CONFIDENTIAL

TRANSPUTER SPECIFICATION

CONTENTS

TRANSPUTER INSTRUCTION SET

1 Notation

In this document the notation used is that of occam, with the assumption that the variables are infinite-bit two's complement integers.

Any particular processor is assumed to have a finite word length, each register in the processor holding the value of the corresponding variable in the following description. It is therefore natural to interpret a word as a fixed length twos-complement integer. Before and after execution of any instruction, the numerical value taken by each variable is correctly representable in the corresponding single word register.

The following constants are used in the description of the machine.

BitsInWord	The number of bits a machine word.
Range	The number of distinct values storeable in a word. (Range = 2**BitsInWord).
MaxInt	The largest (most positive) value representable in a word. (MaxInt = (Range/2) - 1).
MinInt	The smallest (most negative) value representable in a word. (MinInt = -(Range/2)).

The following three procedures are used. They do not affect the value held in a processor register; only the value of the corresponding variable. Consequently, they are used in the following description to change the interpretation of the register value, rather than the value itself.

```
PROC UnSign(Reg) =
  IF
    Reg < Ø
      Reg := Reg + Range
    TRUE
      SKIP :


PROC Sign(Reg) =
  IF
    Reg > MaxInt
      Reg := Reg - Range
    TRUE
      SKIP :
```

The following procedure is used to produce the value of (T1 AFTER T2) appropriate to the wordlength of the processor.

```
PROC Later(T1, T2, LaterFlag) =
  VAR TimeDiff :
  SEQ
    TimeDiff  := T1 - T2
    LaterFlag :=
      (    ((TimeDiff > Ø) AND (TimeDiff <= MaxInt    ))
       OR ((TimeDiff < Ø) AND (TimeDiff < (MinInt-1) )) )
```

2 Summary of Registers, Flags and Special Locations

Timer:

ClockReg	the current value of the processor clock
TPtrLocØ	either indicates that the level Ø timer is not in use or points to the first process on the level Ø timer queue
TNextReg[Ø]	indicates the time of the first event on the level Ø timer queue
TPtrLoc1	either indicates that the level 1 timer is not in use or points to the first process on the level 1 timer queue
TNextReg[1]	indicates the time of the first event on the level 1 timer queue

Priority Ø Queue control:

FptrReg[Ø]	pointer to front of active process list
BptrReg[Ø]	pointer to back of active process list

Priority 1 Queue control:

FptrReg[1]	pointer to front of active process list
BptrReg[1]	pointer to back of active process list

Sequential process execution:

IptrReg	pointer to next instruction to be executed
WdescReg	process descriptor of the current proces
Areg	top of evaluation stack
Breg	middle of evaluation stack
Creg	bottom of evaluation stack
Oreg	operand register
StatusReg	contains status information - see below

Initialisation, booting and analysis

MemStart	this is the most negative word in store not used by the machine for any special purpose (eg as a link-channel process word, register save word or timer pointer).

2.1 StatusReg

The only assembler programmer visible bit in the StatusReg is the ErrorFlag; this is the most significant bit. MORE EXPLAINATION.

Bit	Name	Purpose
1	GotoSNP	Cause processor to execute StartNextProcess
2	IOBit	
3	MoveBit	
4	TimeDelBit	
5	TimeInsBit	
	DistAndInsBit	
msb	ErrorFlag	

3 Workspace

In the following description, the process descriptor of the current process is also held as two variables Wptr and Priority.

```
Wptr          =   WdescReg /\ (-2)
Priority      =   WdescReg /\ 1
```

Consequently, Wptr always holds a pointer to the current process workspace, and Priority always holds the priority of the current process.

For each concurrent process, a number of locations are used to hold scheduling information. These locations are accessed using fixed word offsets from the workspace pointer, as follows:

```
Iptr.s        =   -1

Link.s        =   -2

State.s       =   -3
Pointer.s     =   -3

TLink.s       =   -4

Time.s        =   -5
```

4 Special values

The special value taken by a channel location:

```
NotProcess.p =   MinInt
```

The special values taken by the State location in the implementation of channel guards are:

```
Enabling.p    =   MinInt + 1
Waiting.p     =   MinInt + 2
Ready.p       =   MinInt + 3
```

The special values taken by the Tlink location in the implementation of timer guards are:

```
TimeSet.p       MinInt + 1
TimeNotSet.p    MinInt + 2
```

The values of true and false are:

```
MachineTRUE     1
MachineFALSE    Ø
```

5 Memory Access Procedures

In the description of the processor and instruction the following memory access procedures are used:

AtWord(Base, N, A)	sets A to point at the Nth word past Base
AtByte(Base, N, A)	sets A to point at the Nth byte past Base
RIndexWord(Base, N, X)	sets X to the value of the Nth word past Base
RIndexByte(Base, N, X)	sets X to the value of the Nth byte past Base
WIndexWord(Base, N, X)	sets the value of the Nth word past Base to X
WIndexByte(Base, N, X)	sets the value of the Nth byte past Base to X

Memory addresses start from MinInt, the process locations of the links and the event channel occupying the first few locations in memory. The number of process locations used for the links and the event pin is:

LinkChans

Other very negative addresses are used for the following special purposes

Save region	-	stores the state of an interrupted process
Timer pointer registers	-	store pointers to the first process in the timer queue

An address is a single word value divided into two parts:

a word address
a byte selector

The byte selector occupies the least significant bits in the word. The number of bits used for the byte selector is BselLength, where

```
BselLengthTab = TABLE [ Ø, Ø, 1, 2, 2, 3, 3, 3, 3 ]
BselLength    = BselLengthTab [ BitsInWord / 8 ]
```

6 Processor and Link-Channel interactions

6.1 Overview and terminology

The link-channels operate concurrently with, and are controlled by, the processor.

When a process executes an 'output message' instruction which specifies a link-channel the processor must cause the link-channel to transfer the specified message from the transputer's memory. To do this, the processor makes a 'PerformIO' request on the link-channel. This request specifies a pointer to the message, the length of the message and the priority of the process. When the message has been transferred, the link channel signals the processor with a 'RunRequest'. This will cause the processor to run the process which output the message.

When a process excutes an 'input message' instruction the interactions between the processor and an input link are similar. The processor makes a 'PerformIO' request as before and when the message has been transferred, the link channel signals the processor with a 'RunRequest' as before.

When a process refers to an input link-channel in a guard of an alternative construct the processor makes use of two further requests on the link-channel.

The first of these, called an 'Enable' request, specifies the priority of the process performing the alternative and 'enables' the link-channel. When an 'enabled' link-channel starts to receive a message it signals the processor with a 'ReadyRequest'.

The second, called a 'StatusEnquiry', does two things. Firstly, it causes the link-channel to send a message to the processor indicating if it has yet started to receive a message and, secondly, it 'disables' the link-channel if it is enabled.

<< RESETABLE LINKS TO BE ADDED >>

6.2 Occam description

Each connection between the processor and a link-channel uses 4 channels. For the i'th link channel these are

```
ProcessorToLink[i]
LinkToProcessorLØ[i]
LinkToProcessorL1[i]
LinkToProcessorStatus[i]
```

The protocol used for communication has to resolve the situation which can arise when an input link-channel signals the processor with a ReadyRequest at the same time as the processor makes a StatusEnquiry on that link-channel. This is solved by adding ReadyAck and DummyRequest messages to those mentioned above.

6.2.1 Messages on ProcessorToLink

ProcessorToLink[i] carries requests and their parameters from the processor to the link channel. The possible messages sent by the processor are

1) PerformIO <priority> <pointer> <count>

This requests the link-channel to transfer a message of <count> bytes starting at <pointer>. The priority of the link-channel for this transfer is <priority>. (Because a link-channel is one directional there is no need for the processor to specify the transfer direction).

