```
--** an occam model of Oyvind's 'XCHAN' (CPA 2012).
--
--  We model an XCHAN (buffered or unbuffered) with an occam process:
--
--  [@text
--  :
--                     ----------------------
--            ready   |                      |
--          ----<----|                      |   out
--                    |      xchan (n)       |---->----
--          ---->----|                      |
--             in     |                      |
--                     ----------------------
--  ]
--
--  Application messages flow from 'in' to 'out'.
--  The device signals [@code TRUE] down the 'ready' channel when, and only
--  when, it will accept input: this signal must be taken by a writing
--  process before sending anything.
--  Events on 'ready' and 'in' strictly alternate, starting with 'ready'.
--  The buffering capacity is 'n' (>= 0).
--
--  A reading process simply reads from the output channel of the device.
--  [@em Warning:] a current implementation restriction means reading from
--  a [@em zero-buffered] XCHAN must be done twice, discarding the first item.
--
--  A writing process has three choices: an [@em asynchronous] write (that
--  immediately returns with an indication of whether the write succeeded),
--  a [@em synchronous] write (that does not return until the write succeeds),
--  or [@code ALT] on the 'ready' channel and other events (until the 'ready'
--  is signalled and, then, write the message).
--
--  This module provides two versions of XCHAN: [@ref xchan.a] and
--  [@ref xchan.b].
```

```
--  They differ only in behaviour if their buffering capacity is set
--  to zero: [@ref xchan.b] is better (but slightly more expensive).
--
--  The structure of this module is described in [@ref MODULE.STRUCTURE].


--* First, the data type carried by the XCHAN ([@ref STUFF]) is declared,
--  together with a constant of that type ([@ref DUMMY.STUFF]) (which can have
--  any value).
--  Because occam-pi does not currently support [@em generic] types, these must
--  be edited to the type required by the application using the XCHAN.
--
--  Next come the [@em basic] mechanisms for writing to XCHANs:
--  [@ref xchan.async.write], [@ref xchan.blocking.write] and [@ref PATTERN.A].
--  Beware that these do not work for the second [@em (better)] implementation
--  of an [@em unbuffered] XCHAN - see below.
--
--  The simplest XCHAN implementation is for a buffered XCHAN with capacity 1
--  ([@ref xchan.one]) and this comes next.
--
--  Then follow a series of [@em blocking buffer] processes
--  ([@ref buffer.one], [@ref buffer.a], [@ref buffer.b], [@ref buffer.c]
--  and [@ref buffer.d]).
--  These are [@em private] processes, used in the implementation below.
--
--  A buffered XCHAN with capacity greater than 1 is just a [@em one-place]
--  XCHAN ([@ref xchan.one]) pipelined into a blocking buffer ([@ref buffer.d]),
--  that provides the rest of the capacity.
--  This is wrapped into ([@ref xchan.buffered.a]), which is a buffered XCHAN
--  with capacity greater than or equal to 1.
--
--  A modest optimisation on the above follows: [@ref xchan.buffered.b].
--  This is preceded by two [@em private] support processes:
--  [@ref buffer.one.bool] and [@ref x.ring.buffer].
--
```

```
--  Now follow two implementations for an unbuffered XCHAN:
--  [@em simple unbuffered] ([@ref xchan.zero.a]) and
--  [@em better unbuffered] ([@ref xchan.zero.b]).
--  The latter requires small changes in the mechanisms for writing to it:
--  [@ref xchan.async.write.b], [@ref xchan.blocking.write.b]
--  and [@ref PATTERN.B].
--
--  Unfortunately, both the above unbuffered XCHAN implementations require
--  [@em extended output]: a feature not yet supported by occam-pi.
--  There is a simple [@ref WORK.AROUND], given next by [@ref xchan.zero.a2]
--  and [@ref xchan.zero.b2] (for the simple and better behaviour,
--  respectively).
--  However, these require a change to the way the unbuffered XCHAN is read
--  ([@ref xchan.zero.sync.read] or [@ref PATTERN.C]).
--
--  [@em Note:] the goal is to have the same mechanisms for reading and writing
--  XCHANs, regardless of whether they are buffered or unbuffered.
--  If occam-pi supported [@em extended output], this could be achieved:
--  reading would be an [@em occam primitive] read ([@code ?]),
--  from the output channel of the XCHAN, and
--  writing would use either [@ref xchan.async.write.b],
--  [@ref xchan.blocking.write.b] or [@ref PATTERN.B].
--
--  As things stand, an [@em occam primitive] read ([@code ?]) can only be
--  used for [@em buffered] XCHANs but the [@em read-discard-read-keep]
--  pattern ([@ref xchan.zero.sync.read] or [@ref PATTERN.C]) is needed for
--  [@em unbuffered] XCHANs.
