars] Do Begin
      name := token;
      tipe := t;
      usage := u;
      common := c;
      dimension := NIL;
    End;
  End;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to issue a set a variable definition.            }
{                                                                         }
Procedure SetVariable(RefToken : Strings; Ref : Word);
Var I, BC : Word;
    N     : CallPtr;
Begin

{$IFDEF Debug}
writeln('IN SETVARIABLE "',reftoken,'" : ',ref);
{$ENDIF}

  I := FindVariable(RefToken);
  If (I > 0) Then Begin
    Variables[I].Usage := (Variables[I].Usage OR Ref);
    Ref := Variables[I].Usage;
  End Else Begin
    AddVariable(RefToken,UnKnType,Ref,0);
    I := NVars;
  End;

{$IFDEF Debug}
writeln('OUT SETVARIABLE "',RefToken,'"');
{$ENDIF}
End;

{-------------------------------------------------------------------------}
{ This procedure adds the variables in brackets which is an offeset to an  }
{ array position.                                                          }
{                                                                          }
Procedure Add_Array_Refs;
Var BC, I, L                : Word;
    Token, Dummy            : Strings;
    Got, DelComma, CallFlag : Boolean;
    TmpLine                 : LinePtr;
Begin
  {ACTUAL PARAMETER EXTRACTION LOOP}
  BC := 0;   DelComma := FALSE;

{$IFDEF Debug}
writeln('adding array index reference');
{$ENDIF}

  L := Length(DoLine^.Line);   I := 1;
  Repeat

{$IFDEF Debug}
write(copy(doline^.line,1,pred(i)));  highvideo;  write(doline^.line[i]);  lowvideo;
```

```
writeln(copy(doline^.line,succ(i),255));
{$ENDIF}

    if (i > l) then begin
      {INPUT THE CONTINUATION LINE}
      TmpLine := DoLine;
      DoLine := TmpLine^.Next;
      Dispose(TmpLine);

      If (DoLine = NIL) Then Begin
        writeln(out,'ERROR: Invalid array reference at line ',LineNo);
        writeln(out,'ERROR: Skipped to next line');
        bc := 0;
      End Else Begin
        If DelComma AND (Copy(DoLine^.Line,1,1) = ',') Then
          Delete(DoLine^.Line,1,1);
        L := Length(DoLine^.Line);  I := 1;
      End;
    end else begin
      If (DoLine^.Line[I] = '(') Then Begin
        inc(bc);
        inc(i);
      End Else If (DoLine^.Line[I] = ')') Then Begin
        dec(bc);
        inc(i);
      End Else Begin {TRY AND EXTRACT ARRAY INDEX}
        {Ettempt extract}
        got := FALSE;  token := '';
        repeat

{$IFDEF Debug}
writeln('extracting array index token');
{$ENDIF}

          If (DoLine^.Line[I] IN Digits + ['.']) Then Begin
            Delete(DoLine^.Line,1,Pred(I));
            Remove_Number(DoLine^.Line,Dummy);
            If Copy(DoLine^.Line,1,1) = 'H' Then Begin
              {TEXT PARAMETER}
              Val(Dummy,I,L);
              If (L = 0) Then delete(doline^.line,1,succ(i))  {REMOVE CHARS}
              Else Begin
                Writeln(Out,'ERROR: Invalid text string size "',dummy,'" at line ',LineNo);
                Writeln(Out,'ERROR: Attempting to continue.');
              End;
            End;
            Remove_Spaces(DoLine);
            L := Length(DoLine^.Line);  I := 1;
          End Else If (DoLine^.Line[I] IN (SkipChars + [' '])) Then Inc(I)
          Else If (DoLine^.Line[I] = ')') Then Got := TRUE
          Else If (DoLine^.Line[I] = '(') Then Begin
            inc(i);
            inc(bc);
          End Else If (DoLine^.Line[I] = ',') Then Begin
            Inc(I);
            Got := TRUE;
          End Else If (DoLine^.Line[I] IN ValidChars) Then Begin
            {EXTRACT INDEX TOKEN}
            J := 0;
            Repeat
              inc(j);
              token[j] := doline^.line[i];
              inc(i);
            Until (I > L) OR NOT (DoLine^.Line[I] IN ValidChars) OR Got;

            {SET THE TOKEN}
            Got := TRUE;
            Token[0] := Chr(J);

            {REMOVE TOKEN FROM LINE}
            Delete(DoLine^.Line,1,Pred(I));
```

```
            Remove_Spaces(DoLine);
            L := Length(DoLine^.Line);   I := 1;
          End Else Begin
            writeln(out,'ERROR: Invalid character "',DoLine^.Line[I],'" encountered at line ',LineNo);
            writeln(out,'ERROR: Skipped to next line');
            delete(doline^.line,1,pred(i));
            got := TRUE;
          End;
        until got or (i > l);

{$IFDEF Debug}
writeln('index token is "',token,'"');
{$ENDIF}

        {Set reference}
        if (token <> '') then RefVariable(Token,RefBit,CallFlag);

{$IFDEF Debug}
show('added index reference, line is',doline);
{$ENDIF}

        {Remove Comma if one exists}
        If (Copy(DoLine^.Line,1,1) = ',') Then Begin
          Delete(DoLine^.Line,1,1);
          DelComma := FALSE;
          L := Length(DoLine^.Line);  I := 1;
        End Else DelComma := TRUE;
      End;
    end;
  Until (BC = 0);

  {Remove what was processed}
  Delete(DoLine^.Line,1,Pred(I));
  Remove_Spaces(DoLine);

{$IFDEF Debug}
show('done add array index, line is',doline);
{$ENDIF}
End;

{-------------------------------------------------------------------------}
{ This procedure adds the variables in brackets which may be parameters to  }
{ a function call.                                                          }
{                                                                           }
Procedure Add_Parameters(Var Param : RefPtr);
Var BC, I, J, K, L                                  : Word;
    Token, Dummy                                    : Strings;
    LastPtr, NewPtr                                 : RefPtr;
    Got, DelComma, NextParam, ParamAdded, CallFlag : Boolean;
    TmpLine                                         : LinePtr;
Begin
  {SET UP PARAMETER LIST}
  LastPtr := Param;
  If (LastPtr <> NIL) Then
   While (LastPtr^.Next <> NIL) Do LastPtr := LastPtr^.Next;

  {ACTUAL PARAMETER EXTRACTION LOOP}
  If DoLine = NIL Then Exit;
  BC := 0;              L := Length(DoLine^.Line);
  I := 1;               LastPtr := NIL;

  DelComma := FALSE;    NextParam := TRUE;    ParamAdded := FALSE;

{$IFDEF Debug}
writeln('adding parameter');
{$ENDIF}

  Repeat

{$IFDEF Debug}
write(copy(doline^.line,1,pred(i)));  highvideo;  write(doline^.line[i]);  lowvideo;
```

```
writeln(copy(doline^.line,succ(i),255));
{$ENDIF}

    if (i > l) then begin
      {INPUT THE CONTINUATION LINE}
      TmpLine := DoLine;
      DoLine := TmpLine^.Next;
      Dispose(TmpLine);

      If (DoLine = NIL) Then Begin
        writeln(out,'ERROR: Invalid parameter reference at line ',LineNo);
        writeln(out,'ERROR: Skipped to next line');
        bc := 0;
      End Else Begin
        If DelComma AND (Copy(DoLine^.Line,1,1) = ',') Then Begin
          Delete(DoLine^.Line,1,1);
          NextParam := TRUE;
          ParamAdded := FALSE;
          DelComma := FALSE;
          J := 0;
        End;
        L := Length(DoLine^.Line);  I := 1;
      End;
    end else begin
      If (DoLine^.Line[I] = '(') Then Begin
        inc(bc);
        inc(i);
      End Else If (DoLine^.Line[I] = ')') Then Begin
        dec(bc);
        inc(i);
      End Else Begin {TRY AND EXTRACT PARAMETER}
        {Ettempt extract}
        got := FALSE;  token := '';
        repeat

{$IFDEF Debug}
writeln('extracting parameter token');
{$ENDIF}

          If (DoLine^.Line[I] IN Digits + ['.']) Then Begin
            Delete(DoLine^.Line,1,Pred(I));
            Remove_Number(DoLine^.Line,Dummy);
            If Copy(DoLine^.Line,1,1) = 'H' Then Begin
              {TEXT PARAMETER}
              Val(Dummy,I,L);
              If (L = 0) Then delete(doline^.line,1,succ(i))  {REMOVE CHARS}
              Else Begin
                Writeln(Out,'ERROR: Invalid text string size "',dummy,'" at line ',LineNo);
                Writeln(Out,'ERROR: Attempting to continue.');
              End;
            End;
            Remove_Spaces(DoLine);
            L := Length(DoLine^.Line);  I := 1;
            NextParam := FALSE;
          End Else If (DoLine^.Line[I] IN SkipChars) Then Begin
            Inc(I);
            NextParam := FALSE;
          End Else If (DoLine^.Line[I] = ' ') Then Inc(I)
          Else If (DoLine^.Line[I] = ')') Then Got := TRUE
          Else If (DoLine^.Line[I] = '(') Then Begin
            inc(i);
            inc(bc);
          End Else If (DoLine^.Line[I] = ',') Then Begin
            Delete(DoLine^.Line,1,Pred(I));
            Got := TRUE;
            NextParam := TRUE;
            {J := 0;}
          End Else If (DoLine^.Line[I] IN ValidChars) Then Begin
            {EXTRACT PARAMETER TOKEN}
            K := 0;
            Repeat
```

```
            inc(k);
            token[k] := doline^.line[i];
            inc(i);
          Until (I > L) OR NOT (DoLine^.Line[I] IN ValidChars) OR Got;

          {SET THE TOKEN}
          Got := TRUE;
          Token[0] := Chr(K);

          {REMOVE TOKEN FROM LINE}
          Delete(DoLine^.Line,1,Pred(I));
          Remove_Spaces(DoLine);
          L := Length(DoLine^.Line);   I := 1;
        End Else Begin
          writeln(out,'ERROR: Invalid character "',DoLine^.Line[I],'" encountered at line ',LineNo);
          writeln(out,'ERROR: Skipped to next line');
          delete(doline^.line,1,pred(i));
          got := TRUE;
        End;
      until got or (i > l);

      {Set reference}
      if (token = '') then begin
        J := 0;  CallFlag := FALSE;
        If ParamAdded Then If (LastPtr^.V^.Name <> '')
         Then SetVariable(LastPtr^.V^.Name,RefBit);
      end else
      if not nextparam then begin
        RefVariable(Token,RefBit,CallFlag);
        L := Length(DoLine^.Line);  I := 1;

        {PARAMETER A REFERENCE, WILL BE DELETED SO UPDATE}
        If ParamAdded Then If (LastPtr^.V^.Name <> '')
         Then SetVariable(LastPtr^.V^.Name,RefBit);

        J := 0;
      end else begin
        RefVariable(Token,NoBit,CallFlag);
        L := Length(DoLine^.Line);  I := 1;
        J := FindVariable(Token);
      end;

      {Add item to list}
      if not paramadded then begin
        New(NewPtr);
        NewPtr^.APosn := CallFlag;
        NewPtr^.Next := NIL;
        If (Param = NIL)
         then Param := NewPtr
         else LastPtr^.Next := NewPtr;
        LastPtr := Newptr;
        ParamAdded := TRUE;
        NextParam := FALSE;
      end;

{$IFDEF Debug}
writeln('token is "',token,'"');
{$ENDIF}

      {SET VARIABLE POINTER}
      LastPtr^.V := Addr(Variables[J]);

{$IFDEF Debug}
show('added parameter, line is',doline);
{$ENDIF}

      {Remove Comma if one exists}
      If (Copy(DoLine^.Line,1,1) = ',') Then Begin
        Delete(DoLine^.Line,1,1);
        DelComma := FALSE;
        L := Length(DoLine^.Line);  I := 1;
```

```
          NextParam := TRUE;
          J := 0;
        End Else DelComma := TRUE;

        {Move onto next parameter ?}
        If NextParam Then ParamAdded := FALSE;
      End;
    end;
  Until (BC = 0);

  {Remove what was processed}
  Delete(DoLine^.Line,1,Pred(I));
  Remove_Spaces(DoLine);

{$IFDEF Debug}
show('done add parameter, line is',doline);
{$ENDIF}
End;

{-------------------------------------------------------------------------}
{ This procedure is used to issue a reference to a variable.              }
{                                                                         }
Procedure RefVariable(RefToken : Strings; Ref : Word; Var Call : Boolean);
Var I, BC  : Word;
    N      : CallPtr;
    Params : Boolean;
Begin

{$IFDEF Debug}
writeln('IN REFVARIABLE "',reftoken,'" : ',ref);
{$ENDIF}

  I := FindVariable(RefToken);
  Params := Copy(DoLine^.Line,1,1) = '(';

