The University of Newcastle upon Tyne
Computing Laboratory

A Structured Description of Protocols for the Cambridge Ring

by

F. Panziere and S.K. Shrivastava

22 December 1981

I. Introduction

The Cambridge Ring is finding a widespread use, at least in the U.K., as a means of Connecting a number of computers to form a local area network. Despite its popularity, we have not come across any agreed set of protocol specifications for it. Rather, most users have **1** fined and implemented their own versions of protocols based more or less on the work done at Cambridge. This work is documented in an informal technical note [1] that has provided guidelines to most implementors of Ring protocols. Because of the informality of that document, implementors have rarely agreed on its interpretation and as a result, no standards exist. We wish to rectify this situation by presenting a sufficiently precise description of the protocols most commonly implemented on the Cambridge Ring. As a starting point, we present a procedural description of the Basic Block Protocol. We describe the BBP by making use of the programming language Concurrent PASCAL [2]; a similar approach has been used for the description of the Ethernet communication protocols [3] . We use Concurrent Pascal strictly as a notational tool.

In order to simplify our description and to overcome some limitations imposed by this language (e.g. no features for the handling of asynchronous exceptions such as time out's) we have introduced few additional features which are fully discussed in section 6 of this document. The descriptions of the BBP is given in the Appendices; the main body of this document contains an overall view of the various communication protocols implemented on the Cambridge Ring.

2. The Cambridge Ring

The Cambridge Ring [4] provides a reliable communication medium for the exchange of messages (each of two bytes in length) at a high speed between computers (nodes). At the Ring level, a message is transported within a 'minipacket' of a fixed length (38 bits in most of the current installations of the Cambridge Ring) which circulates around the Ring. Each minipacket has available a 2 byte data field for a message. Each node is identified over the Ring by a unique station address. In order to transmit a minipacket a station waits for the arrival of an empty minipacket, fills it with two bytes of data, source and destination addresses and forwards it onto the Ring. A copy of the minipacket is maintained by the transmitting station. The source station then waits for that minipacket to return acknowledged by the destination. The acknowledgement of the minipacket (e.g. accepted, ignored, etc.) is encoded in a few status bits of the minipacket itself which the source can examine as it returns. The specific details of the various possible results of a minipacket transmission are given in Appendix A. One last detail worth mentioning is that a station can be made selective to the reception of messages. Each station incorporates a Source Select Register (SSR) which has the following three properties: i) if the SSR is set to 0 then the SSR's home station will reject any incoming minipackets; ii) if the SSR is set to a value ranging from 1 to 254 then the SSR's home station will accept only minipackets coming from the station whose address matches the content of the SSR; and iii) if the SSR is set to 255 then the SSR's home station will accept minipackets coming from any source station. These features of the Ring are crucial to the construction of high level protocols.

3- Protocol Hierarchy

The Ring only provides facilities for the exchange of 2 bytes long messages between nodes; yet most applications will require the facilities of transmitting and receiving arbitrary length messages between user processes. As is well known, a systematic way of achieving this abstraction is to construct a hierarchy of protocols. Figure 1 skows the protocol hierarchy that has been widely used over the Cambridge Ring (its close resemblance to the protocol hierarchy as employed in the Ethernet] [5] is not accidental).