2) Enable <priority>

This requests an input link-channel to become enabled and sets the priority of the link-channel to <priority>.

3) StatusEnquiry <priority>

This asks an input link-channel if it has started to receive a message. It also disables the link-channel if it was enabled. The link-channel responds on LinkToProcessorStatus[i], sending TRUE if it has started to receive a message, FALSE otherwise.

4) AckReady

The processor sends this to acknowledge a ReadyRequest made by the link-channel.

6.2.2 Messages on LinkToProcessorLØ and LinktoProcessorL1

The i'th link-channel uses LinkToProcessorLØ[i] to signal the processor when it is at priority Ø (high priority) and LinkToProcessorL1[i] when it is at priority 1 (low priority). The messages that sent to the processor on these channels are

1) RunRequest

This signals that a link-channel has completed passing a message.

2) ReadyRequest

This signals that an enabled link-channel has started to receive a message.

3) DummyRequest

If a link-channel has not signalled the processor with a ReadyRequest it acknowledges receipt of a StatusEnquiry with a DummyRequest.

6.2.3 Messages on LinkToProcessorStatus

A link-channel uses this channel to respond to a StatusEnquiry. It will send TRUE if it has started to receive a message; otherwise it will send FALSE.

6.2.4 Summary of message interactions

To clarify the processor and link-channel interactions, a trace of the behaviour of a link-channel is given below for all possible interactions. The traces given involve a level 1 process interacting with the i'th link-channel; the interactions involving a level Ø process are similar but have Ø substituted for 1 when the processor sends a priority and they have LinkToProcessorLØ substitutued for LinkToProcessorL1.

When the processor executes either an 'input message' or 'output message' instruction the interaction is:

```
SEQ
  ProcessorToLink[i] ? Interaction; Priority
  --                   PerformIO;   1
  ProcessorToLink[i] ? Pointer; Count
  LinkToProcessorL1[i] ! RunRequest
```

When a process perform an alternative input there are four possible interactions to consider:

1) The processor making a StatusEnquiry on the i'th link-channel.

```
SEQ
  ProcessorToLink[i] ? Interaction;  Priority -- StatusEnquiry; 1
  LinkToProcessorL1[i] ! DummyRequest
  ProcessorToLink[i] ? ANY -- AckReady
  LinkToProcessorStatus[i] ! Ready -- TRUE or FALSE
```

2) The processor Enabling the i'th link-channel which is not ready and which does not become ready before the processor makes a StatusEnquiry

```
SEQ
  ProcessorToLink[i] ? Interaction; Priority -- Enable; 1
  ProcessorToLink[i] ? StatusEnquiry; 1
  LinkToProcessorL1[i] ! DummyRequest
  ProcessorToLink[i] ? ANY -- AckReady
  LinkToProcessorStatus[i] ! FALSE
```

3) The processor Enabling the i'th link-channel which is either ready or becomes ready before the processor makes a StatusEnquiry.

```
SEQ
  ProcessorToLink[i] ? Interaction; Priority -- Enable; 1
  LinkToProcessorL1[i] ! ReadyRequest
  ProcessorToLink[i] ? ANY -- AckReady
```

4) As (3) but where the processor makes a StatusEnquiry at the same time as the link sends a ReadyRequest

```
SEQ
  ProcessorToLink[i] ? Interaction; Priority -- Enable; 1
  PAR
    LinkToProcessorL1[i] ! ReadyRequest
    ProcessorToLink[i] ? Interaction; Priority -- StatusEnquiry; 1
  PAR
    ProcessorToLink[i] ? ANY -- AckReady
    LinkToProcessorStatus[i] ! TRUE
```

6.2.5 Link-channel behaviour

```
PROC SignalProcessor(VALUE i, Signal) =
  IF
    Priority = Ø
      LinkToProcessorLØ[i] ! Signal
    Priority = 1
      LinkToProcessorL1[i] ! Signal :


PROC OutputLinkChannel(VALUE i) =
  WHILE TRUE
    VAR Request :
    SEQ
      ProcessorToLink[i] ? Request; Priority; Pointer; Count -- Perf
      SEQ Offset = [Ø FOR Count]
        VAR Byte :
        SEQ
          RIndexByte(Pointer, Offset, Byte)
          ... output byte and receive acknowledge
      SignalProcessor(i, RunRequest) :


PROC InputLinkChannel(VALUE i) =
  VAR Byte,  Pointer, Count, Priority, ChannelActive :
  VAR Ready, Enabled, Requested :

  PROC InputByteAction =
    SEQ
      ... acknowledge Byte and write it to memory
      Count := Count - 1
      IF
        Count = Ø
          SEQ
            SignalProcessor(i, RunRequest)
            Requested := FALSE
        TRUE
          SKIP :

  SEQ
    Ready     := FALSE
    Enabled   := FALSE
    Requested := FALSE
    WHILE TRUE

      VAR Interaction :
      ALT
        ... byte arrival from outside world into Byte
          IF
            Requested
              InputByteAction
            TRUE
              Ready := TRUE

        (Enabled AND Ready) & SKIP
```

```
    -- Send ReadyRequest and accept AckReady or StatusRequest
    PAR
      SignalProcessor(i, ReadyRequest)

      --  accept AckReady or StatusRequest
      SEQ
        ProcessorToLink[i] ? Interaction
        IF
          Interaction = StatusEnquiry
            SEQ
              ProcessorToLink[i] ? Priority
              LinkToProcessorStatus[i] ! TRUE
              ProcessorToLink[i] ? ANY -- AckReady
          Interaction = AckReady
            SKIP
        Enabled := FALSE

  ProcessorToLink[i] ? Interaction
    SEQ
      ProcessorToLink[i] ? Priority
      IF
        Interaction = Enable
          Enabled := TRUE
        Interaction = StatusEnquiry
          SEQ
            Enabled := FALSE
            SignalProcessor(i, DummyRequest)
            ProcessorToLink[i] ? ANY
            LinkToProcessorStatus[i] ! Ready
        Interaction = PerformIO
          SEQ
            ProcessorToLink[i] ? Pointer; Count
            Requested := TRUE
            IF
              Ready
                SEQ
                  Ready := FALSE
                  InputByteAction
              TRUE
                SKIP :
```

The behaviour of the processor is described later, its interactions with the link-channels occur only in the instruction execution loop and the execution of the 'input message', 'output message', 'enable channel' and 'disable channel' instructions.

7 Initialisation

The following registers and special locations are not set when the machine is powered on reset.

```
ClockReg
TPtrLocØ[Ø]
TPtrLoc1[1]
FptrReg[Ø]
BptrReg[Ø]
FptrReg[1]
BptrReg[1]
msb of the StatusReg (ie the errorflag)
```

The ClockReg does not increment after a power-on, reset or analyse until a store timer instruction has been executed. The states of the other registers are set as below:

```
TNextReg[Ø] = ClockReg >< (-1)
TNextReg[1] = ClockReg >< (-1)
Areg        = IptrReg
Breg        = WdescReg
Oreg        = Ø
```

If the machine is booting from external memory then

```
WdescReg    = MemStart \/ 1
IptrReg     = MaxInt - 1
Creg        = ANY
```

If the machine is booting from a link channel then

<< MEMORY READ/WRITE TO BE DESCRIBED HERE >>

```
WdescReg    = first word after boot program
IptrReg     = MemStart
Creg        = pointer to boot channel
```

8 Instruction execution

<< TO BE AMENDED TO INDICATE INTERRUPTABILITY AND STATE OF AN INTERRUPTED PROCESS. >>

```
WHILE TRUE
  SEQ
    Get Instruction
    Decode into Function and Operand
    Oreg := Oreg \/ operand
    IF
      function = prefix
        Oreg := Oreg << 4
      function = negative prefix
        Oreg := (Oreg << 4) >< (-1)
      TRUE
        SEQ
          IF
            function = operate
              secondary(oreg)
            TRUE
              primary
          Oreg := Ø
```

8.1 Prioritised scheduling

The following instructions are interruptable:

move message
input message
output message

timer alt wait
timer input

disable timer

When the machine is idle any request from the timer or a channel can be acted upon.