  If (I > 0) Then Begin
    If (Variables[I].Dimension = NIL) AND Params Then
      {BRACKET AND NO DIMENSION INDICATE FUNCTION CALL}
      Ref := Ref + CallBit;

    Variables[I].Usage := (Variables[I].Usage OR Ref);
    Ref := Variables[I].Usage;
  End Else Begin
    If Params Then Begin
      {NOT FOUND AND BRACKET INDICATE FUNCTION CALL}
      Ref := Ref + CallBit;
    End;

    AddVariable(RefToken,UnKnType,Ref,0);
    I := NVars;
  End;

  If Params Then
  If ((Ref AND CallBit) = CallBit) Then Begin {FUNCTION CALL, PLACE REFERENCE}
    Call := TRUE;

    If Not(Intrinsic(RefToken)) Then
     begin
       {ADD TO CALL LIST}
       New(N);
       With N^ Do Begin
         Name := RefToken;
         Line := LineNo;
         Refs := NIL;
         Next := CalledRefs;
       End;
       CalledRefs := N;

       {ADD PARAMETERS OF FUNCTION}
       Add_Parameters(N^.Refs);
     end else add_array_refs;          {INTRINSIC, SO REFERENCE PARAMETERS}
```

```
  End Else Begin
    Call := TRUE;
    Add_Array_Refs;
  End Else Call := FALSE;

{$IFDEF Debug}
writeln('OUT REFVARIABLE "',RefToken,'"');
{$ENDIF}
End;

{-------------------------------------------------------------------------}
{ This procedure is used to issue a reference to a procedure call.        }
{                                                                         }
Procedure Handle_Call;
Var Token  : Strings;
    I      : Integer;
    N      : CallPtr;
    I1, I2 : IntPtr;
Begin
  {CALLED PROCEDURE NAME}
  Token := GetString;
  I := FindVariable(Token);
  If I > 0
   Then Variables[I].Usage := (Variables[I].Usage OR CallBit)
   Else AddVariable(Token,UnKnType,CallBit,0);

  {ADD TO CALL LIST}
  New(N);

{$IFDEF Debug}
writeln('ADD CALL "',token,'"');
show('LINE IS',DoLine);
{$ENDIF}

  With N^ Do Begin
    Name := Token;
    Line := LineNo;
    Next := CalledRefs;
    Refs := NIL;
  End;
  CalledRefs := N;

  Add_Parameters(N^.Refs);
  Token := GetString;
  If (Token <> '') Then Begin
    Writeln(Out,'ERROR: Call at line ',lineno,' cannot be recognised.');
    Writeln(Out,'ERROR: Attempting to recover.');
    Replace_Token(Token);
  End;
End;

{-------------------------------------------------------------------------}
{ This function will return whether a token is reserved or not.           }
{                                                                         }
Function Token_Reserved(Token : Strings) : Boolean;
Var Flag : Boolean;
Begin
  Flag := FALSE;
  If (Token = 'GOTO')        Then Flag := TRUE Else
  If (Token = 'GO')          Then Flag := TRUE Else
  If (Token = 'THEN')        Then Flag := TRUE Else
  If (Token = 'RETURN')      Then Flag := TRUE Else
  If (Token = 'ENDIF')       Then Flag := TRUE Else
  If (Token = 'CONTINUE')    Then Flag := TRUE Else
  If (Token = 'OPEN')        Then Flag := TRUE Else
  If (Token = 'CLOSE')       Then Flag := TRUE Else
  If (Token = 'INQUIRE')     Then Flag := TRUE Else
  If (Token = 'REWIND')      Then Flag := TRUE Else
  If (Token = 'BACKSPACE')   Then Flag := TRUE Else
  If (Token = 'ENDFILE')     Then Flag := TRUE Else
  If (Token = 'ENTRY')       Then Flag := TRUE Else
```

```
  If (Token = 'SAVE')        Then Flag := TRUE Else
  If (Token = 'STOP')        Then Flag := TRUE Else
  If (Token = 'PAUSE')       Then Flag := TRUE;

  Token_Reserved := Flag;
End;

{-------------------------------------------------------------------------}
{ This procedure will handle if statements.                               }
{                                                                         }
Procedure Handle_If;
Var BC    : Word;
    Dummy : Boolean;
Begin
  ContLine := NIL;
  BracketCount := 0;

  {Remove the parameter}
  FixParameter(BracketCount);

  Remove_Boolean_Ops;

  {REFERENCES AND ASSIGNMENTS}
  Token := GetString;
  While (Token <> '') Do Begin
    If (Token <> 'THEN') Then Begin
      RefVariable(Token,RefBit,Dummy);
      Token := GetString;
    End Else Repeat  Token := GetString;  Until (Token = '');
  End;

End;

{-------------------------------------------------------------------------}
Procedure Add_Definition(Token : Strings; Tipe : Byte; Var I : Word);
Var J                 : Word;
    Start, Term       : LongInt;
    TmpPtr            : LinePtr;
    Error             : Boolean;
    Ch                : Char;
    Root, This, Last  : DimPtr;
Begin
  I := FindVariable(token);
  If (I = 0) then Begin
    AddVariable(Token,Tipe,NoBit,0);
    I := NVars;
  End Else If (Tipe <> UnKnType) Then Variables[I].Tipe := Tipe;

  Remove_Spaces(DoLine);
  If (Copy(DoLine^.Line,1,1) = '(') Then Begin
    {DIMENSION DECLARATION}
    j := 0;
    fixparameter(j);

    {DUMMY 1st DIMENSION}
    new(root);
    root^.next := NIL;
    last := root;

    {REMOVE THE BRACKETS ENCLOSING}
    delete(doline^.line,1,1);
    tmpptr := doline;  while (tmpptr^.next <> NIL) do tmpptr := tmpptr^.next;
    delete(doline^.line,length(doline^.line),1);

    {GET THE DIMENSION SIZES}
    repeat
      Term := Round(Evaluate(Error));
      If Error AND (Term = MinInt) Then Error := FALSE;

      If Error Then Begin
        writeln(out,'ERROR: Illegal dimension at line ',lineno,'.');
```

```
        writeln(out,'ERROR: Ignoring dimension and continuing.');
      End Else Begin
        if (doline = NIL) then ch := ' ' else begin
          ch := doline^.line[1];
          delete(doline^.line,1,1);
        end;
        if (ch = ',') OR (ch = ' ') then start := 1 else
        if (ch = ':') then begin
          Start := Term;
          Term := Round(Evaluate(Error));
          If Error AND (Term = MinInt) Then Error := FALSE;
          If Error Then Begin
            Writeln(Out,'ERROR: Illegal upper bound at line ',lineno,'.');
            Writeln(Out,'ERROR: Setting dimension size to 1 and continuing.');
            Term := 1;   Start := 1;
          End;
        end else begin
          Writeln(Out,'ERROR: Illegal character "',Ch,'" in dimension statement at line ',lineno,'.');
          Writeln(Out,'ERROR: Attempting to continue.');
          Term := 1;   Start := 1;
        end;

        {GET DIMENSION SIZE}
        if (term = minint) OR (start = minint) then start := minint
        else start := (term-start)+1;

        {ADD DIMENSION SIZE}
        new(this);
        this^.d := start;
        this^.next := NIL;
        last^.next := this;
        last := this;
      End;
    until (doline = NIL);

    {REMAINDER OF LINE}
    doline := contline;
    contline := NIL;

    {INCLUDE IN VARIABLE TYPE}
    variables[I].dimension := root^.next;    {1st IS DUMMY}
    dispose(root);                           {DISPOSE DUMMY}
  End;
End;

{-------------------------------------------------------------------------}
{ This procedure handle common block definitions.                         }
{                                                                         }
Procedure Handle_Common;
Var CB        : Integer;
    CV        : Word;
    This, Old : IntPtr;
Begin
  {COMMON BLOCK DEFINITIONS}
  Remove_Spaces(DoLine);
  If (Copy(DoLine^.Line,1,1)='/') Then Begin {GET COMMON BLOCK NAME}
    Inc(NBlock);
    CB := NBlock;
    CBlock[NBlock].Name := GetString;
  End Else cb := -1;

  {ADD REFERENCES}
  Token := GetString;  New(Old);  CBlock[CB].Refs := Old;
  While (Token <> '') Do Begin
    add_definition(token,unkntype,cv);
    variables[cv].common := cb;
    variables[cv].usage := (variables[cv].usage OR combit);

    {ADD TO LIST}
    new(this);
    this^.v := addr(variables[cv]);
```

```
    this^.next := NIL;
    old^.next := this;
    old := this;

    {NEXT TOKEN}
    token := getstring;
  End;

  Old := CBlock[CB].Refs;
  CBlock[CB].Refs := Old^.Next;
  Dispose(Old);
End;

{-------------------------------------------------------------------------}
{ This procedure is used to skip to the next line.                        }
{                                                                         }
Procedure Kill_Data;
Var TmpLine : LinePtr;
Begin
  While (DoLine <> NIL) DO Begin

{$IFDEF Debug}
writeln('KILLING DATA "',DoLine^.Line,'"');
{$ENDIF}
    tmpline := doline;
    doline := tmpline^.next;
    dispose(tmpline);
  End;
  ReadLine;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to handle the data initialisation statements.    }
{                                                                         }
Procedure Handle_Data;
Var Token   : Strings;
    I, L, Q : Word;
    TmpLine : LinePtr;
    Done    : Boolean;
Begin
  Token := GetString;
  While (Token <> '') DO Begin
    If (Token[Length(Token)] = '=') Then Delete(Token,Length(Token),1);
    SetVariable(Token,AssBit);
    Remove_Spaces(DoLine);
    If (DoLine <> NIL) AND (DoLine^.Line[1] = '/') Then Begin
      {Remove data initialisation}
      Delete(DoLine^.Line,1,1);

      Q := 0;   Done := FALSE;
      Repeat
        i := 1;   l := length(doline^.line);
        while not done AND (i <= l) do begin
          If (DoLine^.Line[I] = '''') Then Begin Inc(Q);  Inc(I); End Else
          If (DoLine^.Line[I] = '/') Then Done := TRUE
          Else Inc(I);
        end;
        if not done OR (doline^.line = '') then begin
          {NEXT LINE}
          TmpLine := DoLine;
          DoLine := DoLine^.Next;
          Dispose(TmpLine);
        end else delete(doline^.line,1,i);
      Until (((Q MOD 2) = 0) AND Done) OR (DoLine = NIL);

      If Not Done Then Begin
        Writeln(Out,'ERROR: Error in data initialisation statement at line ',LineNo);
        Writeln(Out,'ERROR: Ignoring initialisation and continuing.');
      End;
    End;
```

```
    Token := GetString;
  End;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to handle the main name definitions.             }
{                                                                         }
Procedure Handle_Main(Tipe : Byte);
Var PNo, I      : Word;
    This, Last : IntPtr;

   Procedure AddPar(No : Word);
   Begin
     New(This);
     This^.Next := NIL;
     This^.V := Addr(Variables[No]);
     Last^.Next := This;
     Last := This;
   End;

Begin
  MainName := GetString;
  Write(' ':60-Length(MainName),MainName);
  For I := 1 To 60 Do Write(^H);
  MainType := Tipe;

  Last := MainPars;
  While (Last^.Next <> NIL) Do Last := Last^.Next;

  Token := GetString;  PNo := 0;
  While (Token <> '') Do Begin
    {ALL variables should NOT be declared and hence are simply added}
    I := FindVariable(Token);
    If (I = 0) Then Begin
      Inc(PNo);
      AddVariable(Token,UnKnType,ParBit,PNo);
      AddPar(NVars);
    End Else With Variables[I] Do Begin
      Usage := Usage OR ParBit;
      Common := PNo;
      Inc(PNo);
      AddPar(I);
    End;
    Token := GetString;
  End;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to handle the variable name definitions.         }
{                                                                         }
Procedure Handle_Def(Tipe : Byte);
Var Token : Strings;
    I     : Word;
Begin
  Token := GetString;
  If (Token = 'FUNCTION') Then Handle_Main(Tipe) Else
  While (Token <> '') Do Begin
    Add_Definition(token,tipe,i);

    Token := GetString;
  End;
End;

{-------------------------------------------------------------------------}
{ This procedure is used to handle a DO ?=?,?[,?] statement.              }
{                                                                         }
Procedure Handle_Do;
Var Token : Strings;
    Dummy : Boolean;
Begin
  {Loop Index}
```

```
  Token := GetString;
  If (Token[Length(Token)] = '=') Then Delete(Token,Length(Token),1) Else Begin
    Writeln('ERROR: Index variable expected.');
    Writeln('ERROR: Attempting to continue.');
  End;

  RefVariable(Token,AssBit,Dummy);

  {The remaining tokens are referenced}
  Token := GetString;
  While (Token <> '') Do Begin
    refvariable(token,refbit,dummy);
    token := getstring;
  End;
End;