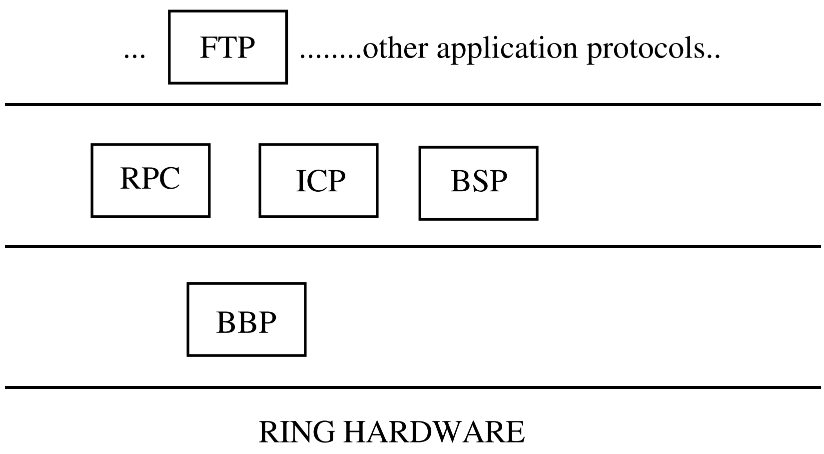

Fig. 1 Hierarchy of Ring Protocols

The first protocol (Basic Block Protocol) provides a service whereby a variable length message can be delivered to the destination. The next level protocols use BBP to implement slightly more sophisticated services. The RPC (Remote Procedure Call) protocol provides a procedure call abstraction, while the BSP (Byte Stream Protocol) provides a data stream abstraction. Before a stream can be opened, it is necessary to establish a connection between two processes, this function is performed by the ICP (Initial Connection Protocol). These protocols provide a convenient basis for the construction of specific user level protocols (e.g. File Transfer Protocol).

4. The Basic Block Protocol

An overview of the protocol is presented here; specific details are documented in Appendix B. BBP provides the facilities for sending and receiving variable length blocks of data (messages). It transports a message on a best effort basis, that is messages are delivered with a high probability of success. It also implements a simple naming scheme such that a message is delivered to a specified 'port'. Ports are integer valued names defined in a name space local to each node. A process can 'attach' itself to a port thus enabling it to receive messages directed to that port. The following four operations are available on the BBP interface:

(i) send message (...): this operation is used to transmit a message to a named destination port of a specified station. The functions of the send_message(...) primitive are: i) to construct a data object, termed a Basic Block, which contains the message to be transmitted and ii) to transmit each pair of bytes composing the Basic Block as a minipacket. The layout of a Basic Block is depicted in figure 2.

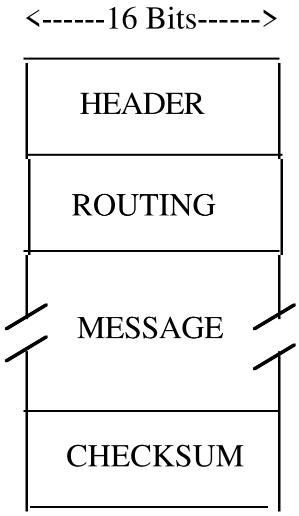

Fig. 2 Layout of a Basic Block

(ii) receive_message (...): this operation is used to receive a message on a specified port.

(iii)get_port (...): the BBP software maintains a pool of ports; this operation returns a free port.

(iv) release_port(...): a port owning process can release its port by calling this operation; this port then joins the pool of free ports.

We will use a simple example to help the reader to appreciate how the services provided by the BBP are intended to be used. Assume that a process P1 on node N1 wishes to receive a message from process P2 on Node N2; assume further that the port number on which P2 is receiving messages (PN2) is known to P1 (P1 might be a user process and P2 a server, say a file server process). A simple means of achieving this is as illustrated in figure 3. Process P1 informs P2 of the identity of the port on which P1 is willing to receive. Of course, P2 must be in a position to decode the messages directed to it appropriately. Obviously this is ensured at higher levels of the software and not at the level of the BBP.

```
Process P1                                Process P2

get_port(...);                              .

                                            .
```

```
send_message(...);    ---------------- >    receive_message(...);
"The sent message contains                 "Receive messages on PN2"
the port number returned
by getport(...)"                            "decode received message"

receive_ message(...); <----------------     send_ message(...);
"receive messages on the                    "send message to the
port returned by getport(...)"               port number contained
                                             in the received message"
release_port(...);                           .
```

Fig. 3 Client Server interaction

Finally, a few remarks on process structure might be helpful to understand the details of the protocol presented in Appendix B.

Remark 1: The transmit resources of the Ring are used exclusively during a send_message(..) operation. That is at any time, a station can send only one basic block. For this reason a 'resource' monitor is used to give the required exclusive access.

Remark 2: The pool of ports is also managed by a monitor.

Remark 3: Yet another monitor is needed that maintains a list of 'active ports - ports' - on which processes are waiting for messages.