--
--  For writing, [@ref xchan.async.write.b], [@ref xchan.blocking.write.b] or
--  [@ref PATTERN.B] can be used for [@em all] XCHANs.
--  However, [@ref xchan.async.write], [@ref xchan.blocking.write] or
--  [@ref PATTERN.A] are marginally more efficient for [@em buffered] XCHANs
--  and for the first version [@em unbuffered] XCHANs ([@ref xchan.zero.a2],
--  [@em simple behaviour]), but cannot be used the second version
--  ([@ref xchan.zero.b2], [@em better behaviour]).
```

```
--
--  Finally, [@ref xchan.a] and [@ref xchan.b] are processes implementing
--  XCHANs of any capacity (i.e. [@em buffered] or [@em unbuffered]),
--  offering different implementation choices taking into account
--  the points in the last two paragraphs.
--  Please see their documentation for how they [@em must] be used.
--
VAL INT MODULE.STRUCTURE IS 0:


--* occam-pi currently has no generic types, so we must define code to operate
--  on some specific type.  To build an XCHAN for another type, change this
--  declaration to what you want.  See also [@ref DUMMY.STUFF].
--
DATA TYPE STUFF IS REAL64:


--* This is currently needed to support reading from an unbuffered XCHAN
--  (see [@ref WORK.AROUND], [@ref xchan.zero.a2] and [@ref xchan.zero.b2]).
--
--  [@em Any] value of the [@ref STUFF] type may be chosen for this constant.
--  [@em Implementor's note:] choose a value with minimal memory footprint.
--
VAL STUFF DUMMY.STUFF IS 0.0:


--* This is an [@em asynchronous] write for an XCHAN.  It never blocks and
--  returns with an indication of whether it was able to perform the write.
--
--  Commonly, this is the first thing tried by a writing process: if it
--  fails, then the [@code ALT]ing pattern on the [@em ready] channel
--  may be engaged ([@ref PATTERN.A]) rather than continued attempts
--  to write using this process.
--
--  @param data This is the message to be written.
```

```
--  @param success This indicates whether the write happened.
--  @param ready.xchan This is the [@em ready] channel from the XCHAN device.
--  @param to.xchan This is the [@em input] channel to the XCHAN device.
--
PROC xchan.async.write (VAL STUFF data, BOOL success,
                        CHAN BOOL ready.xchan?, CHAN STUFF to.xchan!)
  PRI ALT
    BOOL any:
    ready.xchan ? any
      SEQ
        to.xchan ! data
        success := TRUE
    SKIP
      success := FALSE
:


--* This is a [@em synchronous] write for an XCHAN.  It will block until
--  the XCHAN is able to take the message.
--
--  This procedure would not normally be used (since a primitive channel
--  or conventional blocking FIFO process would be more efficient).
--  It is included for completeness.
--
--  @param data This is the message to be written.
--  @param ready.xchan This is the [@em ready] channel from the XCHAN device.
--  @param to.xchan This is the [@em input] channel to the XCHAN device.
--
PROC xchan.blocking.write (VAL STUFF data, CHAN BOOL ready.xchan?,
                           CHAN STUFF to.xchan!)
  BOOL any:
  SEQ
    ready.xchan ? any
    to.xchan ! data
:
```

```
--* This is the third choice for writing to an XCHAN: wait for the device
--  to become [@em ready], whilst servicing other events.  For example:
--
--  [@code
--  :
--    -- Pattern 'A'
--    INITIAL BOOL wanting.to.write IS TRUE:
--    WHILE wanting.to.write
--      ALT                         -- or PRI ALT
--        BOOL any:
--        ready.xchan ? any
--          SEQ
--            to.xchan ! data
--            wanting.to.write := FALSE
--        ...  process other guards (which may change 'data')
--  ]
--
--  The writer may adopt this pattern at any time: there is no obligation
--  to try an [@ref xchan.async.write] first.
--
--  Note that there is no obligation on the writer to send the data it
--  originally had; it is free to discard that and send, for example, data
--  acquired since it started waiting.
--
VAL INT PATTERN.A IS 0:


--* This is a [@em one-place buffered] XCHAN process.
--
--  Its behaviour is exactly that of an [@em auto-prompter], a common occam
--  idiom.
--
--  @param ready This is signalled (with [@code TRUE]) when, and only when,
```

```
--    data on the [@ref in] channel can be taken.  This signal [@em must]
--    be taken before data may be sent.
--  @param in Data input
--  @param out Data output
--
PROC xchan.one (CHAN BOOL ready!, CHAN STUFF in?, out!)