When the machine is at level 1 (low priority) any level Ø request can be acted upon between instructions or during any of the interruptable instructions.

When the machine is at level 1 (low priority) any level 1 request can be acted upon between instructions.

When the machine is at level Ø (high priority) any level Ø request can be acted upon between instructions.

8.2 Action taken in response to timer and link requests

```
PROC HandleTimerRequest =
  ...


PROC HandleRunRequest(VAR ChanId) =
  VAR ProcWord :
  SEQ
    RIndexWord(ChanId, Ø, ProcWord)
    -- check for
    IF
      ProcWord = NotProcess.p
        SKIP
      ProcWord <> NotProcess.p
        Run(ProcWord) :


PROC HandleReadyRequest =
  ...
```

9 Procedures Used in the Description of the Instructions

```
PROC SetErrorFlag =
  StatusReg := StatusReg \/ MinInt :


PROC ClearErrorFlag =
  StatusReg := (StatusReg /\ NOT (MinInt) ) :


PROC ReadErrorFlag(VAR State) =
  IF
    (StatusReg /\ MinInt) = MinInt
      State := TRUE
    TRUE
      State := FALSE :


PROC OverflowCheck(VAR Register) =
  IF
    (Reg > MaxInt)
      SEQ
        SetErrorFlag
        Areg := Areg - Range
    (Reg < MinInt)
      SEQ
        SetErrorFlag
        Areg := Areg + Range
    TRUE
      SKIP :
```

```
PROC Wait =
  SEQ
    WindexWord(Wptr, State.s, Waiting.p)
    WindexWord(Wptr, Iptr.s, IptrReg)
    StartNextProcess :

PROC InsertAndWait =
  SEQ
    Areg := Areg + 1
    IF
      Areg > MaxInt
        Areg := MinInt
      TRUE
        SKIP
    Plus1(Areg, Areg)
    Insert
    Wait :
```

```
PROC UpDateWdescReg(VALUE NewWdescReg) =
  -- modify the current process descriptor
  SEQ
    WdescReg := NewWdescReg
    Wptr     := WdescReg /\ (-2)
    Priority := WdescReg /\  1 :


PROC Enqueue(VALUE ProcPtr, VAR FptrReg, BptrReg) =
  -- add a process to a scheduling list
  SEQ
    IF
      FptrReg = NotProcess.p
        FptrReg := ProcPtr
      FptrReg <> NotProcess.p
        WIndexWord(BptrReg, Link.s, ProcPtr)
    BptrReg := ProcPtr :


PROC Dequeue(VALUE Level) =
  -- take a process from a scheduling list
  SEQ
    UpDateWdescReg(FptrReg[Level] \/ Level)
    IF
      FptrReg[Level] = BptrReg[Level]
        FptrReg[Level] := NotProcess.p
      FptrReg[Level] <> BptrReg[Level]
        RIndexWord(FptrReg[Level], Link.s, FptrReg[Level]) :
```

```
PROC StartNextProcess =
  -- This activates the next process to be run (if one exists).
  IF
    Priority = Ø
      IF
        FptrReg[Ø] <> NotProcess.p
          SEQ
            Dequeue(Ø)
            Oreg := Ø
            RIndexWord(Wptr, Iptr.s, IptrReg)
        FptrReg[Ø] = NotProcess.p
          SEQ
            RestoreRegisters
            IF
              (Wptr = NotProcess.p) AND (FptrReg[1] <> NotProcess.p)
                SEQ -- there was no interrupted process
                  Dequeue(1)
                  Oreg := Ø
                  RIndexWord(Wptr, Iptr.s, IptrReg)
              TRUE
                SKIP
    Priority = 1
      IF
        FptrReg[1] <> NotProcess.p
          SEQ
            Dequeue(1)
            Oreg := Ø
            RIndexWord(Wptr, Iptr.s, IptrReg)
        FptrReg[1] = NotProcess.p
          UpDateWdescReg(NotProcess.p \/ 1) :
```

```
PROC Run(VALUE ProcDesc) =
  -- schedule a process
  VAR ProcPriority, ProcPtr :
  SEQ
    ProcPriority := ProcDesc /\ 1
    ProcPtr := ProcDesc /\ (-2)
    IF
      Priority = Ø -- machine at high priority; queue process
        Enqueue(ProcPtr, FptrReg[ProcPriority], BptrReg[ProcPriority
      Priority = 1 -- machine at low priority
        IF
          ProcPriority = Ø  -- high priority process; execute it
            SEQ
              SaveRegisters
              UpDateWdescReg(ProcDesc)
              Oreg := Ø
              RIndexWord(Wptr, Iptr.s, IptrReg) :
          ProcPriority = 1  -- low priority process; queue it
            IF
              Wptr = NotProcess.p
                SEQ
                  UpDateWdescReg(ProcDesc)
                  Oreg := Ø
                  RIndexWord(Wptr, Iptr.s, IptrReg) :
              Wptr <> NotProcess.p
                Enqueue(ProcPtr, FptrReg[1], BptrReg[1]) :
```

```
PROC Insert =
  -- Insert the current process into the timer
  -- queue. The time is in Areg.
  -- Use Breg as Previous, Creg as Subsequent, Oreg as SubsequentTim
  --
  -- Previous points to the location to be updated if the current
  -- process is to be inserted in front of the process pointed to by
  -- Subsequent.
  VAR Previous, Subsequent, SubsequentTime, LaterFlag :
  SEQ
    WIndexWord(Wptr, Time.s, Areg)
    AtWord(TptrLocØ, Priority, Previous)
    RIndexWord(Previous, Ø, Subsequent)
    IF
      Subsequent <> NotProcess.p
        RIndexWord(Subsequent, Time.s, SubsequentTime)
      Subsequent = NotProcess.p
        SKIP
    Later(Areg, SubSequentTime, LaterFlag)
    WHILE (Subsequent <> NotProcess.p) AND LaterFlag
      SEQ
        AtWord(Subsequent, Tlink.s, Previous)
        RIndexWord(Previous, Ø, Subsequent)
        IF
          Subsequent <> NotProcess.p
            RIndexWord(Subsequent, Time.s, SubsequentTime)
          Subsequent = NotProcess.p
            SKIP
        Later(Areg, SubSequentTime, LaterFlag)
    WIndexWord(Previous, Ø, Wptr)
    WIndexWord(Wptr,   Tlink.s, Subsequent)
    -- Get the earliest time
    RIndexWord(TptrLocØ, Priority, Previous)
    RIndexWord(Previous, Time.s, TNextReg[Priority]) :
```

```
PROC Delete =
  -- Delete the current process from the timer queue
  -- Use Breg as Previous, Creg as Subsequent.
  VAR Previous, Subsequent :
  SEQ
    AtWord(TptrLocØ, Priority, Previous)
    RIndexWord(Previous, Ø, Subsequent)
    WHILE Subsequent <> Wptr
      SEQ
        AtWord(Subsequent, Tlink.s, Previous)
        RIndexWord(Previous, Ø, Subsequent)
    RIndexWord(Wptr, Tlink.s, Subsequent)
    WIndexWord(Previous, Ø, Subsequent)
    -- Get the earliest time
    RIndexWord(TptrLocØ, Priority, Previous)
    IF
      Previous = NotProcess.p
        SKIP
      Previous <> NotProcess.p
        RIndexWord(Previous, Time.s, TNextReg[Priority]) :

PROC TimeSlice =
  -- deschedule and reschedule the current process
  IF
    (Priority = 1) AND ???
      WindexWord(Wptr, Iptr.s, Iptr)
      Run(WdescReg)
      startnextprocess
    TRUE
      SKIP :

PROC IsThisSelectedProcess =
  -- this is used by all the disable instructions
  VAR DisableStatus :
  SEQ
    RIndexWord(Wptr, Ø, DisableStatus)
    IF
      DisableStatus = (-1)
        SEQ
          WIndexWord(Wptr, Ø, Areg)
          Areg := MachineTRUE
      DisableStatus <> (-1)
        Areg := MachineFALSE :
```

```
PROC BlockMove(VALUE Source, Destination, Count) =
  SEQ Index = [Ø FOR Count]
    VAR EightBits :
    SEQ
      RIndexByte(Source, Index, EightBits)
      WIndexByte(Destination, Index, EightBits) :


PROC Input
  VAR ChanNum :                                     -- Areg is count
  ChanOffset (Breg, ChanNum)                        -- Breg is channel
  IF                                                -- Creg is pointer
    ChanNum > LinkChans                             -- Internal channel
      VAR ProcDesc :
      SEQ
        RindexWord(Breg, Ø, ProcDesc)
        IF
          ProcDesc = NotProcess.p                   -- Not ready; wait
            SEQ
              WindexWord(Breg, Ø, WdescReg)
              WindexWord(Wptr, Iptr.s,     IptrReg)
              WindexWord(Wptr, Pointer.s, Creg)
              StartNextProcess
          ProcDesc <> NotProcess.p                  -- Ready; transfer
            VAR SourcePtr, ProcPtr :
            SEQ
              WindexWord(Breg, Ø, NotProcess.p)
              ProcPtr := ProcDesc /\ (-2)
              RindexWord(ProcPtr, Pointer.s, SourcePtr)
              BlockMove(SourcePtr, Creg, Areg)
              Run(ProcDesc)
    ChanNum <= LinkChans                            -- Link channel
      VAR PortStatus :
      SEQ
        WindexWord(Wptr, Iptr.s, IptrReg)
        WindexWord(Breg, Ø, WdescReg)
        ProcessorToLink[ChanNum] ! PerformIO; Priority; Creg; Areg
        StartNextProcess
```

```
PROC output
  VAR ChanNum :                                    -- Areg is count
  SEQ                                              -- Breg is channel
    ChanOffset(Breg, ChanNum)                      -- Creg is pointer
    IF
      ChanNum > LinkChans                          -- Internal channel
        VAR ProcDesc :
        SEQ
          RindexWord(Areg, Ø, ProcDesc)
          IF
            ProcDesc = NotProcess.p                -- Not ready; wait
              SEQ
                WindexWord(Breg, Ø, WdescReg)
                WindexWord(Wptr, Iptr.s,    IptrReg)
                WindexWord(Wptr, Pointer.s, Creg)
                StartNextProcess
            ProcDesc <> NotProcess.p               -- Ready
              VAR DestPtr, ProcPtr :
              SEQ
                ProcPtr := ProcDesc /\ (-2)
                RindexWord(ProcPtr, Pointer.s, DestPtr)
                IF -- scheduler interlock for ALT
                  DestPtr = Enabling.p
                    SEQ
                      WindexWord(ProcPtr, Pointer.s, Ready.p)
                      WindexWord(Breg, Ø, WdescReg)
                      WindexWord(Wptr, Iptr.s,    IptrReg)
                      WindexWord(Wptr, Pointer.s, Creg)
                      StartNextProcess
                  DestPtr = Waiting.p
                    SEQ
                      WindexWord(ProcPtr, Pointer.s, Ready.p)
                      WindexWord(Breg, Ø, WdescReg)
                      WindexWord(Wptr, Iptr.s,    IptrReg)
                      WindexWord(Wptr, Pointer.s, Creg)
                      Run(ProcDesc)
                      StartNextProcess
                  DestPtr = Ready.p
                    SEQ
                      WindexWord(Breg, Ø,  WdescReg)
                      WindexWord(Wptr, Iptr.s,    IptrReg)
                      WindexWord(Wptr, Pointer.s, Creg)
                      StartNextProcess
                  TRUE                             -- Ready for input
                    SEQ                            -- transfer
                      WindexWord(Breg, Ø, NotProcess.p)
                      BlockMove(Creg, DestPtr, Areg)
                      Run(ProcDesc)
      ChanNum <= LinkChans                         -- Link channel
        SEQ
          WindexWord(Wptr, Iptr.s, IptrReg)
          WIndexWord(Breg, Ø, WdescReg)
          ProcessorToLink[ChanNum] ! PerformIO; Priority; Creg; Areg
          StartNextProcess
```