{-------------------------------------------------------------------------}
{ This procedure masks out the IO parameter references.                   }
{                                                                         }
Procedure Handle_IO_Refs(Ref : Word);
Var I, NIter : Word;
    Token    : Strings;
    Dummy    : Boolean;
Begin
  {REMOVE FORMAT AND FILE REFERENCE}
  Repeat
    I := Pos(')',DoLine^.Line);
    If (I=0) Then Begin
      readline;
    End;
  Until (I <> 0) OR (EOF(InFile));

  If (I <> 0) Then Begin
     {THE ACTUAL REMOVAL}
     delete(doline^.line,1,i);

     token := getstring;
     while (token <> '') do begin
       If Token[Length(Token)] = '=' Then Begin
         Delete(Token,Length(Token),1);
         RefVariable(Token,Ref+AssBit,Dummy);
       End Else RefVariable(Token,Ref,Dummy);
       Token := GetString;
     end;
  End;
End;

{-------------------------------------------------------------------------}
{ This is the main calling procedure which builds the SAP analyse file.   }
{                                                                         }
Procedure Analyse(InName          : BigString;
                  Var Root        : ListPtr;
                  Var ENo         : Word);
Var NewData : ListPtr;
    Varble  : Variable;
    Comn    : CBPtr;
    RPtr    : RefPtr;
    CnstPtr : ConstantPtr;
    Flag    : Boolean;
Begin
  {Check for input files existence}
  Assign(InFile,InName);
  GetMem(InBuf,20000);
  SetTextBuf(InFile,InBuf^,20000);
  {$I-}  Reset(InFile);   {$I+}
  If IOResult <> 0 Then Begin
    ENo := 1;
    FreeMem(InBuf,20000);
    Exit;
  End;
  LineNum := 0;
```

```
  Variables[0].Name := '';   Variables[0].Next := NIL;

  Writeln('Analysing   "',InName,'"');
  If LogFlag Then Writeln(Out,'Analysing   "',InName,'"');
  Write(' At line :');
  Error := 0;

  DoLine := NIL;  ContLine := NIL;
  New(NextLine);  NextLine^.Line := '';  NextLine^.Next := NIL;

  Repeat
    {Initialise variables}
    MainName := '';          MainType := 255;
    New(MainPars);           MainPars^.Next := NIL;   {DUMMY 1st}
    NVars := 0;              CalledRefs := NIL;
    NBlock := 0;             Constants := NIL;
    Done := FALSE;

    TypeSets[IntType]  := ['I'..'N'];
    TypeSets[RealType] := ['A'..'H','O'..'Z'];
    TypeSets[CompType] := [];
    TypeSets[CharType] := [];
    TypeSets[LogType]  := [];
    TypeSets[DblRType] := [];

    {Simple test}
{$IFDEF Debug}
writeln('                                                ',LineNo:5,' NEXTLINE');
{$ENDIF}


    {*** PROCESS DEFINITION PORTIONS START ***}
    Repeat
      repeat token := getstring; until token <> '';

      Write(lineno:8,^H^H^H^H^H^H^H^H);

      if (token = 'PROGRAM') then begin
        {PROGRAM DEFINITION}
        MainName := GetString;
        MainType := ProgType;

        {Not interested in the rest}
        Repeat Until (GetString = '');
      end else if (token = 'SUBROUTINE') then begin
        {SUBROUTINE DEFINITION}
        Handle_Main(SubrType);
      end else if (token = 'FUNCTION') then begin
        {SUBROUTINE DEFINITION}
        Handle_Main(FuncType);
      end else if (token = 'COMMON') then begin
        Handle_Common;
      end else if (token = 'EXTERNAL') then begin
        {EXTERNAL REFERENCES}
        Repeat
          token := getstring;
          if (token <> '') then SetVariable(Token,ExtBit);
        Until (Token = '');
      end else if (token = 'PARAMETER') then begin
        {PARAMETER CONSTANTS}
        Repeat
          token := getstring;
          delete(token,length(token),1);    {Remove "="}
          remove_spaces(doline);
          if (token <> '') then Begin (*TBD*)  {HANDLE COMPLEX ?}
            {IF LAST CHARACTER ')', DELETE}
            If (DoLine <> NIL) AND (DoLine^.Next = NIL) Then
            If (DoLine^.Line[Length(DoLine^.Line)] = ')') Then Begin
              delete(doline^.line,length(doline^.line),1);
            End;
```

```
      SetVariable(Token,ConBit);
      New(CnstPtr);
      CnstPtr^.V := Addr(Variables[FindVariable(Token)]);
      CnstPtr^.Val := Evaluate(Flag);
      CnstPtr^.Next := Constants;
      Constants := CnstPtr;
      If Flag Then Begin
        Writeln(Out,'ERROR: Constant for "',Token,'" invalid.');
        Writeln(Out,'ERROR: Attempting to continue.');
      End;
    End;
  Until (Token = '');
end else if (token = 'EQUIVALENCE') then begin
  {EQUIVALENCE DEFINITIONS}
  Repeat
    token := getstring;
    if (token <> '') then SetVariable(Token,NoBit);
  Until (Token = '');
end else if (token = 'IMPLICIT') then begin
  {NAMING CONSTANTS}
  token := getstring;
  if (token = 'NONE') then begin
    For I := 1 To 6 Do TypeSets[I] := [];
    repeat until (getstring='');
  end else if (token = 'UNDEFINED') then begin
    GetSet(AddSet);
    For I := 1 To 6 Do typesets[i] := typesets[i] - addset;
    repeat until (getstring='');
  end else repeat
    If (Token = 'REAL') Then Begin
      If (Copy(DoLine^.Line,1,2) = '*8')
        Then FixSets(DblRType)
        Else FixSets(RealType);
    End Else If (Token = 'INTEGER') Then Begin
      If (Copy(DoLine^.Line,1,2) = '*8')
        Then FixSets(DblIType)
        Else FixSets(IntType);
    End Else If (Token = 'LOGICAL') Then Begin
      FixSets(LogType);
    End Else If (Token = 'CHARACTER') Then Begin
      FixSets(CharType);
    End Else If (Token = 'COMPLEX') Then Begin
      FixSets(CompType);
    End Else If (Token = 'DOUBLE') Then Begin
      token := getstring;   {SKIP PRECISION}
      FixSets(DblRType);
    End Else Begin
      writeln(Out,'ERROR: UNRECOGNISED "',token,'" in IMPLICIT STATEMENT');
      Error := 2;
      Exit;
    End;

    Token := GetString;
  until (token = '');
end else if (token = 'DIMENSION') then begin
  {UNKNOWN DEFS}
  Handle_Def(UnKnType);
end else if (token = 'COMPLEX') then begin
  {COMPLEX DEFS}
  Handle_Def(CompType);
end else if (token = 'INTEGER') then begin
  {INTEGER DEFS}
  If (Copy(DoLine^.Line,1,2) = '*8')
   Then B := DblIType
   Else B := IntType;
  Handle_Def(B);
end else if (token = 'REAL') then begin
  {REAL DEFS}
  If (Copy(DoLine^.Line,1,2) = '*8')
   Then B := DblRType
   Else B := RealType;
```

```
      Handle_Def(B);
    end else if (token = 'DOUBLE') then begin
      Token := GetString;

      {REAL DEFS}
      If (Token <> '') Then
      If (Token = 'PRECISION') Then B := DblRType
      Else Begin
        writeln(out,'ERROR: DOUBLE ',Token,' not recognised');
        writeln(out,'       Added ',token,' to variable list');
        AddVariable(Token,UnKnType,0,0);
        B := UnKnType;
      End;

      Handle_Def(B);
    end else if (token = 'CHARACTER') then begin
      {CHARACTER DEFS}
      Handle_Def(CharType);
    end else if (token = 'LOGICAL') then begin
      {LOGICAL DEFS}
      Handle_Def(LogType);
    end else if (token = 'DATA') then begin
      {DATA STATEMENT}
      Handle_Data;
    end else if (token = 'BLOCK') then begin
      {BLOCK DATA}
      token := getstring;
      if (token = 'DATA') Then Begin
        MainName := 'BLOCK DATA';
        MainType := BlckType;
      End Else Begin
        Writeln(Out,'ERROR: ILLEGAL BLOCK DATA ENCOUNTERED AT LINE ',LineNo);
        Writeln(Out,'ERROR: ATTEMPTING RECOVER');
        MainName := 'BLOCK DATA';
        MainType := BlckType;
        while (token <> '') do token := getstring;
      End;
    end else if (token = 'INCLUDE') then begin
      {INCLUDE STATEMENT}
      Kill_Data;
    end else if (token = '') then begin
      Writeln(Out,'ERROR: "END" Expected before the end of the file');
      Writeln(Out,'ERROR: ATTEMPTING RECOVER');
      Done := TRUE;
    end else done := TRUE;
  Until Done;
  {*** PROCESS DEFINITION PORTIONS END ***}

  {*** PROCESS BODY START ***}
  Done := FALSE;
  Repeat
    if (token = 'IF') OR (token = 'ELSEIF') then begin
      {IF REFERENCES}
      Handle_If;
    end else if (token = 'CALL') then begin
      {CALL REFERENCES}
      Handle_Call;
    end else if (token = 'END') then begin
      {END SUBROUTINE/FUNCTION/PROGRAM DEFINITION, DUMP DATA TO FILE}
      token := getstring;
      if (token <> 'IF') AND (token <> 'DO') Then Done := TRUE
      Else While (Token <> '') Do Token := GetString;
    end else if (token = 'DO') then begin
      {CHECK FOR 'DO WHILE' STATEMENT}
      Token := GetString;
      If (Token = 'WHILE') Then Handle_If Else
       begin
         replace_token(token);
         handle_do;
       end;
    end else if (token = 'READ') then begin
```

```
    {READ STATEMENT}
    Handle_IO_Refs(InBit);
  end else if (token = 'WRITE') then begin
    {WRITE STATEMENT}
    Handle_IO_Refs(OutBit);
  end else if (token = 'PRINT') then begin
    {PRINT STATEMENT}
    Kill_Data;
  end else if (token = 'FORMAT') then begin
    {FORMAT STATEMENT}
    Kill_Data;
  end else if (token = 'ELSE') then begin
    {ELSE STATEMENT}
    if (getstring = 'IF') Then Handle_If;
  end else if token_reserved(token) then begin
    {RESERVED TOKEN}
    repeat until (getstring = '');
  end else if (token = '') then begin
    Writeln('Warning: Empty token');
    Done := EOF(InFile);
  end else begin
    {NORMAL ASSIGNMENTS}
    If (Token[Length(Token)] = '=') Then Begin
      Delete(Token,Length(Token),1);
      If (Copy(DoLine^.Line,1,1) = '(') Then add_array_refs;
      RefVariable(Token,AssBit,Flag);
    End Else Begin
      writeln(out,'ERROR: UNIDENTIFIED TOKEN "',Token,'"');
      writeln(out,'LINE : ',LineNo:6);
      show('Line',DoLine);
      ENo := 2;
      Exit;
    End;

    {REFERENCED VARIABLES}
    Token := GetString;
    While (Token <> '') Do Begin
      RefVariable(Token,RefBit,Flag);
      Token := GetString;
    End;
  end;

  token := getstring;

  Write(lineno:8,^H^H^H^H^H^H^H^H);
Until Done;
{*** PROCESS BODY END ***}

{REPLACE TOKEN}
Replace_Token(Token);

{DELETE CONSTANT DECLARATIONS}
While (Constants <> NIL) Do Begin
  cnstptr := constants;
  constants := constants^.next;
  dispose(cnstptr);
End;

{SET THE DATA TYPES}
For I := 1 to NVars Do With Variables[I] Do Begin
  if (tipe = 0) then begin  {TRY AND SET THE TYPE}
     Done := TRUE; J := 1;
     While (J <= 6) AND Done Do Begin
       If (Name[1] IN TypeSets[J]) Then Begin
         tipe := j + impltype;
         done := FALSE;
       End Else Inc(J);
     End;
  end;
End;
```

```
{STORE THE DATA OF THIS PROGRAM/SUBROUTINE/FUNCTION}
New(NewData);
With NewData^ Do Begin
  {Patch root on the end (New item on list)}
  next := root;

  {set the main name}
  name := mainname;
  fyl  := inname;
  tipe := maintype;
  clist := NIL;

  {build the list of variables}
  vars := NIL;    numv := nvars;
  for i := nvars downto 1 do begin
    {CREATE VARIABLE, ADDING ON TOP}
    New(Varble);
    Varble^ := Variables[I];
    Varble^.Next := Vars;
    Vars := Varble;

    {SET VARIABLES NEXT TO THIS DUPLICATE RECORD}
    Variables[I].Next := Varble;
  end;

  {set the parameter list, fixing to point to the new var list}
  cbrec := mainpars;    mainpars := cbrec^.next;     dispose(cbrec); {dummy}
  params := mainpars;
  while (mainpars <> NIL) do begin
    MainPars^.V := MainPars^.V^.Next;
    MainPars := MainPars^.Next;
  end;