Remark 4: A receiver process is necessary to receive the Basic Blocks as they arrive. Its main functions are i) to construct a Basic Block out of the arriving minipackets; ii) to check if the indicated port in the Basic Block is indeed 'active'; iii) and if so, deliver the message contained in the received Basic Block to the waiting process through the active ports monitor.

5. Other Protocols

WATCH THIS SPACE

6. Use of Concurrent Pascal in the Procedural Model

The procedural model of the Basic Block Protocol presented in this document is described using the programming language Concurrent Pascal. Concurrent Pascal was chosen because of its clarity and concurrency features and is used simply as a tool for the purposes of our description. Minor modifications (addition of some facilities) to the language have been made to simplify the presentation:

1) The type 'bit ' was assumed as a standard type. It allows us to define data objects which occupy a single word.

2) The lack of variable array bounds in the language Pascal had led us to treat arrays as if they are always of a fixed size. In fact, the size of a message varies from 1 to some maximum number of words; hence the array bounds of the data objects of type messages defined in our model have to be thought of as varying within the appropriate range

3) An **exit** command has been assumed to terminate a loop.

4) The following feature has been assumed for the treatment of asynchronous time-out events.

within t **do** Sl <EXPIRED ? (S2)>;

If the statement 'S1' is executed within the specified time interval 't ' then flow of control never reaches the command sequence enclosed in the angular brackets. Otherwise a 'time-out' exception occurs which has the effect of stopping the execution of 'S1' and starting the execution of statement 'S2'.

5) The standard procedure 'wait' is used to halt the executing process for a period of time 't':

wait(t: integer);

6) Any other liberties taken of the language are indicated as comments in the description. We have chosen ease of description as our main objective rather than staying rigorously within the framework of the language. We believe that this has resulted in a more readable description than would otherwise have been possible.

References

1. M. A. Johnson, "Ring byte stream protocol specification," Technical Note, Computer Laboratory, Cambridge (April 1980).

2. P. Brinch Hansen, The Architecture of Concurrent Programs, Prentice Hall (1977).

3. Digital, Intel, and Xerox, "The Ethernet - A Local Area Network - Data Link Layer and Physical Layer Specifications,.' Computer Communication Review Vol. **11**(3), ACM (July 1981).

4. SERC (DCS), Cambridge Data Ring, September 1980.

5. D.R. Boggs, T.F. Sochs, E.A. Taft, and R.M. Metcalfe, "Pup: Internetwork Architecture, IEEE Transactions on Communications vol . **COM-28**(4) (April 1980).

APPENDIX A

Ring Interface

Several different manufactures market their own version of the Ring. In order to abstract away from the peculiarities of a given product, we have chosen to describe the Ring hardware interface in the following idealised manner.

The data types defined on the Ring interface are:

```
type
    SSR_ value        = 0..255;      {range of Source Select Register's
                                      values }
    station_ address  =              l..254;    (subrange of station address
                                      values }
    word = uinv array[0..15] of bit;    {type "bit" is predefined
                                      standing for a single bit}
```

```
pkt _status = (busy, accepted, deaf, absent, transmission-error);
```

The following primitive operations are available on the Ring inter-face

```
function transmit_packet(destination : station_address;
                         packet      : word
                         ): pkt _status;

procedure receive_packet(var source: station_address;
                          var packet: word
                          );

procedure set_SSR(selected_stn: SSR_value);
```

It is relatively straightforward to map the operations of a given Ring onto the above operations.

The function 'transmit_packet (. . .)' and the procedure 'receive_packet (...)' implement the Ring hardware communication protocol. The returned values of the transmit_packet(...) primitive have the following meaning:

i) 'busy': the transmitted packet was not accepted by the destination station because that station was not listening to anyone (i.e. the Source Select Register of the destination station was set to 0);

ii) 'accepted' : the destination station has received the transmitted packet;

iii) 'deaf': the transmitted packet was rejected by the destination station because that station was currently receiving from some other source (i.e. the Source Select Register of the destination station was set to a value different from the station address of the transmitting station);

iv) 'absent': the transmitted packet was ignored by the destination station because that station was not on line;

v) 'transmission-error': the transmitted packet got corrupted somewhere during its passage through the Ring. In this case the response of the destination is not known.