1Ø Function Set

The instructions executed by the procesor include direct functions, the prefixing functions pfix and nfix, and an indirect function opr which uses the operand register Oreg to select one of a set of operations.

The set of direct functions and operations is as follows:

1Ø.1 Direct, Prefixing and Indirect Functions

Code No.	Abbreviation	Name
??	ldl	load local
??	stl	store local
??	ldlp	load local pointer
??	ldnl	load non-local
??	stnl	store non-local
??	ldnlp	load non-local pointer
??	eqc	equals constant
??	ldc	load constant
??	adc	add constant
??	j	jump
??	cj	conditional jump
??	call	call
??	ajw	adjust workspace
??	pfix	prefix
??	nfix	negative prefix
??	opr	operate

1Ø.2 Operations

Code No.	Abbreviation	Name
short	rev	reverse
long	ret	return
long	ldpi	load pointer to instruction
long	gajw	general adjust workspace
short	gcall	general call
long	mint	mimimum integer
long	lend	loop end
long	csubØ	check subscript from Ø
long	ccntl	check count from 1
long	testerr	test error
long	stoperr	stop on error
long	seterr	set error
short	bsub	byte subscript
short	wsub	word subscript
long	bcnt	byte count
long	wcnt	word count
short	lb	load byte
long	sb	store byte
long	move	move message
long	and	and
long	or	or
long	xor	exclusive or
long	not	bitwise not
long	shl	shift left
long	shr	shift right
short	add	add
short	sub	subtract
long	mul	multiply
long	div	divide
long	rem	remainder
short	gt	greater than
short	diff	difference
short	sum	sum
short	prod	product

1Ø.3 Operations Continued

Code No.	Abbreviation	Name
short	startp	start process
short	endp	end process
long	runp	run process
long	stopp	stop process
long	ldpri	load current priority
short	in	input message
short	out	output message
short	outword	output word
short	outbyte	output byte
long	resetchan	reset channel
long	alt	alt start
long	altwt	alt wait
long	altend	alt end
long	enbs	enable skip
long	diss	disable skip
long	enbc	enable channel
long	disc	disable channel
long	ldtimer	load timer
long	tin	timer input
long	talt	timer alt start
long	taltwt	timer alt wait
long	enbt	enable timer
long	dist	disable timer
long	xword	extend to word
long	cword	check word
long	xdble	extend to double
long	csngl	check single
long	ladd	long add
long	lsub	long subtract
long	lsum	long sum
long	ldiff	long diff
long	lmul	long multiply
long	ldiv	long divide
long	lshl	long shift left
long	lshr	long shift right
long	norm	normalise

1Ø.4 Operations Continued

long	testpranal	test processor analysing
long	saveh	save high priority queue registers
long	savel	save low priority queue registers
long	sthf	store high priority front pointer
long	sthb	store high priority back pointer
long	stlf	store low priority front pointer
long	stlb	store low priority back pointer
long	sttimer	store timer