  WHILE TRUE
    STUFF x:
    SEQ
      ready ! TRUE
      in ? x
      out ! x
:


--* To build a [@em buffered] XCHAN process with application-defined capacity,
--  we just need a [@em one-place buffered] XCHAN process ([@ref xchan.one])
--  pipelined with a standard blocking buffer (with capacity one less than
--  required for the [@em buffered] XCHAN).  First, we build the latter.
--
VAL INT BUFFERED.XCHAN.CAPACITY IS 0:


--* This is a standard [@em one-place blocking buffer], commonly known as
--  an [@em id-process].  It just copies input to output.
--
-- @param in Data input
-- @param out Data output
--
PROC buffer.one (CHAN STUFF in?, out!)
  WHILE TRUE
    STUFF x:
    SEQ
      in ? x
      out ! x
```

```
:


#IF FALSE

--* This is a standard [@em blocking buffer] process with application-defined
--  capacity, implemented as a pipeline of [@em one-place blocking buffers]
--  ([@ref buffer.one]).  For this implementation, the capacity must be more
--  than one.
--
--  [@em Warning:] this process does not compile (because occam-pi runtime
--  sized channel arrays currently have to be built from arrays of mobile
--  channel-ends - see [@ref buffer.b]).  It is presented here for
--  easier understanding of its code (and because occam-pi will [@em eventually]
--  compile it).
--
--  @param max The maximum capacity of this buffer ([@code max >= 1]).
--  @param in Data input
--  @param out Data output
PROC buffer.a (VAL INT max, CHAN STUFF in?, out!)
  IF
    max < 1
      STOP                    -- illegal parameter value
    max = 1
      buffer.one (in?, out!)
    TRUE                      -- DEDUCE: max >= 2
      [max - 1]CHAN STUFF c:   -- runtime sized channel array (will not compile)
      PAR
        buffer.one (in?, c[0]!)
        PAR i = 0 FOR max - 2
          buffer.one (c[i]?, c[i+1]!)
        buffer.one (c[max - 2]?, out!)
:

#ENDIF
```

```
--* To implement [@ref buffer.two.plus.a] in a way that compiles, we must
--  build the channel array from mobile channel-ends.  This declares the
--  needed mobile channel type (a trivial structure with one field).
--  This is a [@em private] declaration, used only for the implementation
--  of [@ref buffer.b].
--
CHAN TYPE STUFF.CHAN
  MOBILE RECORD
    CHAN STUFF c?:
:


--* This is a standard [@em blocking buffer] process with application-defined
--  capacity, implemented as a pipeline of [@em one-place blocking buffers]
--  ([@ref buffer.one]).  This implementation will compile and run correctly.
--
--  @param max The maximum capacity of this buffer ([@code max >= 1]).
--  @param in Data input
--  @param out Data output
--
PROC buffer.b (VAL INT max, CHAN STUFF in?, out!)
  IF
    max < 1
      STOP                      -- illegal parameter value
    max = 1
      buffer.one (in?, out!)
    TRUE                        -- DEDUCE: max >= 2
      INITIAL MOBILE []STUFF.CHAN! c0 IS MOBILE [max]STUFF.CHAN!:
      INITIAL MOBILE []STUFF.CHAN? c1 IS MOBILE [max]STUFF.CHAN?:
      SEQ
        SEQ i = 0 FOR max
          c0[i], c1[i] := MOBILE STUFF.CHAN   -- connect the ends
        PAR
```

```
          STUFF.CHAN! x IS c0[0]:
          buffer.one (in?, x[c]!)
          PAR i = 0 FOR max - 3
            STUFF.CHAN? x IS c1[i]:
            STUFF.CHAN! y IS c0[i + 1]:
            buffer.one (x[c]?, y[c]!)
          STUFF.CHAN? x IS c1[max - 3]:
          buffer.one (x[c]?, out!)
:


--* We can build a buffer process in a more serial way (that will be much
--  more efficient if implemented by software).  In concept, it is slightly
--  more complicated (but only [@em slightly]) than a pipeline of one-place
--  blocking buffers.  It is taken from the [@em "Concurrency Design and
--  Practice"] course at the University of Kent.
--
VAL INT BUFFER.SERIAL IS 0:


--* This is a standard [@em blocking buffer] process with application-defined
--  capacity, implemented as classic [@em ring buffer].  However, this needs
--  a [@em request] channel that the reader process must signal before reading.
--
--  @param max The maximum capacity of this buffer ([@code max >= 1]).
--  @param in Data input
--  @param out Data output
--  @param request The reader must signal (value irrelevant) on this before reading.
--
PROC buffer.c (VAL INT max, CHAN STUFF in?, out!, CHAN BOOL request?)