It is worth mentioning here that the 'transmit_ packet(...)' function does not have a time-out response associated with it. As a consequence, the execution of this operation will not return if the packet is not acknowledged due to a fault in the Ring hardware. This suggests that the software using this operation should contain measures to avoid indefinite waiting. The same comments apply to the procedure 'receive packet(...)' which terminates only after a packet has been received. In the BBP description given below, appropriate time-out mechanisms have been used.

The operation 'set_SSR(...)' is used to set the contents of the Source Select Register to the value of the input parameter 'selected_ stn'. The effect of this operation has been described in section 2 of this document.

APPENDIX B

Basic Block Protocol interface

The following constants and data types are defined on the BBP interface:

```
const
      MAX_ MSG_ SIZE  = ...;    {implementation dependent constant,
                                 typically 1024}
      MAX_PORT        = ...,    {implementation dependent constant,
                                 typically 4096}

type

      port_number = 0..MAX_ PORT;

      msg_size    = 1. .MAX_MSG_ SIZE;

      s_option    = (checksum, no-checksum);
      message     = array[msg_size] of word;

      send_ status = (ok, absent, not-done, unable);

      destination_ address =  record
                               station: station_ address;
                               port: port_number
                             end; {destination address)
```

The following operations are available on the BBP interface:

```
function send_message(to     : destination _address;
                      msg     : message;
                      service : s_option;
                      length  : msg_size
                      ): send_status;
```

The returned values of the function 'send_ message(...)' have the following meaning:

i) ' ok': all the words composing the message 'msg' have been accepted by the destination station; note that an 'ok ' response implies that there is a good chance that the process listening on 'to' will receive the message;

ii) 'absent' : message not delivered. The destination station is not available;

iii) 'not-done': message not delivered. The destination station is either receiving from another source or is receiving from no one;

iv) 'unable': some Ring malfunctions occurred during the transmission of the message; the message may or may not have reached the destination.

```
      procedure receive_message(at            : port_ number;
                               var src_ stn   : station_address;
                                var msg       : message
{
      This procedure receives in  msg  a message
       directed to port  at  from  src stn

}
      procedure get_port(var port: port_number);
{
      This procedure returns the name of a free port in  'port'.
}

      procedure release_port(port: port_number);
{
      This procedure adds  port to the pool of free ports.
}
```

BBP implementation

The following declarations of constants and data types are internal to the BBP implementation.

```
const
      MAX_ RETRY        = ...; {implementation dependent constant,
                                typically 1OOO}
      MIN_ RETRY        = ...; {implementation dependent constant,
                                typically 250}
      X_PKT_TIME_OUT    = ...; {implementation dependent constant indicating
                                the maximum time taken for a minipacket to
                                return to the sender}
      R_BLK_TIME_OUT    = ...; implementation dependent constant indicating
                                the maximum time taken for the transmission
                                 of a Basic block
      H_ID              = B'1OO1'; {bit pattern identifying the Basic Block
                                 header}

type
      header    =  record
                     id     : H_ID;
                      b_type: s_option;
                      length: msg_size
                   end;{header is two bytes long}

     Basic Block =  record
                     hdr_pkt   : header;
                     route_ pkt: port_number;
                     Data      : message;
                     checksum  : integer
                   end; Basic_ Block
```