***** test instructions to be included here *****

DIRECT FUNCTIONS

load local

```
SEQ
  Creg := Breg
  Breg := Areg
  RIndexWord(Wptr, Oreg, Areg)
```

store local

```
SEQ
  WIndexWord(Wptr, Oreg, Areg)
  Areg := Breg
  Breg := Creg
```

load local pointer

```
SEQ
  Creg := Breg
  Breg := Areg
  AtWord(Wptr, Oreg, Areg)
```

load non-local

```
RIndexWord(Areg, Oreg, Areg)
```

store non-local

```
SEQ
  WIndexWord(Areg, Oreg, Breg)
  Areg := Creg
```

load non-local pointer

```
AtWord(Areg, Oreg, Areg)
```

equals constant

```
IF
  Areg = Oreg
    Areg := MachineTRUE
  Areg <> Oreg
    Areg := MachineFALSE
```

load constant

```
SEQ
  Creg := Breg
  Breg := Areg
  Areg := Oreg
```

add constant

```
SEQ
  Areg := Areg + Oreg
  OverflowCheck(Areg)
```

jump

```
AtByte(IptrReg, Oreg, IptrReg)
timeslice
```

conditional jump

```
IF
  Areg = Ø
    AtByte(IptrReg, Oreg, IptrReg)
  Areg <> Ø
    SEQ
      Areg := Breg
      Breg := Creg
```

call

```
SEQ
  WIndexWord(Wptr, -1, Creg)
  WIndexWord(Wptr, -2, Breg)
  WIndexWord(Wptr, -3, Areg)
  WIndexWord(Wptr, -4, IptrReg)
  Areg := IptrReg
  VAR Temp :
  SEQ
    AtWord(Wptr, -4, Temp)
    UpDateWdescReg(Temp \/ Priority)
  AtByte(IptrReg, Oreg, IptrReg)
```

adjust workspace

```
VAR Temp :
SEQ
  AtWord(Wptr, Oreg, Temp)
  UpDateWdescReg(Temp \/ Priority)
```

1Ø.5 Register Manipulation Etc

reverse

```
SEQ
  Oreg := Areg
  Areg := Breg
  Breg := Oreg
```

return

```
SEQ
  RIndexWord(Wptr, Ø, IptrReg)
  VAR Temp :
  SEQ
    AtWord(Wptr, 4, Temp)
    UpDateWdescReg(Temp \/ Priority)
```

load pointer to instruction

```
AtByte(IptrReg, Areg, Areg)
```

general adjust workspace

```
VAR temp:
SEQ
  temp := Wptr
  UpDateWdescReg(Areg \/ Priority)
  Areg := temp
```

general call

```
VAR temp:
SEQ
  temp := IptrReg
  IptrReg := Areg
  Areg := temp
```

minimum integer

```
SEQ
  Creg := Breg
  Breg := Areg
  Areg := MinInt
```

loop end

```
SEQ
  RIndexWord(Breg, 1, Creg)
  Creg := Creg - 1
  WIndexWord(Breg, 1, Creg)
  IF
    Creg > Ø
      SEQ
        RIndexWord(Breg, Ø, Creg)
        Creg := Creg + 1
        WIndexWord(Breg, Ø, Creg)
        AtByte(IptrReg, -Areg, IptrReg)
    Creg <= Ø
      SKIP
  TimeSlice
```

1Ø.6 Checking

check subscript from Ø

```
SEQ
  UnSign(Areg)
  UnSign(Breg)
  IF
    Breg >= Areg -- unsigned compare
      SetErrorFlag
    TRUE
      SKIP
  Sign(Breg)
  Areg := Breg
  Breg := Creg
```

check count from 1

```
SEQ
  UnSign(Areg)
  UnSign(Breg)
  IF
    (Breg = Ø) OR (Breg > Areg) -- unsigned comparison
      SetErrorFlag
    TRUE
      SKIP
  Sign(Breg)
  Areg := Breg
  Breg := Creg
```

test error false and clear

```
VAR ErrorSet :
SEQ
  Creg := Breg
  Breg := Areg
  ReadErrorFlag(ErrorSet)
  IF
    ErrorSet
      Areg := MachineFALSE
    NOT ErrorSet
      Areg := MachineTRUE
  ClearErrorFlag
```

stop on error

```
VAR ErrorSet :
SEQ
  ReadErrorFlag(ErrorSet)
  IF
    ErrorSet
      SEQ
        WIndexWord(Wptr, Iptr.s, IptrReg)
        StartNextProcess
    NOT ErrorSet
```

```
        SKIP
set error
    SetErrorFlag
```

10.7 Addressing

byte subscript

```
SEQ
  AtByte(Areg, Breg, Areg)
  Breg := Creg
```

word subscript

```
SEQ
  AtWord(Areg, Breg, Areg)
  Breg := Creg
```

byte count

```
Areg := Areg * (BitsInWord / 8)
```

word count

```
SEQ
  Creg := Breg
  Breg := Areg /\ ((1 << BselLength) - 1)
  Areg := Areg >> BselLength
```

1Ø.8 Data Access and Move

load byte

```
RIndexByte(Areg, Ø, Areg)
```

store byte

```
SEQ
  WIndexByte(Areg, Ø, Breg)
  Areg := Creg
```

move message

```
BlockMove(Creg, Breg, Areg)
```

1Ø.9 Logic and Bits

and

```
SEQ
  Areg := Areg /\ Breg
  Breg := Creg
```

or

```
SEQ
  Areg := Breg \/ Areg
  Breg := Creg
```

xor

```
SEQ
  Areg := Breg >< Areg
  Breg := Creg
```

not

```
Areg := Areg >< (-1)
```

shift left

```
SEQ
  Unsign(Areg)
  IF
    Areg > BitsInWord
      SKIP
    TRUE
      SEQ
        Unsign(Breg)
        Areg := (Breg << Areg) \ Range
        Sign(Areg)
  Breg := Creg
```

shift right

```
SEQ
  UnSign(Breg)
  IF
    Areg > BitsInWord
      SKIP
    TRUE
      Areg := Breg >> Areg
  Sign(Areg)
  Breg := Creg
```

1Ø.1Ø Basic Arithmetic

add

```
SEQ
  Areg := (Breg + Areg)
  OverflowCheck(Areg)
  Breg := Creg
```

subtract

```
SEQ
  Areg := (Breg - Areg)
  OverflowCheck(Areg)
  Breg := Creg
```

multiply

```
SEQ
  UnSign(Areg)
  UnSign(Breg)
  Areg := Breg * Areg
  Breg := Areg / Range
  Areg := Areg \ Range
  Sign(Areg)
  Sign(Breg)
  IF
    ((Areg < Ø) AND (Breg <> -1)) OR
    ((Areg >=Ø ) AND (Breg <> Ø))
      SetErrorFlag
    TRUE
      SKIP
  Breg := Creg
```

divide

```
SEQ
  IF
    ((Breg = MinInt) AND (Areg = (-1))) OR (Areg = Ø)
      SetErrorFlag
    TRUE
      Areg := Breg / Areg
  Breg := Creg
```

remainder

```
SEQ
  IF
    ((Breg = MinInt) AND (Areg = (-1))) OR (Areg = Ø)
      SetErrorFlag
    TRUE
      Areg := Breg \ Areg
  Breg := Creg
```

1Ø.11 Comparison and modulo arithmetic

greater than

```
SEQ
  IF
    Breg > Areg
      Areg := MachineTRUE
    Breg <= Areg
      Areg := MachineFALSE
  Breg := Creg
```

difference

```
SEQ
  Areg := (Breg - Areg)
  IF
    (Areg > MaxInt)
      Areg := Areg - Range
    (Areg < MinInt)
      Areg := Areg + Range
    TRUE
      SKIP
  Breg := Creg
```

sum

```
SEQ
  Areg := Breg + Areg
  IF
    (Areg > MaxInt)
      Areg := Areg - Range
    (Areg < MinInt)
      Areg := Areg + Range
    TRUE
      SKIP
  Breg := Creg
```

product

```
SEQ                              -- quick unchecked multiply
  UnSign(Areg)                   -- short operand in Areg
  UnSign(Breg)
  Areg := Breg * Areg
  Areg := Areg \ Range
  Sign(Areg)
  Breg := Creg
```