  IF
    max < 1
      STOP                    -- illegal parameter value
    max = 1
      WHILE TRUE              -- this case does not need separate coding,
```

```
        STUFF x:                -- since its logic is implemented by the general
        SEQ                     -- code in the next condition; it's added here
          in ? x                -- to show the logic for this trivial case (and
          BOOL any:             -- for efficiency).
          request ? any
          out ! x
    TRUE                        -- DEDUCE: max >= 1
      INITIAL MOBILE []STUFF hold IS MOBILE [max]STUFF:
      INITIAL INT size IS 0:      -- current size of buffer
      INITIAL INT lo IS 0:        -- index of oldest item in buffer (if size > 0)
      INITIAL INT hi IS 0:        -- index of next free slot (if size < max)
      WHILE TRUE
        ALT
          (size < max) & in ? hold[hi]
            SEQ
              hi := (hi + 1)\max
              size := size + 1
          BOOL any:
          (size > 0) & request ? any
            SEQ
              out ! hold[lo]
              lo := (lo + 1)\max
              size := size - 1
:


--* This is a standard [@em blocking buffer] process with application-defined
--  capacity, implemented as classic [@em ring buffer].  It eliminates the
--  need for a [@em request] channel by pipelining [@ref buffer.c] with an
--  [@em auto-prompter] (which is, of course, [@ref xchan.one]).
--
--  @param max The maximum capacity of this buffer ([@code max >= 1]).
--  @param in Data input
--  @param out Data output
--
```

```
PROC buffer.d (VAL INT max, CHAN STUFF in?, out!)
  IF
    max < 1
      STOP                    -- illegal parameter value
    max = 1
      buffer.one (in?, out!)
    TRUE                      -- DEDUCE: max >= 2
      CHAN BOOL request:
      CHAN STUFF c:
      PAR
        buffer.c (max - 1, in?, c!, request?)
        xchan.one (request!, c?, out!)
:


--* This is a [@em one-buffered] XCHAN process with application-defined capacity.
--
--  It is built from a [@em one-place buffered] XCHAN process ([@ref xchan.one])
--  pipelined with a standard blocking buffer ([@ref buffer.d]).
--
--  @param max The maximum capacity of this XCHAN ([@code max >= 1]).
--  @param ready This is signalled (with [@code TRUE]) when, and only when,
--    data on the [@ref in] channel can be taken.  This signal [@em must]
--    be taken before data may be sent.
--  @param in Data input
--  @param out Data output
--
PROC xchan.buffered.a (VAL INT max, CHAN BOOL ready!, CHAN STUFF in?, out!)
  IF
    max < 1
      STOP                    -- illegal parameter value
    max = 1
      xchan.one (ready!, in?, out!)
    TRUE                      -- DEDUCE: max >= 2
      CHAN STUFF c:
```

```
      PAR
        xchan.one (ready!, in?, c!)
        buffer.d (max - 1, c?, out!)
:


--* [@em Note:] messages passing through [@ref xchan.buffered.a]
--  pass through three hops (for capacities greater than 2).
--  The version originally devised ([@ref xchan.buffered.b]) makes messages
--  pass through only two hops.  However, it makes the [@em ready] signal also
--  pass through two hops - so may not be any faster!  First, we need an
--  [@em id-process] for those [@em ready] signals ([@ref buffer.one.bool])
--  and, then, a [@em ring buffer] folded with XCHAN code ([@ref x.ring.buffer]).
--
VAL INT OPTIMISED.BUFFERED.XCHAN IS 0:


--* This is a standard [@em one-place blocking buffer], commonly known as
--  an [@em id-process].  It just copies input to output.
--
-- @param in Data input
-- @param out Data output
--
PROC buffer.one.bool (CHAN BOOL in?, out!)
  WHILE TRUE
    BOOL x:
    SEQ
      in ? x
      out ! x
:


--* Standard ring buffer modified to provide an XCHAN ready signal.
--
--  This is a service process for the [@ref xchan.buffered.b] (below).
```

```
--  It should not be used directly by systems.
--
--  There must be an [@ref xchan.one] [@em auto-prompter] driving
--  the [@code prompt] and [@code out] channels.
--  There must be a [@ref buffer.one.bool] [@em id-process] forwarding
--  [@code ready] signals.
--
--  A 'ready' signal is offered if and only if space is available
--  to buffer another item of data.  Events 'ready' and 'in' must
--  strictly alternate, starting with 'ready'.
--
--  To write to 'in', a 'ready' signal (forwarded by 'id.bool')
--  must first be accepted by the writer.  Disregarding this
--  protocol leads to this process [@code STOP]ping and probable deadlock.