  {create the common block list, fixing to point to new var list}
  common := NIL;   numb := nblock;
  for i := nblock downto 1 do begin
    New(Comn);
    Comn^ := CBlock[I];
    Comn^.Next := Common;
    Common := Comn;


    {correct variable list}
    CBRec := CBlock[I].Refs;
    While (CBRec <> NIL) Do Begin
      cbrec^.v := cbrec^.v^.next;
      cbrec := cbrec^.next;
    End;
  end;

  {set the calls and their references, fixing to point to new var list}
  calls := calledrefs;
  while (calledrefs <> NIL) do begin
    RPtr := CalledRefs^.Refs;
    While (RPtr <> NIL) Do Begin
      rptr^.v := rptr^.v^.next;
      rptr := rptr^.next;
    End;
    CalledRefs := CalledRefs^.Next
  end;

  {sort the variables}
  Sort_Variables;

  {Adjust the linked list to be sorted as well}
  vars := variables[1].next;          {SET THE ROOT}
  for i := 1 to pred(nvars) do begin
    Variables[I].Next^.Next := Variables[Succ(I)].Next;
  end;
  variables[nvars].next^.next := NIL;  {LAST ELEMENT}
End;
```

```
    {New item in list}
    Root := NewData;


{$IFDEF Debug}
writeln;
writeln('================================================================================');
writeln;
{$ENDIF}

  Until EOF(InFile);

  Write(^M);  ClrEol;
  Writeln('Analysis of "',InName,'" complete.');
  Writeln('  => ',LineNo:6,' lines processed.');
  Writeln;

  If LogFlag Then Begin
    Writeln(Out,'Analysis of "',InName,'" complete.');
    Writeln(Out,'  => ',LineNo:6,' lines processed.');
    Writeln;
  End;

  {Finished, Close input & output file}
  Close(InFile);
  FreeMem(InBuf,20000);

  ENo := Error;
End;

End.
```

"SAPFXREF.PAS"

```
Unit SAPFXRef;

Interface

Uses Crt, SAPFConstants;

Procedure CrossReference(Var List : ListPtr);

Implementation

Var  DoneList, FrontList, BackList : ListPtr;

{-------------------------------------------------------------------------}
{ This procedure displays a brief message and then waits for a keypress.  }
{                                                                         }
Procedure Pause;
Var Ch : Char;
Begin
  If Not LogFlag Then Begin
    write('                    Press any key to continue');
    ch := readkey;
    write(^M); clreol;
    writeln;
  End;
  Writeln(Out);
End;

{-------------------------------------------------------------------------}
{ This function return the uppercase if the lowercase or the lowercase if  }
{ the uppercase was passed.                                               }
{                                                                         }
Function Toggle(Ch : Char) : Char;
Begin
  If (Ch >= 'A') AND (Ch <= 'Z')
   Then Ch := Chr(Ord(Ch) + 32)
   Else Ch := Upcase(Ch);
  Toggle := Ch;
End;

{-------------------------------------------------------------------------}
{ This function returns the string representing the type passed.          }
{                                                                         }
Function TypeStr(Tipe : Byte) : Strings;
Var Result : Strings;
Begin
  tipe := tipe MOD impltype;
  case tipe of
   0   : Result := 'UNKNOWN';
   1   : Result := 'INTEGER';
   2   : Result := 'REAL';
   3   : Result := 'COMPLEX';
   4   : Result := 'CHARACTER';
   5   : Result := 'LOGICAL';
   6   : Result := 'LONG INTEGER';
   7   : Result := 'DOUBLE PRECISION REAL';
   98  : Result := 'ALL TYPES';
   99  : Result := 'COMMON BLOCK';
   124 : Result := 'BLOCK DATA';
   125 : Result := 'FUNCTION';
   126 : Result := 'SUBROUTINE';
   127 : Result := 'PROGRAM';
  else result := 'UNKNOWN';
  end;
  TypeStr := Result;
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure allows user input of a variable type.                    }
{                                                                         }
Procedure Get_Type(Var Tipe : Byte);
Var X, Y : Byte;
    Ch   : Char;
Begin
  x:= wherex;  y := wherey;
  writeln('ENTER TYPE (SIRCHLNEA) ');
  write('Subroutine  Integer  Real  Complex  cHaracter  Logical  iNteger*8  rEal*8  Any');
  repeat
    ch := upcase(readkey);
    tipe := pos(ch,'IRCHLNESA');
  until (tipe > 0);
  gotoxy(x,y);  write('TYPE = '); highvideo; clreol;
  if tipe = 8 then tipe := subrtype else
  if tipe = 9 then tipe := anytype;
  write(typestr(tipe));
  lowvideo;   writeln;   clreol;
End;

{-------------------------------------------------------------------------}
{ This procedure shows the portion of the file containing the call passed on }
{ the screen.                                                             }
{                                                                         }
Procedure ShowFile(FName : BigString; Line : Word; Call : Strings);
Var Line1, Line2 : BigString;
    F            : Text;
    I            : Word;
    Cont, Done   : Boolean;
    InBuf        : Pointer;
Begin
  If NOLFlag Then Exit;

  Writeln(Out,'================================================================================');
  Assign(F,FName);
  GetMem(InBuf,20000);
  SetTextBuf(F,InBuf^,20000);
  Cont := TRUE;
  {$I-}  Reset(F);  {$I+}
  If (IOResult <> 0) Then Begin
    writeln(Out,'ERROR: File "',FName,'" has disappeared.');
    writeln(Out,'ERROR: IOResult = ',I);
    cont := FALSE;
  End;

  Writeln(Out,'File : ',FName);
  Writeln(Out,'Line : ',Line);

  If Cont Then Begin
    {GET TO THE LINE}
    while (line > 1) do begin
      Dec(Line);
      Readln(F);
    end;

    {SHOULD BE AT THE LINE, OTHERWISE ERROR.}
    If EOF(F) Then Begin
      writeln(out,'ERROR: File "',FName,'" has changed.');
      cont := FALSE;
    End;
  End;

  If Cont Then Begin
    {READ IN THE LINE AND CHECK.}
    readln(f,line1);
    i := pos(#9,line1);
    while (i <> 0) do begin
      line1 := copy(line1,1,pred(i)) + '      ' + copy(line1,succ(i),255);
      i := pos(#9,line1);
    end;
```

```
    Repeat
      line2 := line1;
      for i := 1 to length(line2) do line2[i] := upcase(line2[i]);
      i := pos(call,line2);
      if (i = 0) then i := length(line1)+1;

      {Display the normal portion}
      write(out,copy(line1,1,pred(i)));

      {Display the call}
      highvideo;
      write(out,copy(line1,i,length(call)));
      lowvideo;

      {Display the rest}
      writeln(out,copy(line1,i+length(call),255));

      done := TRUE;
      readln(f,line1);
      i := pos(#9,line1);
      while (i <> 0) do begin
        line1 := copy(line1,1,pred(i)) + '      ' + copy(line1,succ(i),255);
        i := pos(#9,line1);
      end;

      if (upcase(line1[1]) <> 'C') AND (copy(line1,6,1) <> ' ')
        then done := FALSE;
    Until Done OR EOF(F);
  End;

  If Not Cont Then Begin
    writeln(out,'ERROR: Cannot display portion of file required !!!');
  End;
  Writeln(out,'================================================================================');
  Close(F);
  FreeMem(InBuf,20000);
End;

{-------------------------------------------------------------------------}
{ This procedure shows the portion of the file containing the call passed on }
{ the screen.                                                               }
{                                                                           }
Procedure ShowCrtFile(FName : BigString; Line : Word; Call : Strings);
Var Line1, Line2 : BigString;
    F            : Text;
    I, L, BC     : Word;
    Cont, Done   : Boolean;
Begin
  Writeln('================================================================================');
  Assign(F,FName);
  Cont := TRUE;
  {$I-}  Reset(F);  {$I+}
  If (IOResult <> 0) Then Begin
    writeln('ERROR: File "',FName,'" has disappeared.');
    writeln('ERROR: IOResult = ',I);
    cont := FALSE;
  End;

  Writeln('File : ',FName);
  Writeln('Line : ',Line);
```

```
  If Cont Then Begin
    {GET TO THE LINE}
    while (line > 1) do begin
      Dec(Line);
      Readln(F);
    end;

    {SHOULD BE AT THE LINE, OTHERWISE ERROR.}
    If EOF(F) Then Begin
      writeln('ERROR: File "',FName,'" has changed.');
      cont := FALSE;
    End;
  End;

  If Cont Then Begin
    {READ IN THE LINE AND CHECK.}
    readln(f,line1);
    i := pos(#9,line1);
    while (i <> 0) do begin
      line1 := copy(line1,1,pred(i)) + '      ' + copy(line1,succ(i),255);
      i := pos(#9,line1);
    end;

    line2 := line1;
    for i := 1 to length(line2) do line2[i] := upcase(line2[i]);

    i := pos(call,line2);
    if (i = 0) then begin
      writeln('ERROR: Cannot locate "',call,'" in "',FName,'"');
    end;
  End;

  If Cont Then Begin
    {Display the normal portion}
    write(copy(line1,1,pred(i)));

    {Display the call}
    bc := 0;
    highvideo;
    repeat
      l := length(line1);
      repeat
        write(line1[i]);
        if (line1[i]) = '(' then inc(bc) else
        if (line1[i]) = ')' then dec(bc);
        inc(i);
      until (i > l) OR ((bc=0) AND (line1[pred(i)] = ')'));

      if (bc = 0) AND (line1[pred(i)] = ')') then normvideo;
      writeln(copy(line1,i,255));
      readln(f,line1);
      i := pos(#9,line1);
      while (i <> 0) do begin
        line1 := copy(line1,1,pred(i)) + '      ' + copy(line1,succ(i),255);
        i := pos(#9,line1);
      end;
      i := 1;
      done := TRUE;
      if (upcase(line1[1]) = 'C') then done := FALSE else
      if (copy(line1,6,1) <> ' ') then done := FALSE;
    until done OR eof(F);
  End;

  If Not Cont Then Begin
    writeln('ERROR: Cannot display portion of file required !!!');
  End;
  Writeln('================================================================================');
  Close(F);
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure is used to sort the variables passed. It allows up to 1000  }
{ variables to be sorted.                                                 }
{                                                                         }
Procedure SortVars(Var List : Variable);
Var Arr          : Array[1..1000] Of Variable;
    N, I, NVars  : Word;
    NoSwap       : Boolean;
    Temp         : Variable;
Begin
  If (List <> NIL) AND (List^.Next <> NIL) Then Begin
    {PLACE VARAIABLES IN ARRAY}
    NVars := 0;
    While (List <> NIL) Do Begin
      inc(nvars);
      arr[nvars] := list;
      list := list^.next;
    End;

    {SORT THE ARRAY}
    N := Pred(NVars);
    Repeat
      NoSwap := TRUE;
      For I := 1 To N Do
       if (Arr[I]^.Name > Arr[Succ(I)]^.Name) Then begin
         Temp := Arr[I];
         Arr[I] := Arr[Succ(I)];
         Arr[Succ(I)] := Temp;
         NoSwap := FALSE;
       end;
      Dec(N);
    Until NoSwap;

    {PLACE THE ARRAY BACK INTO A LINKED LIST}
    List := Arr[1];
    For I := 1 To Pred(NVars) Do  Arr[I]^.Next := Arr[Succ(I)];
    Arr[NVars]^.Next := NIL;
  End;
End;

{-------------------------------------------------------------------------}
{ Procedure to give a simple warning message.                             }
{                                                                         }
Procedure Warn;
Begin
  write(out,'WARNING: ');
End;

{-------------------------------------------------------------------------}
{ Function to return a string with the type.                              }
{                                                                         }
Function ShowProc(I : ListPtr) : BigString;
Var St : Strings;
Begin
  If I^.Tipe < 100 Then St := ' FUNCTION ' Else St := ' ';
  ShowProc := TypeStr(I^.Tipe) + St + I^.Name;
End;

{-------------------------------------------------------------------------}
{ This procedure searches the list passed for the name occurance.         }
{                                                                         }
Function InList(Name : Strings; List : ListPtr) : ListPtr;
Begin
  While (Name <> List^.Name) AND (List <> NIL) Do  List := List^.Next;
  InList := List;
End;
```

```
{-------------------------------------------------------------------------}
{ This procedure checks the variable types and dimensions when passed in  }
{ either a procedure or a common block.                                   }
{                                                                         }
Procedure CheckVariables(V1 : Variable; Var I1 : ListPtr;
                         V2 : Variable; Var I2 : ListPtr;
                         Var Fyl : BigString;
                         Line : Word;
                         PCnt : Word;
                         Flag : Boolean;
                         CB   : Strings);