10.12 Scheduling

start process

```
VAR Temp :
SEQ
  AtByte(IptrReg, Breg, Temp)
  WIndexWord(Areg, Iptr.s, Temp)
  Run(Areg \/ Priority)
```

end process

```
VAR Temp :
SEQ
  RIndexWord(Areg, 1, Temp)
  IF
    Temp = 1
      SEQ
        RIndexWord(Areg, Ø, IptrReg)
        UpDateWdescReg(Areg \/ Priority)
    Temp <> 1
      SEQ
        WIndexWord(Areg, 1, Temp-1)
        StartNextProcess
```

run process

```
run(Areg)
```

stop process

```
SEQ
  WIndexWord(Wptr, Iptr.s, IptrReg)
  StartNextProcess
```

load current priority

```
SEQ
  Creg := Breg
  Breg := Areg
  Areg := Priority
```

1Ø.13 Communication

input message

```
    input
```

output message

```
    output
```

output word

```
  SEQ
    WIndexWord(Wptr, Ø, Areg)
    Areg := BitsInWord / 8
    Creg := Wptr
    output
```

output byte

```
  SEQ
    WIndexWord(Wptr, Ø, Areg)
    Areg :=     1
    Creg := Wptr
    output
```

Reset Channel

```
VAR Temp :
SEQ
  -- Channel ID in Areg
  RIndexWord(Areg, Ø, Temp)
  WIndexWord(Areg, Ø, NotProcess.p)
  IF
    hard(Areg)
      VAR ChanId :
      SEQ
        ... decode channel ID into ChanId
        ProcessorToLink[ChanID] ! ForceEndOfMessage
        ... any consequent housekeeping
    soft(Areg)
      SKIP
  Areg := Temp
```

1Ø.14 Timer Input

read timer

```
SEQ
  Creg := Breg
  Breg := Areg
  Areg := ClockReg
```

timer input

```
VAR LaterFlag :
SEQ
  Later(Clockreg, Areg, LaterFlag)
  IF
    LaterFlag
      SKIP
    NOT LaterFlag
      InsertAndWait
```

1Ø.15 Alternative Input

alt start

```
WIndexWord(Wptr, State.s, Enabling.p)
```

alt wait

```
SEQ
  -- set up -1 in local Ø to signify
  -- that the no ready process has been selected
  WIndexWord(Wptr, Ø, -1)
  -- Is any channel or skip guard ready?
  RIndexWord(Wptr, State.s, Areg)
  IF
    Areg = Ready.p
      SKIP
    TRUE
      Wait
```

alt end

```
VAR Temp :
SEQ
  RIndexWord(Wptr, Ø, Temp)
  AtByte(IptrReg, Temp, IptrReg)
```

1Ø.16 Skip Guards

enable skip

```
IF
  Areg <> MachineFALSE
    WIndexWord(Wptr, State.s, Ready.p)
  TRUE
    SKIP
```

disable skip

```
SEQ
  IF
    Breg <> MachineFALSE
      IsThisSelectedProcess
    TRUE
      Areg := MachineFALSE
  Breg := Creg
```

10.17 Channel Guards

enable channel

```
SEQ
  IF
    Areg <> MachineFALSE
      VAR ChanNum:
      SEQ
        ChanOffset(Breg, ChanNum)
        IF
          ChanNum > LinkChans
            VAR Temp :
            SEQ
              RIndexWord(Breg, 0, Temp)
              IF
                Temp = NotProcess.p
                  WIndexWord(Breg, 0, WdescReg)
                Temp = WdescReg
                  SKIP
                TRUE
                  WIndexWord(Wptr, State.s, Ready.p)

          ChanNum <= LinkChans
            VAR Ready :
            SEQ
              -- is channel ready
              ProcessorToLink[ChanNum] ! StatusEnquiry; Priority
              PAR
                LinkToProcessorStatus[ChanNum] ? Ready
                ConditionalOutputInhibit(ChanNum)

              IF
                Ready
                  WIndexWord(Wptr, State.s, Ready.p)
                NOT Ready
                  SEQ
                    ProcessorToLink[ChanNum] ! Enable; Priority
                    WIndexWord(Breg, 0, WdescReg)

    TRUE
      SKIP
  Breg := Creg
```

disable channel

```
    IF
      Breg <> MachineFALSE
        VAR ChanNum:
        SEQ
          ChanOffset(Creg, ChanNum)
          IF
            ChanNum > LinkChans
              SEQ
                RindexWord(Creg, Ø, Breg)
                IF
                  Breg = NotProcess.p
                    Areg := MachineFALSE
                  Breg = WdescReg
                    SEQ
                      WIndexWord(Creg, Ø, NotProcess.p)
                      Areg := MachineFALSE
                  TRUE
                    IsThisSelectedProcess

            ChanNum <= LinkChans
              VAR Ready :
              SEQ
                WIndexWord(Creg, Ø, NotProcess.p)
                -- Ask if channel is ready and hence switch off char
                ProcessorToLink[ChanNum] ! StatusEnquiry; Priority
                PAR
                  LinkToProessorStatus[ChanNum] ? Ready
                  ConditionalOutputInhibit(ChanNum)
                IF
                  Ready
                    IsThisSelectedProcess
                  NOT Ready
                    Areg := MachineFALSE
      TRUE
        Areg := MachineFALSE
```

1Ø.18 Alternative Timer Input

timer alt start

```
SEQ
  WIndexWord(Wptr, TLink.s, TimeNotSet.p)
  WIndexWord(Wptr, State.s, Enabling.p)
```

timer alt wait

```
VAR LaterFlag :
SEQ
  -- -1 in local Ø signifies that
  -- no process has yet been selected
  WIndexWord(Wptr, Ø, -1)
  RIndexWord(Wptr, State.s, Creg)
  IF
    Creg = Ready.p
      WIndexWord(Wptr, Time.s, ClockReg)
    Creg <> Ready.p
      SEQ
        RIndexWord(Wptr, Tlink.s, Breg)
        IF
          Breg <> TimeSet.p
            Wait
          Breg = TimeSet.p
            SEQ
              RIndexWord(Wptr, Time.s, Areg)
              Later(ClockReg, Areg, LaterFlag)
              IF
                LaterFlag
                  SEQ
                    -- ready due to clock
                    WIndexWord(Wptr, State.s, Ready.p)
                    WIndexWord(Wptr, Time.s, ClockReg)
                TRUE
                  InsertAndWait
```

1Ø.19 Timer Guards

enable timer

```
SEQ
  IF
    Areg <> MachineFALSE
      VAR Temp :
      SEQ
        RIndexWord(Wptr, Tlink.s, Temp)
        IF
          Temp = TimeNotSet.p
            SEQ
              WIndexWord(Wptr, Tlink.s, TimeSet.p
              WIndexWord(Wptr, Time.s,  Breg)
          Temp = TimeSet.p
            VAR LaterFlag :
            SEQ
              RIndexWord(Wptr, Time.s, Temp)
              Later(Temp, Breg, LaterFlag)
              IF
                LaterFlag
                  WIndexWord(Wptr, Time.s, Breg)
                NOT LaterFlag
                  SKIP
    Areg = MachineFALSE
      SKIP
  Breg := Creg
```

disable timer

```
IF
  Breg <> MachineFALSE
    SEQ
      RIndexWord(Wptr, Tlink.s, Oreg)
      IF
        Oreg = TimeNotSet.p
          Areg := MachineFALSE
        Oreg = TimeSet.p
          VAR LaterFlag :
          SEQ
            RIndexWord(Wptr, Time.s, Oreg)
            Later(Oreg, Creg, LaterFlag)
            IF
              LaterFlag
                IsThisSelectedProcess
              NOT LaterFlag
                Areg := MachineFALSE
        TRUE
          SEQ -- remove process from timer queue
            Delete
            WIndexWord(Wptr, Tlink.s, TimeNotSet.p)
            Areg := MachineFALSE
  Breg = MachineFALSE
    Areg := MachineFALSE
```