--
-- @param max Size of the buffer (>= 1)
-- @param in Data input
-- @param out Data output
-- @param prompt Reader must prompt for output
-- @param ready Writer must take this signal before writing
--
PROC x.ring.buffer (VAL INT max, CHAN STUFF in?, out!,
                    CHAN BOOL prompt?, ready!)             -- , error!)
  --
  -- Note: if (#ready! = #in?) and the writer to 'in' follows the required
  --       protocol, all 'ready' signals generated by this process have
  --       been taken by the writer process and the accompanying 'id.bool'
  --       process is waiting for the next 'ready' from here (i.e. the
  --       next 'ready' will not block).  This holds in all states of this
  --       process (not just at the start of its loop).
  --
  INITIAL MOBILE []STUFF buffer IS MOBILE [max]STUFF:
  INT lo, hi, size:
  SEQ
    lo, hi, size := 0, 0, 0
```

```
ready ! TRUE                -- DEDUCE: will not block ('id.bool' is waiting)
WHILE TRUE
  -- INVARIANT: (size < max) <==> (#ready! = #in? + 1)
  -- INVARIANT: (size = max) <==> (#ready! = #in?)
  ALT
    STUFF any:
    (size = max) & in ? any     -- protocol violation (by writer)
      SEQ
        -- error ! FALSE           -- this error is intended to be fatal
        STOP                    -- if skipped, a more complex loop invariant is needed
    (size < max) & in ? buffer[hi]
      -- DEDUCE: #ready! = #in?
      -- assume: writer has cleared previous 'ready' from 'id.bool'
      --         (see above note).  Otherwise there has been a protocol
      --         violation (which cannot be detected here).
      SEQ
        hi := (hi + 1)\max
        size := size + 1
        IF
          size < max
            ready ! TRUE   -- DEDUCE: will not block ('id.bool' is waiting)
                           -- DEDUCE: (size < max) AND (#ready! = #in? + 1)
          TRUE
            SKIP           -- DEDUCE: (size = max) AND (#ready! = #in?)
    BOOL any:
    (size > 0) & prompt ? any
      SEQ
        out ! buffer[lo]
        lo := (lo + 1)\max
        IF
          size < max
            SKIP           -- DEDUCE: (size < max) AND (#ready! = #in? + 1)
          TRUE
                           -- DEDUCE: (size = max) AND (#ready! = #in?)
            ready ! TRUE   -- DEDUCE: will not block ('id.bool' is waiting)
```

```
                                -- DEDUCE: (size = max) AND (#ready! = #in? + 1)
            size := size - 1
            -- DEDUCE: (size < max) AND (#ready! = #in? + 1)
:


--* This is a [@em one-buffered] XCHAN process with application-defined capacity.
--
--  [@em Historical note:] this was the original version (just before CPA 2012).
--
--  @param max The maximum capacity of this XCHAN ([@code max >= 1]).
--  @param ready This is signalled (with [@code TRUE]) when, and only when,
--    data on the [@ref in] channel can be taken.  This signal [@em must]
--    be taken before data may be sent.
--  @param in Data input
--  @param out Data output
--
PROC xchan.buffered.b (VAL INT max, CHAN BOOL ready!, CHAN STUFF in?, out!)
  IF
    max < 1
      STOP                      -- illegal parameter value
    max = 1
      xchan.one (ready!, in?, out!)
    TRUE                        -- DEDUCE: max >= 2
      CHAN BOOL a, r:
      CHAN STUFF b:
      PAR
        x.ring.buffer (max - 1, in?, b!, a?, r!)        -- , error!)
        xchan.one (a!, b?, out!)
        buffer.one.bool (r?, ready!)
:


--* Next come zero-buffered XCHANs.
--
```

```
VAL INT ZERO.BUFFERED IS 0:


#IF FALSE

--* This is a zero-buffered XCHAN (simple behaviour).
--
--  It fishes for a reader by offering an [@em extended output] ([@code out !!]).
--  When a reader is caught, it fishes for a writer by signalling on [@em ready].
--  When it has caught both, the data is transferred.  No buffering is
--  introduced by this process in the connection between its writer and
--  reader.
--
--  Its weakness is that a reader is sought before there is any indication
--  that a write is pending.  This is addressed in [@ref xchan.zero.b].
--
--  [@em Warning:] this process will not compile since [@em extended output]
--  is not yet supported by occam-pi.  See [@ref xchan.zero.a2] for a
--  work-around.
--
--  @param ready This is signalled (with [@code TRUE]) when, and only when,
--    a reader is committed to read.  This signal [@em must] be taken before
--    data may be sent - the sender is guaranteed that the reader will accept.
--  @param in Data input
--  @param out Data output
--
PROC xchan.zero.a (CHAN BOOL ready!, CHAN STUFF in?, out!)