  Procedure ShowDim(V : Variable; I : ListPtr);
  Var D : DimPtr;
  Begin
    Warn; Write(Out,'"',V^.Name,'" of ',ShowProc(I),' has size ');
    D := V^.Dimension;
    While (D <> NIL) Do Begin
      If (D^.D = MinInt) Then Write(Out,'*') Else Write(Out,D^.D);
      If(D^.Next <> NIL) Then Write(Out,',');
      D := D^.Next;
    End;
    Writeln(Out,'.');
  End;


  Procedure ErrorHead(Message : BigString);
  Begin
    Warn;
    If (cb = '') Then write(out,'Parameter') Else Begin
      write(out,'Common block ',cb,' error, variable');
    End;
    Writeln(Out,' #',PCnt,' ',Message);
  End;

Var
  Dim1, Dim2                    : DimPtr;
  DCnt                          : Word;
  Result                        : Boolean;
Begin
  Result := TRUE;

  {CHECK PARAMETER TYPES}
  If ((V1^.Tipe MOD ImplType) <> (V2^.Tipe MOD ImplType)) AND
     (V1^.Tipe <> AnyType) AND
     (V2^.Tipe <> AnyType) Then Begin
    errorhead('types do not match.');
    showfile(fyl,line,V1^.Name);
    warn; writeln(out,'"',V1^.Name,'" has type ',typestr(V1^.Tipe),' in ',ShowProc(I1),'.');
    warn; writeln(out,'"',V2^.Name,'" has type ',typestr(V2^.Tipe),' in ',ShowProc(I2),'.');
    warn; writeln(out,ShowProc(I1),' calls ',ShowProc(I2),'.');
    pause;
  End;

  {CHECK PARAMETER DIMENSIONS}
  Dim2 := V2^.Dimension;
  If Flag Then Dim1 := NIL Else
  If (V1^.Tipe = AnyType) Then Dim1 := Dim2  Else Dim1 := V1^.Dimension;
  If (V2^.Tipe = AnyType) Then Dim2 := Dim1;

  DCnt := 0;
  If (Dim1 <> NIL) OR (Dim2 <> NIL) Then
  If (Dim1 <> NIL) AND (Dim2 = NIL) Then Begin
    ErrorHead('dimension mismatch');
    Warn; Writeln(Out,'Dimensioned variable "',V1^.Name,'" is passed into undimensioned variable');
    Warn; Writeln(Out,'"',V2^.Name,'" of ',ShowProc(I2),'.');
    ShowDim(V1,I1);
    ShowFile(Fyl,Line,V1^.Name);
    Pause;
  End Else If (Dim1 = NIL) AND (Dim2 <> NIL) Then Begin
    ErrorHead('dimension mismatch');
```

```
    Warn; Writeln(Out,'Undimensioned variable "',V1^.Name,'" of ',ShowProc(I1),' is passed');
    Warn; Writeln(Out,'into "',V2^.Name,'" which is dimensioned.');
    ShowDim(V2,I2);
    ShowFile(Fyl,Line,V1^.Name);
    Pause;
  End Else Begin
    Repeat
      inc(dcnt);
      if (dim1^.d <> minint) AND (dim2^.d <> minint) AND
         (dim1^.d <> dim2^.d) then begin
        ErrorHead('dimension sizes do not match.');
        Warn;  Writeln(Out,'Dimension # ',DCnt);
        ShowDim(V1,I1);
        ShowDim(V2,I2);
        ShowFile(Fyl,Line,V1^.Name);
        Pause;
      end;
      dim1 := dim1^.next;    dim2 := dim2^.next;
    Until (Dim1 = NIL) OR (Dim2 = NIL);

    If (Dim1 <> NIL) Then Begin
      ErrorHead('number of dimensions mismatch.');
      Warn; Writeln(Out,'Variable "',V1^.Name,'" has more dimensions than "',V2^.Name,'"');
      ShowDim(V1,I1);
      ShowDim(V2,I2);
      Warn; Writeln(Out,'Call at line ',Line,' in file "',Fyl,'".');
      Pause;
    End Else If (Dim2 <> NIL) Then Begin
      ErrorHead('number of dimensions mismatch.');
      Warn; Writeln(Out,'Variable "',V1^.Name,'" has more dimensions than "',V2^.Name,'"');
      ShowDim(V1,I1);
      ShowDim(V2,I2);
      Warn; Writeln(Out,'Call at line ',Line,' in file "',Fyl,'".');
      Pause;
    End;
  End;

End;

{-------------------------------------------------------------------------}
{ This function attempts to process the references to other procedures of  }
{ the function/procedure passed.                                           }
{                                                                          }
Function Processed(Item : ListPtr) : Boolean;
Var Result, Tmp8                : Boolean;
    Ref, NewItem                : ListPtr;
    TmpInt, LastInt             : IntPtr;
    TmpRef                      : RefPtr;
    VarItem                     : Variable;
    CallTmp                     : CallPtr;
    PtrC                        : CPtr;
    Com1, Com2                  : CBPtr;
    UI_NParams, I, J, PCnt      : Word;
    UI_Name                     : Strings;
    X,Y                         : Byte;
    RCh, ACh, ICh, OCh, Ch      : Char;
Begin
  Result := TRUE;
  With Item^ Do Begin
    calltmp := calls;
    while (CallTmp <> NIL) AND result do begin
      Ref := InList(CallTmp^.Name,DoneList);
      If (Ref = NIL) Then Begin
        {Reference not processed, cannot process this further normally}
        if inlist(CallTmp^.name,FrontList) <> NIL then Result := FALSE
        Else Begin
```

```
{CALL NOT INCLUDED IN LIST, CREATE IN LIST}
new(newitem);
with newitem^ do begin
  Name := CallTmp^.name;
  Fyl := 'USER INPUT';
  Params := NIL;
  Common := NIL;
  Vars := NIL;
  Calls := NIL;
  CList := NIL;
  NumV := 0;
  NumB := 0;
  Next := DoneList;
end;
donelist := newitem;

{DISPLAY INPUT REQUEST}
clrscr;
highvideo;
writeln('USER INPUT REQUIRED !!!',^G^G);
writeln('========================');
normvideo;
writeln('Reference to SUBROUTINE or FUNCTION not solved, call must be added manually.');
writeln('The call made appears below:');
showcrtfile(fyl,CallTmp^.line,CallTmp^.name);
write('CALL                : "'); highvideo;
write(CallTmp^.name); lowvideo;
writeln('"');

{GET THE TYPE}
get_type(newitem^.tipe);


{GET THE PARAMETERS.}
write('Number of parameters : ');   highvideo;
readln(ui_nparams);                 lowvideo;
newitem^.numv := ui_nparams;
for i := 1 to ui_nparams do begin
  X := WhereX;  Y := WhereY;
  Repeat
    GotoXY(X,Y);
    Write(I:2,') Name ? ');   HighVideo;
    ReadLn(UI_Name);          LowVideo;
  Until (UI_Name <> '');
  For J := 1 To Length(UI_Name) Do UI_Name[J] := Upcase(UI_Name[J]);

  rch := 'r';  ach := 'a';  ich := 'i';   och := 'o';
  Repeat
    GotoXY(X,Y);
    Write(I:2,') ');  HighVideo;
    Write(UI_Name);   LowVideo;
    Write(' ':31-Length(UI_Name),rch,ach,ich,och,'    => Change (RAIO) Q=Quit');
    ClrEOL;
    Repeat
      ch := upcase(readkey);
    Until (Ch IN ['R','A','I','O','Q']);
    Case Ch Of
     'R' : RCh := Toggle(RCh);
     'A' : ACh := Toggle(ACh);
     'I' : ICh := Toggle(ICh);
     'O' : OCH := Toggle(OCh);
    End;
  Until (Ch = 'Q');
  Write(^M,I:2,') ');  HighVideo;
  Write(UI_Name);   LowVideo;
  Write(' ':31-Length(UI_Name),rch,ach,ich,och,'=> ');
  ClrEOL;
```

```
      {ADD PARAMETER}
      New(VarItem);  New(TmpInt);
      VarItem^.Next := NewItem^.Vars;
      NewItem^.Vars := VarItem;
      If (NewItem^.Params = NIL)
       THEN NewItem^.Params := TmpInt
       ELSE LastInt^.Next := TmpInt;
      TmpInt^.Next := NIL;
      TmpInt^.V := VarItem;
      LastInt := TmpInt;

      With VarItem^ Do Begin
        name := UI_Name;
        common := i;
        get_type(tipe);
        usage := parbit;
        dimension := NIL;
        if rch = 'R' then usage := usage OR refbit;
        if ach = 'A' then usage := usage OR assbit;
        if ich = 'I' then usage := usage OR inbit;
        if och = 'O' then usage := usage OR outbit;
      End;
    end;

    sortvars(newitem^.vars);

    clrscr;
  End;
End Else Begin {Reference processed, do actual xref}
  {Remove call}
  calls := calltmp^.next;              lastint := ref^.params;
  pcnt := 0;

  {Check parameters}
  while (calltmp^.refs <> NIL) do begin
    TmpRef := CallTmp^.Refs;          {Get call parameter}
    Inc(PCnt);
    CallTmp^.Refs := TmpRef^.Next;    {Advance to next parameter}

    {Ensure parameters match}
    If (LastInt = NIL) Then Begin
      Dec(PCnt);
      Writeln(Out,'ERROR: To many parameters passed to call to "',calltmp^.name,'"');
      Writeln(Out,'ERROR: Only ',pcnt,' parameters expected.');
      Writeln(Out,'ERROR: At line number ',calltmp^.line,' in file "',fyl,'"');
      Pause;
    End Else Begin
      {CHECK IF PARAMETER TO BE PROCESSED}
      If (TmpRef^.V = NIL) Then Begin  {CHECK PARAMETER}
        if ((lastint^.v^.usage AND (AssBit OR InBit)) <> 0) then begin
          writeln(out,'WARNING: Parameter #',PCnt,' is passed to call "',calltmp^.name,'"');
          writeln(out,'WARNING: at line ',calltmp^.line,' in file "',Fyl,'"');
          writeln(out,'WARNING: as a constant, but the parameter is altered within the call.');
          pause;
        end;
      End Else Begin
        {CALL RESOLVED}
        TmpRef^.V^.Usage := TmpRef^.V^.Usage OR
                            (LastInt^.V^.Usage AND NOT ParBit);

        CheckVariables(TmpRef^.V,Item,
                       LastInt^.V,Ref,
                       Fyl,CallTmp^.Line,PCnt,TmpRef^.APosn,'');
      End;

      {Next Actual Parameter}
      LastInt := LastInt^.Next;
    End;

    Dispose(TmpRef);
  end;
```

```
{Ensure the parameter count is still the same}
if (lastint <> NIL) Then Begin
   Writeln(Out,'ERROR: Number of parameters to call "',calltmp^.name,'" do not match.');
   Writeln(Out,'ERROR: More than ',Pcnt,' parameters expected.');
   Writeln(Out,'ERROR: At line number ',calltmp^.line,' in file "',fyl,'"');
   Pause;
end;

{See if call is in unique call list}
ptrc := clist;
while (ptrc <> NIL) AND (ptrc^.c <> ref) do ptrc := ptrc^.next;

{If not in list, then add and do XREF}
if (ptrc = NIL) then begin
  New(PtrC);
  PtrC^.C := Ref;
  PtrC^.Next := CList;
  CList := PtrC;

  {XRef common blocks}
  Com1 := Common;
  While (Com1 <> NIL) Do Begin  {For all common blocks ....}
    {Find common block in REF}
    Com2 := Ref^.Common;
    While (Com2 <> NIL) AND (Com1^.Name <> Com2^.Name) Do
     com2 := com2^.next;

    If (Com2 <> NIL) Then Begin  {REF also uses, XREF}
      tmpint  := com1^.refs;   {ITEM common block}
      lastint := com2^.refs;   {REF common block}
      pcnt := 0;
      while (tmpint <> NIL) do begin
        {Ensure the number of vars in the common block is the same}
        If (LastInt = NIL) Then Begin
           Write(Out,'ERROR: Number of variables in the common block "',com1^.name);
           Writeln(Out,'" do not match');
           Writeln(Out,'ERROR: between SUBROUTINES/FUNCTIONS "',item^.name,'" and "',ref^.name,'"');
           Pause;
        End Else Begin
          Inc(PCnt);

          {XRef common block}
          TmpInt^.V^.Usage := (TmpInt^.V^.Usage OR LastInt^.V^.Usage);

          TmpB := NOLFlag;
          NOLFlag := TRUE;
          CheckVariables(TmpInt^.V,Item,
                         LastInt^.V,Ref,
                         Fyl,CallTmp^.Line,PCnt,FALSE,Com1^.Name);
          NOLFlag := TmpB;

          LastInt := LastInt^.Next;
        End;

        tmpint := tmpint^.next;
      end;