10.20 Partword arithmetic

extend to word

```
SEQ
  Unsign(Areg)
  IF
    (Breg < Areg)
      Areg := Breg
    TRUE
      Areg := Breg - (2*Areg)
  Breg := Creg
```

check word

```
SEQ
  Unsign(Areg)
  IF
    (Breg >= Areg) OR (Breg < -Areg)
      SetErrorFlag
    TRUE
      SKIP
  Areg := Breg
  Breg := Creg
```

1Ø.21 Long arithmetic

extend to double

```
SEQ
  Creg := Breg
  IF
    Areg < Ø
      Breg := -1
    Areg >=Ø
      Breg := Ø
```

check single

```
SEQ
  IF
    ((Areg < Ø) AND (Breg <> (-1))) OR
     ((Areg >= Ø) AND (Breg <> Ø ))
      SetErrorFlag
    TRUE
      SKIP
  Breg := Creg
```

long add

```
SEQ
  Areg := (Breg + Areg) + (Creg /\ 1)
  OverflowCheck(Areg)
```

```
long subtract

    SEQ
      Areg := (Breg - Areg) - (Creg /\ 1)
      OverflowCheck(Areg)

long sum

    SEQ
      UnSign(Areg)
      UnSign(Breg)
      Areg := (Breg + Areg) + (Creg /\ 1)
      IF
        (Areg > Range)
          SEQ
            Breg := 1
            Areg := Areg - Range
        TRUE
          Breg := Ø
      Sign(Areg)

long diff

    SEQ
      UnSign(Areg)
      UnSign(Breg)
      Areg := Areg + (Creg /\ 1)
      IF
        Breg > Areg
          Areg := Breg - Areg
          Breg := Ø
        Breg <= Areg
          Areg := (Breg - Areg) + Range
          Breg := 1
      Sign(Areg)
```

long multiply

```
SEQ
  UnSign(Areg)
  UnSign(Breg)
  UnSign(Creg)
  Areg := (Breg * Areg) + Creg
  Breg := Areg / Range
  Areg := Areg \ Range
  Sign(Areg)
  Sign(Breg)
```

long divide

```
SEQ
  UnSign(Areg)
  UnSign(Breg)
  UnSign(Creg)
  IF
    Creg >= Areg
      SetErrorFlag
    Creg < Areg
      VAR Temp :
      SEQ
        Temp := Areg
        Breg := (Creg * Range) + Breg
        Areg := Breg / Temp
        Breg := Breg \ Temp
        Sign(Areg)
        Sign(Breg)
```

normalise

```
IF
  (Breg = Ø) AND (Areg = Ø)
    Creg := 2*BitsInWord
  TRUE
    SEQ
      UnSign(Areg)
      UnSign(Breg)
      Areg := (Breg * Range) + Areg
      Creg := Ø
      WHILE Areg < ((Range * Range) / 2)
        SEQ
          Areg := Areg << 1
          Creg := Creg + 1
      Breg := Areg / Range
      Areg := Areg \ Range
      Sign(Areg)
      Sign(Breg)
```

long shift left

```
IF
  (Areg < Ø) OR (Areg > (2 * BitsInWord))
    SKIP
  TRUE
    SEQ
      UnSign(Breg)
      UnSign(Creg)
      Breg := (Creg * Range) + Breg
      Breg := Breg << Areg
      Areg := Areg \ Range
      Breg := (Breg / Range) \ Range
      Sign(Breg)
      Sign(Areg)
```

long shift right

```
IF
  (Areg < Ø) OR (Areg > (2 * BitsInWord))
    SKIP
  TRUE
    SEQ
      UnSign(Breg)
      UnSign(Creg)
      Breg := (Creg * Range) + Breg
      Breg := Breg >> Areg
      Breg := Breg / Range
      Areg := Areg \ Range
      Sign(Breg)
      Sign(Areg)
```

1Ø.22 Booting and analysing

test processor analysing

```
IF
  analysing
    Areg := TRUE
  TRUE
    Areg := FALSE
```

save high priority queue registers

```
SEQ
  WindexWord(Areg, Ø, FptrReg[Ø])
  WindexWord(Areg, 1, BptrReg[Ø])
```

save low priority queue registers

```
SEQ
  WindexWord(Areg, Ø, FptrReg[1])
  WindexWord(Areg, 1, BptrReg[1])
```

store high priority front pointer

```
FptrReg[Ø] := Areg
```

store high priority back pointer

```
BptrReg[Ø] := Areg
```

store low priority front pointer

```
FptrReg[1] := Areg
```

store low priority back pointer

```
BptrReg[1] := Areg
```

store timer

```
SEQ
  TimerReg := Areg
  StartTimer
```

MEMORY CONFIGURATION

11 Configuration register Bits 36, 37, 38 and 39

These have no use and should be removed. This leaves a 36-bit configuration register and the remainder of this note assumes this.

12 Order of reading configuration information

The configuration register is loaded starting at bit Ø and finishing at bit 35. We should make sure that this is stated in the manual as this is necessary information for anyone trying to configure without an address decoder.

13 Memory interface configuration address

The configuration addresses are word addresses. The values put out on the memory interface will have bits AD2 to AD31 corresponding to the word address. Bits AD1 and ADØ should be 1 since neither a byte write nor a refresh cycle is being performed.

We want to waste as little memory space as possible so the configuration information should be held as close to the top of memory as possible. The two highest byte location of the address space are occupied by the ROM boot instructions so the first available full word is #7FFFFFF8. Therefore addresses #7FFFFF6C through #7FFFFFF8 should be used to contain the memory interface configuration information.

In keeping with the standard 'little endian' convention used elsewhere in the transputer architecture the least significant bit should correspond with the least significant address. This means that #7FFFFF6C should contain bit Ø and #7FFFFFF8 should contain bit 35.

This gives the following association of addresses with bits in the configuration register.

Word address	Bit of configuration register	Function
#7FFFFFF6C	Ø	T1 lsb
#7FFFFFF7Ø	1	T1 msb
#7FFFFFF74	2	T2 lsb
#7FFFFFF78	3	T2 msb
#7FFFFFF7C	4	T3 lsb
#7FFFFFF8Ø	5	T3 msb
#7FFFFFF84	6	T4 lsb
#7FFFFFF88	7	T4 msb
#7FFFFFF8C	8	T5 lsb
#7FFFFFF9Ø	9	T5 msb
#7FFFFFF94	1Ø	T6 lsb
#7FFFFFF98	11	T6 msb
#7FFFFFF9C	12	notS1 lsb
#7FFFFFFAØ	13	notS1
#7FFFFFFA4	14	notS1
#7FFFFFFA8	15	notS1
#7FFFFFFAC	16	notS1 msb
#7FFFFFFBØ	17	notS2 lsb
#7FFFFFFB4	18	notS2
#7FFFFFFB8	19	notS2
#7FFFFFFBC	2Ø	notS2
#7FFFFFFCØ	21	notS2 msb
#7FFFFFFC4	22	notS3 lsb
#7FFFFFFC8	23	notS3
#7FFFFFFCC	24	notS3
#7FFFFFFDØ	25	notS3
#7FFFFFFD4	26	notS3 msb
#7FFFFFFD8	27	notS4 lsb
#7FFFFFFDC	28	notS4
#7FFFFFFEØ	29	notS4
#7FFFFFFE4	3Ø	notS4
#7FFFFFFE8	31	notS4 msb
#7FFFFFFEC	32	RefreshInterval lsb
#7FFFFFFFØ	33	RefreshInterval msb
#7FFFFFFF4	34	RefreshEnable
#7FFFFFFF8	35	LateWrite

CHECK, ANALYSE AND RESET

14 Introduction

This note sets out the change in function of the Analyse and Reset pins and the replacement of the Stop pin by the Check pin. It also notes potential future improvements to this specification.