  WHILE TRUE
    STUFF x:
    out !!              -- look for a reader (will not yet compile)
      SEQ
        ready ! TRUE    -- let the writer know a reader is committed
        in ? x          -- the writer delivers (may not be immediate)
        !! x            -- reader is committed to take this
:
```

```
--* This is a zero-buffered XCHAN (better behaviour).
--
--  It fixes the weakness noted in the documentation for [@ref xchan.zero.a].
--  However, it requires slightly different logic for an application process
--  that writes to it ([@ref xchan.async.write.b], [@ref xchan.blocking.write.b]
--  and [@ref PATTERN.B]).  [@em Note:] these revised processes and pattern
--  may also be used with all other versions of buffered and unbuffered XCHANs
--  (with only a slight overhead cost).
--
--  For a [@em writer] to this process, the values from its [@em ready] channel are
--  significant.  This is because this process first fishes for writer by
--  sending a [@code FALSE] signal on [@em ready].  As normal, when and only when the
--  writer has something to send, it waits for a signal on [@em ready].
--  However, if that signal was [@code FALSE], the writer must keep waiting until it
--  gets a [@code TRUE].  All this waiting can, of course, be done whilst processing
--  other events (using [@code ALT]).  Meanwhile a writer, by accepting the [@code FALSE],
--  lets this process know that it has something to send and this process then
--  fishes for a [@em reader] (using [@em extended output], [@code out !!]).
--  Once found, the reader is committed and this process now sends [@code TRUE]
--  on [@em ready] to encourage the writer to write something (which need not,
--  of course, be what it originally had to send).  The writer writes, this process
--  forwards, the reader reads and no buffering semantics have been introduced.
--
--  The reader from an [@ref xchan.zero.b] just does a normal read, as before.
--  Disregarding the new writer protocol leads to deadlock.  Checking
--  that a writer has followed this protocol can be done by a simple
--  visual check of the code (to ensure a write follows, and only follows,
--  a [@code TRUE] on 'ready') or, automatically, by a specialised tool or simple
--  model check.
--
--  [@em Warning:] this process will not compile since [@em extended output]
--  is not yet supported by occam-pi.  See [@ref xchan.zero.b2] for a
--  work-around.
```

```
--
--  @param ready This is signalled with [@code TRUE] when, and only when, a reader
--    is committed to read.  Prior to that, a [@code FALSE] is signalled that should
--    only be accepted when a writer has something to write.  The writer must
--    still wait for the [@code TRUE] signal before writing - when this happens, the
--    writer is guaranteed that the reader will read.
--  @param in Data input
--  @param out Data output
--
PROC xchan.zero.b (CHAN BOOL ready!, CHAN STUFF in?, out!)
  WHILE TRUE
    SEQ
      ready ! FALSE          -- taken by a writer who wants to write
      STUFF x:
      out !!                 -- look for a reader (will not yet compile)
        SEQ
          ready ! TRUE       -- let the writer know a reader is committed
          in ? x             -- the writer delivers (may not be immediate)
          !! x               -- reader is committed to take this
:

#ENDIF


--* This is an [@em asynchronous] write for a [@ref xchan.zero.b]
--  (a zero-buffered XCHAN).
--  It never blocks and returns with an indication of whether it was able
--  to perform the write.
--
--  A writer could simply keep using this process when it has data to send.
--  There will be at least one FALSE result (maybe many) before a TRUE.
--  It is up to the writer whether to keep sending the same data until success
--  or fresh data.  When a write has succeeded, the writer can be assured the
--  reader has taken it (or is about to take it).
--
```

```
--  Commonly, this is the first thing tried by a writing process: if it fails,
--  then the [@code ALT]ing [@ref PATTERN.B] on the [@em ready] channel may
--  be engaged (rather than continued attempts to write using this process).
--
--  @param data This is the message to be written.
--  @param success This indicates whether the write happened.
--  @param ready.xchan This is the [@em ready] channel from the XCHAN device.
--  @param to.xchan This is the [@em input] channel to the XCHAN device.
--
PROC xchan.async.write.b (VAL STUFF data, BOOL success,
                          CHAN BOOL ready.xchan?, CHAN STUFF to.xchan!)
  PRI ALT
    ready.xchan ? success
      IF
        success
          to.xchan ! data
        TRUE
          SKIP
    SKIP
      success := FALSE
:


--* This is a [@em synchronous] write for an XCHAN.  It will block until
--  the XCHAN is able to take the message.
--
--  This procedure would not normally be used (since a primitive channel
--  or conventional blocking FIFO process would be more efficient).
--  It is included here for completeness.
--
--  [@em Note:] the loop in the code is not needed (see the comments).