The function send_message(...) is implemented as follows.

```
function send_ message(to      : process_ address;
                       msg     : message;
                       service: s_ option;
                       length : msg_size
                        ): send_status;

var
     Block:      Basic- Block;

    procedure Build_ Basic_ Block;
    function  Transmit _Basic_ Block: send_status;
       {Nested procedure and function, see bodies below.}

begin
    ring.request;      {acquire exclusive use of the Ring for transmission}
    Build_ Basic_ Block;
    send_message := Transmit_ Basic_ Block;
    ring.release       {release the Ring}
end; {send_message}

procedure Build Basic_Block;
{
This procedure builds an object of type Basic Block
and places the checksum in the checksum fie1d appropriately.
}
function EACCl6(B: Basic Block): integer; {Nested function, see body below.}

begin
with Block do
begin
hdr_pkt.b_type := service;
hdr_pkt.length := size;
route_ pkt := to.port;
Data.msg := msg.msg;
if service = no-checksum then checksum := 0
else
checksum := EACCl6(Block) end {with}
end; {Build_Basic_ Block}

function EACCl6(B: Basic_ Block): integer;
{

    This function computes the checksum
    for an object B of type Basic_ Block.
}
begin
    EACCl6 := {The 16 bit End Around Carry Checksum
               of the Basic Block B from the header up to
               the last word of the message}
end; {EACC16}

function Transmit_Basic Block: send status;
{

    This function attempts to transmit a Basic Block; it returns the result
```

```
     of the Basic Block transmission.
}
type
     n_retry  = MIN RETRY. MAX_RETRY;
     t status = (busy, accepted, deaf, ignored, transmission-error, time-out);

var
     t    :    t status;
     retry: n_retry;
     index: integer;

     function transmit(w: word, n: n retry): t status; {Nested function, see
     body below.}

begin                          {Transmit Basic Block}
     retry := MAX RETRY;        {the receiver might be listening to some other
                                source, so try several times to get acceptance}
case transmit(Block.hdr_pkt, retry) of
ignored:
     Transmit Basic Block := absent;
deaf, busy, transmission-error, time-out:
     Transmit Basic Block := not-done;
accepted:
     begin                     {transmit route packet}
          retry := MIN_RETRY; {reduce the number of possible retransmissions
                               since by now the receiver should be concentrated
                               on this source}
          case transmit(Block.route_pkt, retry) of
      ignored:
          Transmit Basic Block := absent;
      deaf, busy, transmission-error, time-out:
          Transmit Basic Block := not done;
      accepted:
          begin                {transmit message}
                index := 0;
                t := accepted;
          while (index <= length and t = accepted) do
          begin
                index := index + 1;
                t := transmit(Block.Data.msg[index], retry);
                case  t of
          ignored:
                Transmit Basic Block := absent;
          deaf, busy, transmission-error, time-out:
                Transmit Basic Block := not-done;
          accepted:
                ;
                end            {case}
```

```
            end;          {while}
            if (index = length and t=accepted) then
            begin         {transmit checksum}
               case transmit(Block.checksum, retry) of
            ignored:
                Transmit_Basic Block := absent;
            busy, deaf:
                Transmit_Basic Block := not-done;
            transmission-error, time-out:
                Transmit Basic Block := unable;
            accepted:
                Transmit Basic Block := ok
               end     - {case}
           end            {transmit checksum}
         end              {transmit message}
       end                { case}
   end                    {transmit route packet
   end                    {case}
end;                      {Transmit Basic Block}

function transmit (w: word, count: n_retry): t status;
{
    This function attempts to transmit a  word  by means of the  transmit
    packet(...)  primitive.
    The input parameter  w  specifies the word to be transmitted;
    the input parameter  count  specifies the number of retries to be
    attempted in case of unsuccessful termination of the transmit_packet(...)
    primitive.
}
const
    t = K PKT_TIME_OUT;

var
    n : n retry;
begin
    n := count;
    repeat
        within t do
           transmit := transmit_packet(to.station,w)
           <EXPIRED ? (transmit := time-out; exit)>
           n  := n-1
    until (transmit = accepted or transmit = ignored
                 or transmit = transmission-error or n = 0);
end; {transmit}
```

The procedures 'get_port(...)' and 'release port(...)' are implemented as follows:

```
procedure get_port(var p: port number);
begin
     pool .getport( p)
end;

procedure release_port(p: port_number);
begin
      pool .releaseport( p)
end;
```