15 Check

The Check pin causes the transputer processor to be brought to an immediate clean halt when the Error flag is set by an instruction.

The Check pin is sampled on the falling edge of Reset. If the pin is high then the machine will halt when the Error flag next changes from Ø to 1. If the pin is low then the setting or unsetting of the Error bit will not affect the transputer.

The definition that the processor will halt on a Ø to 1 transition of the Error bit ensures that a transputer which has been halted as the result of the Error bit being set can be booted and analysed whilst preserving the Error bit. The act of clearing the Error bit then re-enables the check.

When the processor halts as a result of the Error bit becoming set, the Iptr will point to the byte of memory which follows the instruction which generated the error. The processor does not execute any further instructions or respond to any Run or Ready requests from the links.

A list of the instructions which can cause the Error flag to be set is appended.

16 RESET and ANALYSE

When not used in conjunction with the Analyse pin the specification of the Reset pin is unchanged.

The purpose of the Analyse pin is to enable a transputer system to be brought to a 'clean' halt so that the state of the processors in that system can be examined.

A system is analysed by analysing all transputers in the system in the following amnner. First the Analyse pin is taken high; this will cause the system to come to a 'clean' halt after a specifiable time. The Reset pin is then taken high for at least the specified Reset hold time. The Reset pin is then taken low whilst still holding the Analyse pin high; this will prevent both the re-initialisation of the external memory interface and the start of the booting sequence. The Analyse pin is then taken low which permits the transputer to boot. Note that the earliest time at which the transputer is guarenteed to be able to receive a (boot) message remains specified relative to the fall of Reset rather than the fall of Analyse.

16.1 Analyse

This describes what happens in response to Analyse being brought high.

16.1.1 Processor

The processor will respond to Analyse only at specific times during its operation. The processor responds to Analyse by halting any process which is executing and then ignoring any scheduling requests which may be made by the links or the timer.

If the processor is not executing a process the processor responds to Analyse at once and halts immediately.

If the processor is executing a process the processor responds to Analyse by halting either at the next descheduling point (ie "start next process") or at the next point at which a low priority process would be timesliced (this will be an unconditional jump or a loop end instruction). Note that this permits a high priority process to pre-empt a low priority process, in which case the processor will halt during the execution of the high priority process. The Iptr of a process which has been halted will point to the byte of memory following the final byte of the instruction which caused the process to be halted.

A list of instructions on which a process can halt is appended.

16.1.2 Timer

The clock responds to Analyse by stopping. Any processes on the timer queue will either be scheduled or will remain on the queue.

16.1.3 Links

Analyse has no effect on input links; they continue to operate normally, sending acknowledges and making scheduling requests as appropriate.

Analyse causes output links to output at most a few more data packets. They respond correctly to acknowledge packets and will make scheduling requests as appropriate. The number of data packets which a link will output after Analyse is asserted is bounded by the number of bytes in a processor word.

16.2 Reset

If the Analyse pin is held low when Reset is brought low then the external memory interface will be re-initialised and subsequent TestProcessorAnalysing instructions will generate 'false'.

If the Analyse pin is held high when Reset is brought low then the external memory interface will not be re-initialised and subsequent TestProcessorAnalysing instructions will generate 'true'. In addition the processor will not restart until the Analyse pin has been brought low.

When the processor restarts, the values which were in the processor's Wdesc and Iptr when it halted are placed in the processor's stack.

The values which the process words and counts of the links had at the time that the links halted are readable when the transputer is booted. The count associated with an input link indicates the number of bytes which it has input and acknowledged. The count associated with an output link indicates the number of acknowledgements it has received.

Provided that the process word of a link had been initialised (either on bootstrap or before use) the process word indicates whether a link was in use, and, if the process using that link was performing an input or an output then the count indicates how much of the message had been transferred. If a link which was in use contains a count of zero then the message had been completely transferred but the process had not been scheduled.

If the processor was not executing a process when it halted the Breg will contain NotProcess.p.

If the processor was executing a process when the processor halted the Breg will contain the value of the Wdesc of that process and the Areg will contain the value of the Iptr of that process (which will be as described above).

If the processor was booted from a channel the Creg contains the identity of that channel.

The processor will be at low priority.

The Wptr will contain a pointer to the first free word of memory. This will be MemStart in the case of booting from ROM, or the first word after the end of the loaded program if booting from a channel.

16.3 Information available

If the process word associated with a link contains a process descriptor then the link is being used for output, (unconditional) input or alternative input.

If the link was being used for output then the value in the link's count register indicates whether the message transfer had completed. If the count is Ø then the message transfer had completed and the process would have been scheduled if the processor had not halted.

If the link was being used for (unconditional) input then the value in the link's count register indicates whether the message had completed just as for a link which was being used for output.

If two processes are communicating and waiting on either end of a link then the message being transferred is held in the outputing transputer. If a process has input a message but has not yet resumed execution then the message is held correctly in the inputting transputer.

1) Timer lists
   NB These require initialisation by software
2) Process queues
   NB These require initialisation by software. The low priority front pointer must be saved and initialised by the first analysis program.
3) Channel process words
   NB For complete integrity these must be initialised by software to NotProcess
4) Channel counts.

16.4 Improvements to Reset and Analyse

There are two improvements to this specification for analyse which we should consider in future revisions.

1) Readiness of links.

It should be possible to determine what the readiness of the links were when a transputer was analysed. To be more precise, it should be possible to determine if a link was ready (had received an 'unsolicited byte') when analyse was deasserted. This would greatly simplify the analysis of interconnected transputer systems, and would increase the robustness of such analysis in the presence of hardware failures. It would also improve the degree of analysis possible for transputers connected to non-transputers and would aid certain hardware debugging.

2) Queueing of processes

After the processor has responded to Analyse it should not ignore Ready and Run requests from links as specified here. Rather, it should place waiting processes on the appropriate scheduling list. This change would give rise to a situation where processes would not wait on channels if their message transfer has completed, nor would processes performing alternative input continue to wait on a link channel after that link channel had become ready. This would simplify analysis and hardware debugging.

Appendix 1

Instructions which may cause the processor to halt and the consequence of the processor halting on that instruction.

1) Jump	the jump would have been taken.
2) Loop end	the instruction has updated the count locations and the consequential jump would have occured.
3) End Process	the process count will have been updated and the process would have been descheduled
4) Stop Process	the process would have been descheduled
5) Stop On Error	the process would have been descheduled
6) Input Message	the process descriptor will have been left in the channel and the process would have been descheduled.
7) Output Message Output Word Output Byte	the process descriptor will have been left in the channel and if the process has output to a channel from which another process was performing alternative input that other process will have been scheduled. The process would have been descheduled.
8) Timer Input	the process will have been inserted into the timer queue and would have been descheduled.
9) Alt Wait	the value Waiting.p will have been written into the State location and the process would have been descheduled.
10) Timer Alt Wait	the value Waiting.p will have been written into the State location, the process will have been inserted into the timer queue if appropriate and the process would have been descheduled.

Appendix 2

Instructions which may cause the Error flag to be set.

1) Add Constant
2) Check subscript from Ø
3) Check count from 1
4) Set Error
5) Add
6) Subtract
7) Multiply
8) Divide
9) Remainder
10) Check word
11) Check single
12) Long Add
13) Long Subtract
14) Long Divide