--  However, if this process is used for writing to buffered XCHANS or the
--  previous version of an unbuffered XCHAN ([@ref xchan.zero.a]), the
--  comments do not apply: there will only be one TRUE signal and this coding
--  still works.
```

```
--
--  @param data This is the message to be written.
--  @param ready.xchan This is the [@em ready] channel from the XCHAN device.
--  @param to.xchan This is the [@em input] channel to the XCHAN device.
--
PROC xchan.blocking.write.b (VAL STUFF data, CHAN BOOL ready.xchan?,
                             CHAN STUFF to.xchan!)
  BOOL ok:
  SEQ
    ready.xchan ? ok          -- this will be FALSE
    WHILE NOT ok
      ready.xchan ? ok        -- this will be TRUE
    to.xchan ! data
:


--* More useful may be the following strategy.
--  When a writer has something to send, listen on the [@em ready] channel
--  from the XCHAN: this lets the zero-buffered channel know that a writer is waiting.
--  Ignore [@code FALSE] [@em readys].
--  When [@code TRUE] is received from [@em ready], send the data.
--  The writer can be assured the reader has committed to take it:
--
--  [@code
--  :
--    -- Pattern 'B'
--    INITIAL BOOL wanting.to.write IS TRUE:
--    WHILE wanting.to.write
--      ALT                           -- or PRI ALT
--        BOOL ok:
--        ready.xchan ? ok
--          IF
--            ok                      -- means a reader is waiting
--              SEQ
--                to.xchan ! data
```

```
--                wanting.to.write := FALSE
--              TRUE
--                SKIP
--        ...  process other guards (which may change 'data')
--  ]
--
--  The writer may adopt this pattern at any time: there is no obligation
--  to try an [@ref xchan.async.write.b] first.
--
--  Note that there is no obligation on the writer to send the data it
--  originally had; it is free to discard that and send, for example, data
--  acquired since it started waiting.
--
VAL INT PATTERN.B IS 0:


--* Here are the work-arounds for processes [@ref xchan.zero.a] and
--  [@ref xchan.zero.b] (which do not compile because occam-pi does not yet
--  support [@em extended output]).  They require readers to read [@em twice],
--  discarding the first item ([@ref DUMMY.STUFF]).
--
VAL INT WORK.AROUND IS 0:


--* This is a zero-buffered XCHAN (simple behaviour).
--
--  See [@ref xchan.zero.a] for documentation.  This process works around the
--  current non-implementation of [@em extended output] in occam-pi by
--  applying the standard transformation (given in
--  [@link https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP/142 OEP 42]).
--
--  However, the transformation also requires a reader from this XCHAN to read
--  [@em twice], discarding the first item received ([@ref DUMMY.STUFF]).
--  See [@ref xchan.zero.sync.read] and [@ref PATTERN.C].
--
```

```
--  @param ready This is signalled (value irrelevant) when, and only when,
--    a reader is committed to read.  This signal [@em must] be taken before
--    data may be sent - the sender is guaranteed that the reader will accept.
--  @param in Data input
--  @param out This must always be read [@em twice], discarding the first item
--    received ([@ref DUMMY.STUFF]).
--    The first read may be part of an [@code ALT]; however, the second read
--    must be committed.
--
PROC xchan.zero.a2 (CHAN BOOL ready!, CHAN STUFF in?, out!)
  WHILE TRUE
    STUFF x:
    SEQ
      out ! DUMMY.STUFF     -- look for a reader
      ready ! TRUE          -- let the writer know a reader is committed
      in ? x                -- the writer delivers (may not be immediate)
      out ! x               -- reader is committed to take this
:


--* This is a zero-buffered XCHAN (better behaviour).
--
--  See [@ref xchan.zero.b] for documentation.  This process works around the
--  current non-implementation of [@em extended output] in occam-pi by
--  applying the standard transformation (given in
--  [@link https://www.cs.kent.ac.uk/research/groups/plas/wiki/OEP/142 OEP 42]).
--
--  However, the transformation also requires a reader from this XCHAN to read
--  [@em twice], discarding the first item received ([@ref DUMMY.STUFF]).
--  See [@ref xchan.zero.sync.read] and [@ref PATTERN.C].
--
--  @param ready This is signalled with TRUE when, and only when, a reader
--    is committed to read.  Prior to that, a FALSE is signalled that should
--    only be accepted when a writer has something to write.  The writer must
--    still wait for the TRUE signal before writing - when this happens, the
```

```
--    writer is guaranteed that the reader will read.
--  @param in Data input
--  @param out This must always be read [@em twice], discarding the first item
--    received.  The first read may be part of an [@code ALT]; however, the
--    second read must be committed.
--
PROC xchan.zero.b2 (CHAN BOOL ready!, CHAN STUFF in?, out!)