      {Ensure the number of vars in the common block is the same}
      if (lastint <> NIL) Then Begin
         Write(Out,'ERROR: Number of variables in the common block "',com1^.name);
         Writeln(Out,'" do not match');
         Writeln(Out,'ERROR: between SUBROUTINES/FUNCTIONS "',item^.name,'" and "',ref^.name,'"');
         Pause;
      end;
    End;

    Com1 := Com1^.Next;
  End;
end;
```

```
        {Return to pool}
        dispose(calltmp);
        calltmp := calls;
      End;
    end;
  End;
  Processed := Result;
End;

{-------------------------------------------------------------------------}
{ This procedure controls the cross references.                           }
{                                                                         }
Procedure CrossReference(Var List : ListPtr);
Var TempList, Marker : ListPtr;
    Vars             : Variable;
    RB, AB           : Word;
    RefFlag, AssFlag : Boolean;
Begin
  {ENSURE THERE IS WORK}
  FrontList := List;
  If (FrontList = NIL) Then Exit;

  Writeln;
  Writeln('Resolving externals');
  If LogFlag Then Begin
    Writeln(Out);
    Writeln(Out,'Resolving externals');
  End;

  {Resolve any externals}
  TempList := List;
  While (TempList <> NIL) Do Begin
    vars := templist^.vars;
    while (vars <> NIL) do begin
      If ((Vars^.Usage AND ExtBit) = ExtBit) Then Begin
        {External to resolve}
        marker := list;
        while (marker <> NIL) AND (marker^.name <> vars^.name) do
          marker := marker^.next;

        if (marker = NIL) then begin
          {Cannot resolve external}
          Writeln(Out,'WARNING: Cannot resolve external function/subroutine "',vars^.name,'"');
          Pause;
        end else begin
          If (Vars^.Tipe DIV ImplType) = 0 Then Begin
            {Check if types coincide}
            if (vars^.tipe <> marker^.tipe) then begin
              writeln(out,'ERROR: Declared type and actual type of "',marker^.name,'" do not match.');
              pause;
            end;
          End Else Begin
            {Reset type if a function}
            if (marker^.tipe < impltype) then vars^.tipe := marker^.tipe;
          End;
        end;
      End;
      Vars := Vars^.Next;
    end;

    templist := templist^.next;
  End;

  Write(^M,'Cross-referencing');  ClrEol;
  Writeln;
  If LogFlag Then Begin
    Writeln(Out,'Cross-referencing');
  End;
```

```
{Find the last item in the queue}
BackList := FrontList;
While (BackList^.Next <> NIL) Do BackList := BackList^.Next;

DoneList := NIL;   Marker := NIL;
While (FrontList <> NIL) Do Begin
  {REMOVE FROM QUEUE}
  Write('.');
  TempList := FrontList;
  FrontList := FrontList^.Next;
  TempList^.Next := NIL;

  If Processed(TempList) Then Begin
    {PUT ON DONE LIST}
    TempList^.Next := DoneList;
    DoneList := TempList;

    {RESET MARKER}
    Marker := NIL;
  End Else Begin
    {QUEUE EMPTY => RECURSION}
    If (FrontList = NIL) Then Begin
      writeln(out,'ERROR: Cannot process direct recursion on "',templist^.name,'"');
      writeln(out,'CROSS REFERENCE ABORTED !!!!!!!!!!!!!!!!!!!!!!!!');
      writeln(out);

      {REPLACE ON DONE STACK AND EXIT}
      templist^.next := donelist;
      list := templist;
      exit;
    End;

    {CHECK IF MARKER SET}
    If (Marker = NIL) Then Marker := TempList Else
    If (Marker = TempList) Then Begin
      {NOTHING HAS BEEN PROCESSED SINCE MARKER WAS QUEUED THE FIRST TIME,}
      {THUS THERE MUST BE INDIRECT RECURSION WHICH IS ILLEGAL.          }
      writeln(out,'ERROR: Indirect recursion encountered with the following calls');
      while (marker <> nil) do begin
        Writeln(out,'       ',Marker^.Name);
        Marker := Marker^.Next;
      end;
      writeln;
      writeln(out,'CROSS REFERENCE ABORTED !!!!!!!!!!!!!!!!!!!!!!!!');
      writeln(out);

      {PLACE QUEUED ITEMS ON DONE STACK AND EXIT}
      BackList^.Next := DoneList;
      TempList^.Next := FrontList;
      List := TempList;
      Exit;
    End;

    {PLACE ON BACK ON QUEUE}
    BackList^.Next := TempList;
    BackList := TempList;
    BackList^.Next := NIL;                    {Not necessary, but safe.}
  End;
End;
List := DoneList;
```

```
  {CHECK FOR UNINITIALISED VARIABLES}
  RB := RefBit OR OutBit OR ParBit OR ComBit OR CallBit OR ConBit;
  AB := AssBit OR InBit OR ParBit OR ComBit OR CallBit OR ConBit;
  While (DoneList <> NIL) Do Begin
    vars := donelist^.vars;
    while (vars <> NIL) do begin
      RefFlag := ((Vars^.Usage AND RB) <> 0);
      AssFlag := ((Vars^.Usage AND AB) <> 0);
      If RefFlag AND Not AssFlag Then Begin
        Warn; Writeln(Out,'Variable ',Vars^.Name,' referenced but never');
        Warn; Writeln(Out,'assigned in ',ShowProc(DoneList));
        Pause;
      End Else If (AssFlag AND Not RefFlag) AND (Vars^.Name <> DoneList^.Name) Then Begin
        Warn; Writeln(Out,'Variable ',Vars^.Name,' assigned but never');
        Warn; Writeln(Out,'referenced in ',ShowProc(DoneList));
        Pause;
      End Else If Not RefFlag AND Not AssFlag Then Begin
        Warn; Writeln(Out,'Variable ',Vars^.Name,' never assigned or');
        Warn; Writeln(Out,'referenced in ',ShowProc(DoneList));
        Pause;
      End;

      Vars := Vars^.Next;
    end;

    donelist := donelist^.next;
  End;
End;

End.
```

(ii) Relate Source

"RELATE.PAS"

```
Program Relate;

Uses Crt, Dos;

Type BigString = String[255];
     Strings   = String[30];

     Variable = ^VarRec;
     VarRec = Record
       Name  : Strings;
       Usage : Word;
       Next  : Variable;
     End;

Const
      NoBit   : Word = 0;
      RefBit  : Word = 1;
      AssBit  : Word = 2;
      ParBit  : Word = 4;
      ConBit  : Word = 16;
      ExtBit  : Word = 32;
      ComBit  : Word = 64;
      CallBit : Word = 128;
      OutBit  : Word = 256;
      InBit   : Word = 512;

Var InName1, InName2, Outname       : BigString;
    Main1, Main2                    : Strings;
    Out                             : Text;
    Root1, Root2, Last, This        : Variable;
    Buf                             : Pointer;
    Error                           : Boolean;

{-------------------------------------------------------------------------}
Procedure LoadFile(Var Root : Variable; Name : BigString;
                   Var Prog  : Strings;
                   Var Error : Boolean);
Var  Last, This : Variable;
     InFile     : Text;
     Line       : BigString;
     Buf        : Pointer;
     I          : Word;
     Found      : Boolean;
Begin
  New(Root);              {Dummy 1st}
  Root^.Next := NIL;
  Last := Root;

  Prog := '';  Error := FALSE;

  Assign(InFile,Name);
  GetMem(Buf,20000);
  SetTextBuf(InFile,Buf^,20000);
  {$I-}  Reset(InFile);   {$I+}
  If (IOResult <> 0) Then Error := TRUE  {FILE NOT FOUND}
  Else Begin
    Found := FALSE;

    {START THE SEARCH}
    While Not (EOF(InFile)) AND Not(Found) Do Begin
      repeat
        readln(InFile,line);
        found := copy(line,1,10) = '==========';
      until found or eof(InFile);

      if found and not eof(InFile) then begin
```

```
        readln(InFile,line);
        if copy(line,32,3) = '255' then begin
          Found := TRUE;
          Prog := Copy(Line,1,30);
        end else found := FALSE;
      end;
    End;

    {IF FOUND, START SEARCH FOR VARIABLES}
    If Found Then Begin
      found := FALSE;
      repeat
        Readln(InFile,Line);
        Line := Copy(Line,1,10);
        If Line = 'VARIABLES=' then found := TRUE;
      until found or eof(InFile) or (line = '==========');
    End Else Writeln('PROGRAM NOT FOUND');

    {IF VARIABLES LOCATED, READ THEM IN}
    If Found Then Begin
      Found := FALSE;
      Repeat
        Readln(InFile,Line);
        If (Copy(Line,1,10) = '==========') Then Found := TRUE Else Begin
          new(this);
          this^.name := copy(line,1,30);
          val(copy(line,51,4),this^.usage,i);
          if (i = 0) then begin
            This^.Next := NIL;
            Last^.Next := This;
            Last := This;
          end else dispose(this);
        End;
      Until Found Or Eof(InFile);
    End Else Writeln('PROGRAM VARIABLES NOT FOUND.');
  End;

  Close(InFile);
  FreeMem(Buf,20000);

  {KILL DUMMY 1st}
  Last := Root;
  Root := Root^.Next;
  Dispose(Last);
End;
{------------------------------------------------------------------------------}

Begin
  If ParamCount = 3 Then Begin
    InName1 := ParamStr(1);
    InName2 := ParamStr(2);
    OutName := ParamStr(3);
  End Else Begin
    ClrScr;
    Writeln('REFERENCE FILE GENERATOR FOR SAPF OUTPUT');
    Writeln('========================================');
    Writeln;
    Write('Enter 1st SAPF input file name :');
    Readln(InName1);
    If InName1 = '' Then Exit;

    Write('Enter 2nd SAPF input file name :');
    Readln(InName2);
    If InName2 = '' Then Exit;

    Write('Enter output file name         :');
    Readln(OutName);
    If outName = '' Then Exit;
  End;

  LoadFile(Root1,InName1,Main1,Error);
```

```
  If Error Then Begin
    writeln('Error with file "',inname1,'"');
    while root1 <> NIL do begin
      Last := Root1;
      Root1 := Root1^.Next;
      Dispose(Last);
    end;
    exit;
  End;

  LoadFile(Root2,InName2,Main2,Error);
  If Error Then Begin
    writeln('Error with file "',inname2,'"');
    while root1 <> NIL do begin
      Last := Root1;
      Root1 := Root1^.Next;
      Dispose(Last);
    end;
    while root2 <> NIL do begin
      Last := Root2;
      Root2 := Root2^.Next;
      Dispose(Last);
    end;
    exit;
  End;

  {DO The Output}
  Assign(Out,OutName);
  GetMem(Buf,20000);
  SetTextBuf(Out,Buf^,20000);
  Rewrite(Out);
  Writeln(Out,Main1,' transfer method   ',Main2);
  While (Root1 <> NIL) OR (Root2 <> NIL) Do Begin
    if (root2 = NIL) OR (root1^.name < root2^.name) then begin
      {ADVANCE NEXT ROOT1}
      Last := Root1;
      Root1 := Root1^.Next;
      Dispose(Last);
    end else if (root1 = NIL) OR (root1^.name > root2^.name) then begin
      {ADVANCE NEXT ROOT2}
      Last := Root2;
      Root2 := Root2^.Next;
      Dispose(Last);
    end else begin
      Write(Out,Root1^.Name,'   ');

      {EQUAL, SO CHECK FOR SEND METHOD}
      If ((Root1^.Usage AND (RefBit Or OutBit)) <> 0) AND
         ((Root2^.Usage AND (AssBit OR InBit)) <> 0)
        Then Write(Out,'<=') Else Write(Out,'  ');

      If (Root2^.Usage AND (RefBit Or OutBit) <> 0) AND
         ((Root1^.Usage AND (AssBit OR InBit)) <> 0)
        Then Write(Out,'=>') Else Write(Out,'  ');

      Writeln(Out,'            ',Root2^.Name);

      {ADVANCE NEXT ROOT1}
      Last := Root1;
      Root1 := Root1^.Next;
      Dispose(Last);

      {ADVANCE NEXT ROOT2}
      Last := Root2;
      Root2 := Root2^.Next;
      Dispose(Last);
    end;
  End;
  Close(Out);
  FreeMem(Buf,20000);
End.
```

(iii) MAKECONF Source

"MAKECONF.PAS"

```
Program Make_Config;

Uses CRT;

Type LinkDataType = Record
       Link    : Array[0..3,1..2] Of LongInt;
       TType,
       MemSize : LongInt;
     End;

     PtrSearchType = ^SearchType;
     SearchType = Record
       Num  : Word;
       Next : PtrSearchType;
     End;

     ProcDataType = Record
       RootLink  : Byte;
       Visited   : Boolean;
       PortsUsed : Byte;
       NextPort  : Byte;
     End;

     String255 = String[255];