Receive Block Process implementation.

```
process Receive_Block(active: Active Ports);
{
      The Receive Block process receives Basic Blocks, one at the time, from
      the Ring. It checks whether it is possible to deliver the message
      contained in the Basic Block to its destination port, and if this is the
      case it receives and delivers the message to that port, else it discards
      the message.
}
const
      NO ONE   = 0;
      ANY ONE  = 255;
      t        = R_BLK_TIME_OUT;
var
      Block   : Basic Block;
      service: s option;
      hdr     : header;
      stn     : station address;

      procedure get_header(var h: header; var s: station_address);
      procedure get_Data(L: integer; var N: message);
      function  compute_checksum(B: Basic Block): Boolean;
               {Nested procedures and functions, see bodies below.}

begin
      cycle
            set_SSR(ANY ONE);
            get_header(hdr,stn);
            set_SSR(stn);
            service := hdr.b_type;
            Block.hdr_pkt := hdr;
      within  t  do
      begin
            receive packet(stn, Block.route_pkt); {receive destination port }
            if active.cbeck_active(Block.route_pkt) then
            begin     { receive message}
              get_Data(Block.hdr_pkt.length, Block.Data);
              receive_ packet(stn, Block.checksum);
              if compute_checksum(Block)  then
                 active.send(Block.data, stn)
              else              {checksum failed}
                   reset         {go back to the beginning of cycle}
            end                 { receive message}
            else                {port not active}
                   reset         {go back to the beginning of the cycle}
          end <EXPIRED? (reset)>{within}
      end                        {cycle}


end;

procedure get_header(var b: header; var s: station_address);
{
    This procedure returns a valid header in 'h' and its source
    station address in 's'.
}
begin
    repeat
```

```
     receive packet(s, h);
    until (h.id = H ID);
end; {get_header}

procedure get_Data(L: msg_size; var M: message);
{
    This procedure returns in  M  a message of length  'L'+l.
}
var i: integer; s: station address;

begin
    for i = 0 to L do
        receive_packet(s, M[i]);
end; {get_Data}

procedure reset;
{
    This procedure sets the Source Select Register to 0 for a fixed interval
    of time. Consequently, during this interval, each incoming minipacket
    will be acknowledged  busy  and rejected.
}
const T = ...;          {implementation dependent constant indicating the
                         time taken to transmit the number of minipackets
                         equal to MIN RETRY}
begin
    set SSR(NO_ONE);
    kill time-out;
    wait(T)               {in the hope that the sender
                           will get a  not-done  response}
 end; {reset}

function compute_checksum(BB: Basic_Block): Boolean;
{
    This function computes the checksum of the Basic Block
     BB  and compares it with the checksum contained in  'BB'
}.
     function EACC16(B: Basic_Block): integer;
      {Nested function, body as on page 10.}
begin
     case BB.b type of
checksum:
     compute checksum :-- (BB.checksum = EACC16(BB));
no-checksum:
      compute checksun :-- true
      end  {case}
end; {compute_checksum}
```

Monitors

Assume that a monitor queue can contain more than one process.

```
type Resource = monitor
var
      free: boolean;
      q   : queue;

procedure entry request;
begin
```

```
      If not free then delay(q);
      free := false
end;

procedure entry release;
begin
      free := true;
      continue( q)
end;

begin free := true end;

type Port Pool = monitor;

var
      port : port_number;
      p_set: set of port_number;
      empty: queue;

procedure entry getport(var p: port number);
begin
      if p_set = [] then delay(empty);
      p := 0;
      while p not in p_set do p := p+l;
      p set := p set - [p]
end; {getport}

procedure releaseport(p: port number);
begin
      p_set := p set + [p]
      continue( empty)
end;  {end releaseport}

begin
      port : = 0;
      while port <= MAX PORT do
          p set := p set + [port]
end;

type Active_Ports = monitor;

var
    mg    : message;
    pt    : port number;
    st    : station address;
    active: set of port_number;
    wait_q: queue

procedure entry receive message(at    : port_number;
                                var src stn: station_address;
                                var msg     : message
                                );

begin
    active := active + [at];
    delay(wait q);
    while at <> Pt do delay(wait_q);
    msg := mg;
```

```
    src stn := st;
    active := active - [at]
end; {receive_message]

function entry check active (at: port_number): Boolean;
begin
    if at in active then
    begin
        pt := at;
        check active := true
    end
    else
        check active := false
end; {check_active]

procedure entry send(msg: message; src: station address);
begin
    mg := msg;
    st := src;
    continue_all(wait_q)  {wakes up all of the waiting processes]
end;

begin active := [] end  {empty initially]

Monitor  creation

var
    ring  : Resource;
    pool  : Port_Pool;
    active: Active_ Ports;
```