  WHILE TRUE
    STUFF x:
    SEQ
      ready ! FALSE         -- taken by a writer who wants to write
      out ! DUMMY.STUFF     -- look for a reader
      ready ! TRUE          -- let the writer know a reader is committed
      in ? x                -- the writer delivers (may not be immediate)
      out ! x               -- reader is committed to take this
:


--* This is a trivial process to perform a [@em blocking] read from the
--  current work arounds for zero buffered XCHANs ([@ref xchan.zero.a2] or
--  [@ref xchan.zero.b2]).  It reads from the XCHAN [@em twice], discarding
--  the first value and returning the second.  It's not really needed and
--  only included for completeness.
--
--  @param data This is what is read from the XCHAN.
--  @param from.xchan The channel from the XCHAN.
--
PROC xchan.zero.sync.read (STUFF data, CHAN STUFF from.xchan?)
  SEQ
    from.xchan ? data   -- indicates a writer has something
    from.xchan ? data   -- the actual data from the writer
:


--* The reader may use the first read (from the current work arounds for
```

```
--  zero buffered XCHANs: [@ref xchan.zero.a2] or [@ref xchan.zero.b2])
--  as an [@code ALT] guard and commit to the second read as the first
--  part of the response to the guard:
--
--  [@code
--  :
--    -- Pattern 'C'
--    ALT                               -- or PRI ALT
--      ...  other guarded processes
--      STUFF data:
--      from.xchan ? data               -- dummy (ignore data)
--        SEQ
--          from.xchan ? data           -- response must commit to read
--          ...  process the data
--      ...  other guarded processes
--  ]
--
--  However, the reader must beware that the writer will make only best
--  efforts to supply the data for the second read and that this is not
--  guaranteed to be immediate.
--
VAL INT PATTERN.C IS 0:


--* This is a XCHAN process with application-defined capacity.
--
--  If the capacity is set to zero, the implementation uses the [@em simple]
--  behaviour of the logic documented in [@ref xchan.zero.a].
--
--  If the capacity is zero, reading from this XCHAN must follow the
--  [@em read-discard-read-keep] pattern ([@ref xchan.zero.sync.read] or
--  [@ref PATTERN.C]).  If the capacity is non-zero, an [@em occam primitive]
--  read ([@code ?]) must be used.
--
--  For writing, [@ref xchan.async.write], [@ref xchan.blocking.write] or
```

```
--  [@ref PATTERN.A] should be used, regardless of buffering capacity.
--
--  @param max The maximum capacity of this XCHAN ([@code max >= 0]).
--  @param ready This is signalled (with [@code TRUE]) when, and only when,
--    data on the [@ref in] channel can be taken.  This signal [@em must]
--    be taken before data may be sent.
--  @param in Data input
--  @param out Data output
--
PROC xchan.a (VAL INT max, CHAN BOOL ready!, CHAN STUFF in?, out!)
  IF
    max < 0
      STOP                     -- illegal parameter value
    max = 0
      xchan.zero.a2 (ready!, in?, out!)
    TRUE                       -- DEDUCE: max >= 1
      xchan.buffered.a (max, ready!, in?, out!)
:


--* This is a XCHAN process with application-defined capacity.
--
--  If the capacity is set to zero, the implementation uses the [@em better]
--  behaviour of the logic documented in [@ref xchan.zero.b].
--
--  If the capacity is zero, reading from this XCHAN must follow the
--  [@em read-discard-read-keep] pattern ([@ref xchan.zero.sync.read] or
--  [@ref PATTERN.C]).  If the capacity is non-zero, an [@em occam primitive]
--  read ([@code ?]) must be used.
--
--  For writing, [@ref xchan.async.write.b], [@ref xchan.blocking.write.b] or
--  [@ref PATTERN.B] should be used, regardless of buffering capacity.
--  However, if the capacity is non-zero, [@ref xchan.async.write],
--  [@ref xchan.blocking.write] or [@ref PATTERN.A] are marginally more
--  efficient.
```

```
--
--  @param max The maximum capacity of this XCHAN ([@code max >= 0]).
--  @param ready This is signalled (with [@code TRUE]) when, and only when,
--    data on the [@ref in] channel can be taken.  This signal [@em must]
--    be taken before data may be sent.
--  @param in Data input
--  @param out Data output
--
PROC xchan.b (VAL INT max, CHAN BOOL ready!, CHAN STUFF in?, out!)
  IF
    max < 0
      STOP                      -- illegal parameter value
    max = 0
      xchan.zero.b2 (ready!, in?, out!)
    TRUE                        -- DEDUCE: max >= 1
      xchan.buffered.a (max, ready!, in?, out!)
:
```