     TaskDefType = Record
       Master  : Boolean;
       Fyl     : String255;
       MemArea : Array[1..4] Of String[32];
       Opt     : Array[0..4] Of Boolean;
       Tipe    : Byte;
       LineNo  : Integer;
     End;

Const Memory_Area : Array[0..4] Of String
      = ('CODE','STACK','HEAP','STATIC','DATA');

Var
  WormFile                                  : File Of LinkDataType;
  Network                                   : Array[1..256] Of LinkDataType;
  ProcData                                  : Array[1..256] Of ProcDataType;
  TPs, P, Q, NM, NW, Last, Wire, PS,
  Posn, ProcNo, MP, WP, Temp                : Integer;
  FloodFile, ConfigFile                     : Text;
  Abort_Flag, Worker_On_Master, Done        : Boolean;
  Name, Line, PLine                         : String255;
  Processor                                 : Array[1..4] Of TaskDefType;
  Param                                     : Array[1..3] Of String[100];
  Ch                                        : Char;
  SearchF, SearchB, NewItem                 : PtrSearchType;

  Procedure Terminate(S : String255);
  Begin
    Writeln(S);
    Writeln('Generation of fixed configuration and include file aborted');
    Halt(1);
  End;

  Procedure ConvLowCase(Var Str : String);
  Var I : Integer;
  Begin
    For I := 1 To Length(Str) Do
     if (str[i] >= 'A') AND (str[i] <= 'Z') Then
       str[i] := CHR(Ord(str[i]) + 32);
  End;
```

```
Procedure ShowErr(Err: String255);
Var I : Integer;
Begin
  Writeln(Line);
  For I := 1 To (Pred(p) MOD 80) Do Write(' ');
  Writeln('^');
  Writeln('LINE ',Q,':  ',Err);
  Writeln;  Writeln;
  Abort_Flag := TRUE;
  If KeyPressed Then Begin
    ch := readkey;  ch := readkey;
  End;
End;

Procedure Skip_Spaces;
Begin
  While (Pline[P]=' ') AND (P<=Length(PLine)) Do Inc(P);
  If (Pline[P]='-') AND (P<=Length(PLine)) Then Begin
    {CONTINUATION LINE}
    Readln(FloodFile,Line);

    {CONVERT TO UPPER CASE}
    PLine[0] := Line[0];
    For P := 1 To Length(Line) Do PLine[P] := Upcase(Line[P]);

    P := 1;
    Inc(Q);
    Skip_Spaces;
  End;
End;

Function CopyToSpace : String255;
Var Str   : String255;
    I     : Integer;
    Quote : Boolean;
Begin
  I := 0;  Quote := (PLine[P] = '"');
  While ((P+I) <= Length(PLine)) AND (PLine[P+I] <> '=')
        AND ((PLine[P+I] <> ' ') AND ((PLine[P+I] <> '!') AND
                                      (PLine[P+I] <> '-')) OR Quote) Do
  begin
    Inc(I);
    IF (Length(PLine)>=(P+I)) AND (PLine[P+I]='"') Then Quote := NOT Quote;
    Str[I] := PLine[Pred(P+I)];
  end;
  If Quote Then ShowErr('ERROR, " expected.');
  Str[0] := Chr(I);
  CopyToSpace := Str;
End;

Function InMemArea(Str : String255) : Integer;
Var P : Integer;
Begin
  P := 1;
  While (Memory_Area[P] <> Str) AND (P <= 4) Do Inc(P);
  If P = 5 Then P := 0;
  InMemArea := P;
End;

Function GetWorkerType(Tipe : Byte) : Integer;
Var Num   : Integer;
    Found : Boolean;
Begin
  Num := 1;
  Repeat
    found := NOT(Processor[Num].Master) And
    ((Processor[Num].Tipe = Tipe) OR (Processor[Num].Tipe = 0));
    if not found then inc(num);
  Until Found OR (Num > ProcNo);
  GetWorkerType := Num;
End;
```

```
  Procedure WriteProc(Num : Integer);
  Begin
    If (Num < 10) Then Write(ConfigFile,'0');
    Write(ConfigFile,Num);
  End;

  Procedure WriteName(Ch : Char);
  Var I : Integer;
  Begin
    Writeln(ConfigFile,Ch);
    Writeln(ConfigFile,Ch,' "',Name,'"');
    Writeln(ConfigFile,Ch);
    Writeln(ConfigFile,Ch,' FIXED CONFIGURATION FILE CREATED BY "MAKECONF"  (C)1989 A.M.Schuilenburg');
    Write(ConfigFile,Ch,' ');
    For I := 1 To 72 Do Write(ConfigFile,'=');
    Writeln(ConfigFile);
    Writeln(ConfigFile,Ch);
    Writeln(ConfigFile,Ch,'"MAKECONF" CREATES THIS FILE FROM THE NETWORK DATA PROVIDED TO IT BY THE WORM');
    Writeln(ConfigFile,Ch,'WRITTEN BY A.M.Schuilenburg, AND THE STANDARD FLOOD-FILL CONFIGURATION FILE.');
    Writeln(ConfigFile,Ch);
    Writeln(ConfigFile,Ch,' THE PURPOSE IS TO PROVIDE A FAST EASY METHOD BY WHICH TO GENERATE A FARM');
    Writeln(ConfigFile,Ch,' NETWORK SIMILAR TO THE FLOOD-FILL TYPE NETWORK.  AS A METHOD HAS NOT YET');
    Writeln(ConfigFile,Ch,' BEEN DETERMINED TO OVERCOME THE BUGS FOUND IN THE FLOOD-FILL CONFIGURER');
    Writeln(ConfigFile,Ch,' AND CHANNEL LIBRARIES, THIS FIXED CONFIGURATION METHOD HAS BEEN WRITTEN IN');
    Writeln(ConfigFile,Ch,' ORDER TO ALLOW ALL THE TRANSPUTERS AVAILABLE TO BE USED AS WORKERS AND TO');
    Writeln(ConfigFile,Ch,' ACHIEVE A FAST AND SIMPLE SOLUTION UNTIL THE BUGS MAY BE OVERCOME.');
    Writeln(ConfigFile,Ch);
    Writeln(ConfigFile);
  End;

  Function NumActive(ProcNo : Integer) : Integer;
  Var I, Cntr : Integer;
  Begin
    Cntr := 0;
    For I := 0 To 3 Do If Abs(Network[ProcNo].Link[I,1]) < 32768 Then
      INC(Cntr);
    NumActive := Cntr;
  End;

  Procedure WriteData(Num : Integer);
  Var I    : Integer;
  Begin
    With Processor[Num] Do For I := 1 To 4 Do If MemArea[I] <> '' Then Begin
       write(configfile,' ',memory_area[i],'=',memarea[i]);
     End;
    With Processor[Num] Do For i := 0 To 4 Do If Opt[i] Then
       write(configfile,' opt=',memory_area[i]);
  End;

Begin
  Assign(Output,'');  Rewrite(Output);

  Writeln('Fixed configuration generation program for 3L flood fill configuration file');
  Writeln;

  {SEARCH FOR PARAMETERS}
  For P := 1 To 3  Do Param[P] := '';
  PS := 0;       Worker_On_Master := TRUE;
  For P := 1 To ParamCount do Begin
    line := paramstr(p);
    for q := 1 to Length(Name) Do Line[q] := UpCase(Line[q]);
    if line = '/NWOM' then worker_on_master := FALSE else if ps < 3 then begin
      Inc(PS);
      Param[PS] := ParamStr(P);
    end;
  End;
  If (PS < 3) Then begin
    writeln('First  parameter must be worm configuration data file');
    writeln('Second parameter must be flood fill configuration file');
    writeln('Third  parameter must be output configuration file');
  end else begin
```

```
  Abort_Flag := FALSE;

  Assign(WormFile,Param[1]);
  {$I-}  Reset(WormFile);  {$I+}
  If (IOResult <> 0) Then begin
    Writeln('Worm configuration data file "',param[1],'" not found.');
    Abort_Flag := TRUE;
  end;

  {CHECK FOR FLOOD FILL FILE}
  Assign(FloodFile,Param[2]);
  {$I-}  Reset(FloodFile);  {$I+}
  If (IOResult <> 0) Then begin
    Writeln('Flood fill configuration data file "',param[2],'" not found.');
    Abort_Flag := TRUE;
  end;
  Close(FloodFile);   {* LEAVE THIS OPEN RESULTS IN TURBO BUG.}

  {READ IN WORM CONFIGURATION DATA}
  TPs := 0;
  While Not(EOF(WormFile)) Do Begin
    inc(tps);
    {$I-}  read(wormfile,network[tps]);   {$I+}
    if ioresult <> 0 then begin
      writeln('Error in reading worm configuration data file');
      abort_flag := TRUE;
    end;
  End;
  Close(WormFile);

{BUILD NETWORK COMMS THROUGH BREADTH FIRST SEARCH}
  {INITIALISE NETWORK DATA}
  For P := 1 To TPs Do With ProcData[P] Do Begin
    Visited := FALSE;
    RootLink := 255;
    PortsUsed := 0;
    NextPort := 0;
  End;

  {FIND CONNECTION TO HOST}
  ProcData[1].Visited := TRUE;
  P := 0;
  While (P <= 3) AND (Network[1].Link[P,1] <> (-2147483647)) Do Inc(P);
  If (P=4) Then Terminate('Worm file "'+Param[1]+'" invalid') Else
    ProcData[1].RootLink := P;
  If Worker_On_Master Then Inc(ProcData[1].PortsUsed);
  Inc(ProcData[1].PortsUsed);   {FOR CONNECTION TO FILTER TASK}

  {SEARCH QUEUE CONSISTING OF ROOT}
  New(SearchF);   SearchB := SearchF;
  With SearchF^ Do Begin
    Num := 1;
    Next := NIL;
  End;

  {SEARCH ITEMS IN THE QUEUE}
  While (SearchF <> NIL) Do Begin   {* IF FILE LEFT OPEN BUG SOLVED BY USING WRITELN TO SCREEN}
    temp := searchf^.num;
    for p := 0 to 3 do begin
      Q := Abs(Network[Temp].Link[P,1]);
      If (Q < 256) Then With ProcData[Q] Do If Not(Visited) Then Begin
        inc(procdata[temp].portsused);
        inc(portsused);
        visited := TRUE;
        rootlink := Network[Temp].Link[P,2];
        new(newitem);
        with newitem^ do begin
          Num := Q;
          Next := Nil;
        end;
        searchb^.next := newitem;
```

```
      searchb := newitem;
    End;
  end;

  {ADVANCE DOWN QUEUE}
  newitem := searchf;
  searchf := searchf^.next;
  dispose(newitem);
End;

{READ IN FLOOD FILL CONFIGURATION DATA}
For Q := 1 To 4 Do With Processor[Q] Do Begin
  fyl := '';
  for p := 1 to 4 do begin
    MemArea[P] := '';
    Opt[P] := FALSE;
  end;
  opt[0]  := FALSE;
  tipe := 0;
End;
Q := 0;   NM := 0;  NW := 0;   ProcNo := 0;
Reset(FloodFile);   {* RE-OPEN AS LEAVE THIS OPEN RESULTS IN TURBO BUG.}
While Not(EOF(FloodFile) OR Abort_Flag) Do begin
  readln(floodfile,Line);
  inc(q);

 {PARSE LINE}
  {CONVERT TO UPPER CASE}
  PLine[0] := Line[0];
  For P := 1 To Length(Line) Do PLine[P] := Upcase(Line[P]);

  P := 1;
  Skip_Spaces;
  If NOT ((P > Length(PLine)) OR (Pline[P] = '!')) {NOT BLANK LINE} Then
  If (Copy(PLine,P,4)='TASK') Then begin
    {Move over task}
    p := p + 4;
    if (pline[p]<>' ') Then ShowErr('Syntax Error');
    skip_spaces;
    name := copytospace;
    mp := pos('MASTER',name);     wp := pos('WORKER',name);
    if (((mp = 1) OR (wp = 1)) AND (Length(Name) = 6)) OR
       ((name[1]='T') AND ((mp = 3) OR (wp = 3)) AND
        (Length(Name)=8) AND ((name[2]='4') OR (name[2]='8'))) Then begin
      Inc(ProcNo);
      With Processor[ProcNo] Do Begin
        LineNo := Q;
        Master := (MP <> 0);
        If Master Then INC(NM) Else INC(NW);
        If (NM>1) Then ShowErr('Too many master definitions, only one allowed') Else
        If (NW>2) Then ShowErr('Too many worker definitions') Else Begin
          If Name[1] = 'T' Then Tipe := ORD(Name[2]) - 48;
          If ((NM = 2) OR (NW = 2)) AND (Tipe = 0) Then
           begin  {SEE IF OTHER IS TYPED, AND TYPE THIS}
             for temp := 1 to pred(procno) do
              if (master=processor[temp].master) then
              if (processor[temp].tipe = 0)
               Then ShowErr('2nd definition of WORKER processor must be typed (T4/T8)')
               Else If (Processor[temp].tipe = 4) Then Tipe := 8 Else Tipe := 4;
           end Else
           begin  {ENSURE TYPE IS UNIQUE, TYPE ANY UNTYPED.}
             For Temp := 1 To Pred(ProcNo) Do
              if (master=processor[temp].master) then
              if (tipe=processor[temp].tipe) then
               ShowErr('Duplicate Process definition') else
              if (processor[temp].tipe=0) then
               If (Tipe=4) Then Processor[temp].tipe := 8
                           Else Processor[temp].tipe := 4;
           end;
        End;
      End;
```

```
      P := P + Length(Name);   Skip_Spaces;
      Done := NOT((P < Length(PLine)) AND (PLine[P] <> '!'));
      With Processor[ProcNo] do While Not(Done OR Abort_Flag) Do Begin
        Name := CopyToSpace;
        If Name = '' Then Done := TRUE Else
        If (Name = 'INS') OR (Name = 'OUTS') Then Begin
          showerr('WARNING, Port definition ignored in flood fill file');
          p := p + length(name);
          skip_spaces; inc(p);  skip_spaces;
          name := copytospace;
          p := p + length(name);
          skip_spaces;
          abort_flag := FALSE;
        End Else If (Name = 'URGENT') Then Begin
          showerr('WARNING, Task priority not allowed URGENT ignored');
          p := p + length(name);
          skip_spaces;
          abort_flag := FALSE;
        End Else If (Name = 'FILE') Then Begin
          p := p + length(name);
          skip_spaces;
          if (pline[p] = '=') then begin
            inc(p);  skip_spaces;
            fyl := copytospace;
            p := succ(p) + length(fyl);
            skip_spaces;
          end else ShowErr('"=" expected');
        End Else If (Name = 'OPT') Then Begin
          p := p + length(name);
          skip_spaces;
          if (pline[p] = '=') then begin
            inc(p);  skip_spaces;
            name := copytospace;
            if (name = 'CODE') then opt[0] := TRUE
            else begin
              posn := inmemarea(name);
              if (posn = 0) then
                ShowErr('Expected one of:    CODE | STACK | HEAP | STATIC | DATA')
              else Opt[Posn] := TRUE;
            end;
            p := p + length(name);
            skip_spaces;
          end else ShowErr('"=" expected');
        End Else Begin
          Posn := InMemArea(Name);
          If (Posn = 0) Then ShowErr('ERROR, "'+Name+'" not a valid statement type keyword')
          Else Begin
            p := p + length(name);   skip_spaces;
            if pline[p] <> '=' then ShowErr('"=" expected') else begin
              Inc(P);  Skip_Spaces;
              Name := CopyToSpace;
              If MemArea[Posn] <> '' Then Begin
                showerr('WARNING: '+Memory_Area[Posn]+
                        ' size already defined, this definition ignored.');
                abort_flag := FALSE;
              End Else MemArea[Posn] := Name;
              P := P + Length(Name);
              Skip_Spaces;
            end;
          End;
        End;
      End;
    end else begin
      showerr('WARNING, task '+Name+' ignored.');
      abort_flag := FALSE;
    end;
  end else ShowErr('TASK definition only expected in Flood-Fill configuration file');
end;
Close(FloodFile);
If Abort_Flag Then Terminate('Error in input file/s');
```

```
{ASSUMPTION: ANY PROCESSORS UNTYPED NOW MAY BE PLACED ANYWHERE (T4 OT T8)}

{ENSURE WORKERS AND MASTER DEFINED}
If (NW=0) Then Begin
  writeln('ERROR: Worker task definition expected.');
  Abort_Flag := TRUE;
End;
If (NM=0) Then Begin
  writeln('ERROR: Master task definition expected.');
  Abort_Flag := TRUE;
End;

{ENSURE DATA SPECIFIED FOR WORKER IF ONE TO BE PLACED ON MASTER CHIP}
If Worker_On_Master Then Begin
  MP := 1;
  While (MP < ProcNo) AND NOT(Processor[MP].Master) Do INC(MP);
  If (Processor[MP].memarea[4] = '') Then begin
    p := getworkertype(processor[mp].tipe);
    if (P > procno) then begin
      Writeln('ERROR: Worker process of same compiled type as master not found as this worker');
      Writeln('       is to be placed on the same processor as the master task.');
      If Processor[Mp].Tipe=0 Then Begin
        Writeln('       Either specifically define the master type, or don''t define the type');
        Writeln('       of worker in the case of a single worker.');
      End;
      WriteLn;
      Abort_Flag := TRUE;
    end else if (processor[p].memarea[4] = '') Then begin
      Writeln('ERROR: Either MASTER or WORKER requires a data specification as both tasks');
      Writeln('       are placed on the same processor.');
      Writeln;
      Abort_Flag := TRUE;
    end;
  end;
End;

If Abort_Flag Then Terminate('Error in FARM definition');

{INSERT THE FILE NAME WHERE NECESSARY}
For P := 1 To ProcNo Do With Processor[Q] Do If Fyl = '' Then
  if master then fyl := 'master' else fyl := 'worker';

{CONVERT CONSTANT STRINGS TO LOWER CASE}
For P := 0 to 4 do convlowcase(memory_area[p]);

{CREATE FIXED CONFIGURATION FILE}
Name := Param[3];
For P := 1 to Length(Name) Do Name[P] := UpCase(name[P]);
Assign(ConfigFile,Name);
Rewrite(ConfigFile);

{OUTPUT CONFIGURED HEADER}
WriteName('!');

{OUTPUT PROCESSORS}
Writeln(ConfigFile,'! HARDWARE DECLARATIONS');
Writeln(ConfigFile,'! ---------------------');
Writeln(ConfigFile,'processor host');
Writeln(ConfigFile,'processor root');
For P := 2 To Tps Do Begin
  Write(ConfigFile,'processor procw');
  WriteProc(Pred(P));
  Writeln(ConfigFile);
End;
Writeln(ConfigFile);
Writeln(ConfigFile);

{OUTPUT WIRING}
Writeln(ConfigFile,'! WIRING FOR COMMUNICATIONS      N.B. Only links for communications wired');
Writeln(ConfigFile,'! -------------------------');
Wire := 1;
```

```
{Do the wiring for the root}
for p := 0 to 3 do begin
  if (network[1].link[p,1]= (-2147483647)) then {LINK TO HOST}
    writeln(configfile,'wire ? host[0]    root[',p,']        ! ANONYMOUS LINK CONNECTING ROOT TO PC')
  else if (network[1].link[p,1] > 0) then begin {LINK TO OTHER PROCESSOR}
    write(configfile,'wire ? root[',p,']    procw');
    writeproc(pred(network[1].link[p,1]));
    write(configfile,'[',network[1].link[p,2],']');
    writeln(configfile,'       ! WIRE ',wire);
    inc(wire);
  end;
end;

{Do the wiring for the rest, (connections required for path to host)}
for q := 2 to tps do
 If ProcData[Q].RootLink = 255 Then Terminate('INTERNAL ERROR #1') Else
  Begin
    P := ProcData[Q].RootLink;
    If Abs(NetWork[Q].Link[P,1]) > 1 Then  Begin
      Write(ConfigFile,'wire ? procw');
      WriteProc(Pred(Network[Q].Link[P,1]));
      Write(ConfigFile,'[',Network[Q].Link[P,2],'] procw');
      WriteProc(Pred(Q));
      Writeln(ConfigFile,'[',P,']       ! WIRE ',wire);
      Inc(Wire);
    End;
  End;

{OUTPUT THE TASK DEFINITIONS}
Writeln(ConfigFile);
Writeln(ConfigFile,'! TASK DECLARATIONS         The task declarations indicate channel I/O ports');
Writeln(ConfigFile,'! ------------------         and memory requirement for each task');
Writeln(ConfigFile,'task afserver  ins=1 outs=1');
Writeln(ConfigFile,'task filter    ins=2 outs=2 data=10k');

{FIND THE MASTER}
MP := 1;
While (MP < ProcNo) AND NOT(Processor[MP].Master) Do INC(MP);
Name := Processor[MP].Fyl;

P := ProcData[1].PortsUsed;
Write(ConfigFile,'task master    file=',name,' ins=',P,' outs=',P);
WriteData(MP);
Writeln(ConfigFile);

{CHECK FOR WORKER ON CHIP}
Temp := 1;
If Worker_On_Master Then Begin
  temp := getworkertype(network[1].ttype);
  write(configfile,'task worker00  file=',processor[temp].fyl);
  q := numactive(P);
  write(configfile,' ins=1 outs=1');
  writedata(temp);
  writeln(configfile);
End;

For P := 2 To TPs Do begin
  write(configfile,'task worker');
  writeproc(Pred(P));
  q := procdata[p].portsused;
  posn := getworkertype(network[p].ttype);
  write(configfile,'  file=',processor[posn].fyl,' ins=',q,' outs=',q);
  processor[posn].memarea[4] := '';
  writedata(posn);
  writeln(configfile);
end;

{OUTPUT THE PLACEMENT}
Writeln(ConfigFile);
Writeln(ConfigFile,'! PROCESS PLACEMENT');
Writeln(ConfigFile,'! -----------------');
```

```
    Writeln(ConfigFile,'place afserver host                    ! AFSERVER ALWAYS RUNS ON PC');
    Writeln(ConfigFile,'place filter   root                    ! FILTER ALWAYS RUNS ON ROOT');
    Writeln(ConfigFile,'place master   root                    ! MASTER TASK ON ROOT');
    If Worker_On_Master Then
      writeln(configfile,'place worker00 root                    ! WORKER 1 ON ROOT');

    For P := 2 To TPs Do begin
      write(configfile,'place worker');
      writeproc(Pred(P));
      write(configfile,' procw');
      writeproc(pred(p));
      writeln(configfile,'                  ! WORKER NUMBER ',Pred(P));
    end;

    {OUTPUT THE CONNECTIONS}
    Writeln(ConfigFile);
    Writeln(ConfigFile,'! CONNECTIONS BETWEEN PROCESSES');
    Writeln(ConfigFile,'! ---------------------------');
    Writeln(ConfigFile,'connect ?  afserver[0] filter[0]       ! AFSERVER & FILTER ALWAYS CONNECTED');
    Writeln(ConfigFile,'connect ?  filter[0]   afserver[0]     ! VIA THEIR PORT 0s.');
    Writeln(ConfigFile,'connect ?  master[1]   filter[1]       ! MASTER TASK ALWAYS CONNECTED TO');
    Writeln(ConfigFile,'connect ?  filter[1]   master[1]       ! FILTER VIA THEIR PORT 1s.');

    {THE CONNECTIONS TO THE MASTER}
    If Worker_On_Master Then Begin
      writeln(configfile,'connect ?  master[0]   worker00[0]     ! SOFTWARE CONNECTION');
      writeln(configfile,'connect ?  worker00[0] master[0]       !           "');
    End;
    Wire := 1;
    For P := 0 To 3 Do if (network[1].link[p,1] > 0) then begin
      {LINK FROM MASTER TO OTHER WORKER PROCESSORS}
      q := p;
      temp := pred(network[1].link[p,1]);
      if worker_on_master then inc(q) else if (q = 1) then q := 0;
      write(configfile,'connect ?  master[',q,']   worker');
      writeproc(temp);
      writeln(configfile,'[0]      ! WIRE ',wire);
      write(configfile,'connect ?  worker');
      writeproc(temp);
      write(configfile,'[0] master[',q,']  ');
      writeln(configfile,'     !    "');
      inc(procdata[succ(temp)].nextport);
      inc(wire);
    End;

    {THE CONNECTIONS BETWEEN THE WORKERS}
    For Q := 2 To TPs Do  {SHOW ONLY THE CONNECTION DIRECTION TO THE HOST}
     If ProcData[Q].RootLink = 255 Then Terminate('INTERNAL ERROR #1') Else
      Begin
        P := ProcData[Q].RootLink;
        Temp := Abs(Network[Q].Link[P,1]);
        If Temp > 1 Then  Begin
          Write(ConfigFile,'connect ?  worker');
          WriteProc(Pred(Q));
          Write(ConfigFile,'[',ProcData[Q].NextPort,'] worker');
          WriteProc(Pred(Temp));
          Writeln(ConfigFile,'[',ProcData[Temp].NextPort,']      ! WIRE ',Wire);
          Write(ConfigFile,'connect ?  worker');
          WriteProc(Pred(Temp));
          Write(ConfigFile,'[',ProcData[Temp].NextPort,'] worker');
          WriteProc(Pred(Q));
          Writeln(ConfigFile,'[',ProcData[Q].NextPort,']      !     "');
          Inc(ProcData[Q].NextPort);
          Inc(ProcData[Temp].NextPort);
          Inc(Wire);
        End;
      End;

    Close(ConfigFile);
  end;
End.
```