Message Protocols

Peter Welch (p.h.welch@kent.ac.uk)

Computing Laboratory, University of Kent at Canterbury

Co631 (Concurrency)

Message Protocols

Primitive type protocols ...

Sequential protocols …

A more flexible multiplexer ...

Three monitors ...

Counted array protocols …

A packet multiplexer ...

Variant protocols …

Message Protocols

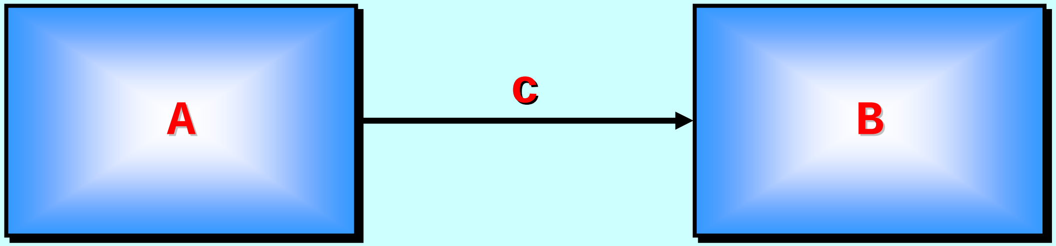
Till now, we have only shown channels carrying one of the ***basic*** **occam-π** types: **`INT`**, **`BYTE`**, **`BOOL`**, **`REAL32`**, …

However, channels may carry ***any*** **occam-π** type: including ***arrays*** and ***records*** (which we have not yet introduced).

occam-π introduces the concept of **`PROTOCOL`**, which enables rich ***message*** structures (containing possibly mixed types) to be declared for individual channels.

The compiler enforces strict adherence – we gain **safety** and **auto-documentation** (of those ***message*** structures).

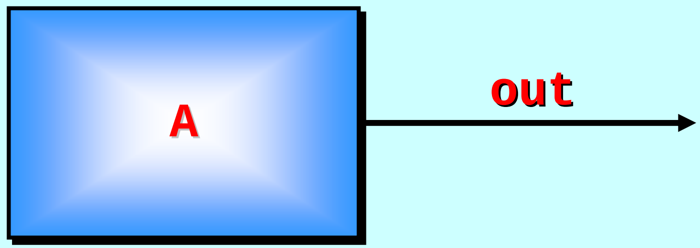
Array Communication




```
CHAN [100]REAL64 c:
PAR
  A (c!)
  B (c?)
```

The channel carries a whole *array* per message …

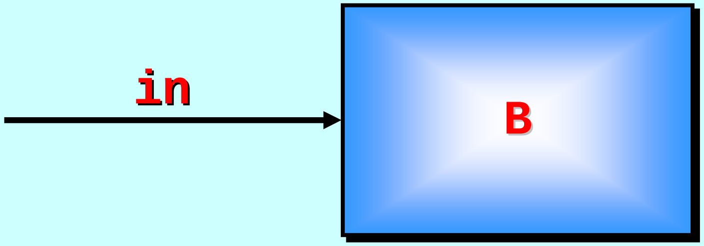
Array Communication




```
PROC A (CHAN [100]REAL64 out!)
  [100]REAL64 data:
  ...  other declarations
  SEQ
    ...  initialise stuff
    WHILE TRUE
      SEQ
        ...  modify data
        out ! data
:
```

the whole array is sent *(copied)*

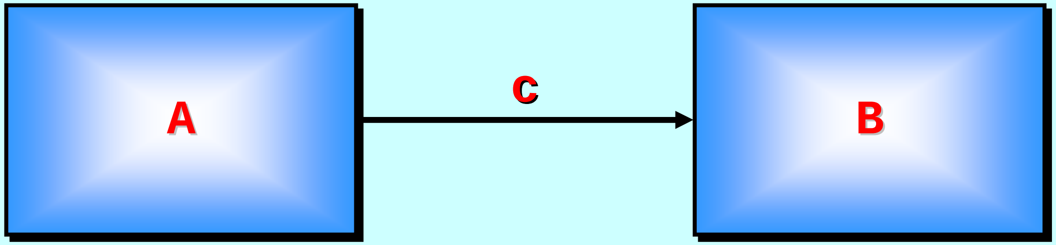
Array Communication




```
PROC B (CHAN [100]REAL64 in?)
  [100]REAL64 data:
  ...  other declarations
  SEQ
    ...  initialise stuff
    WHILE TRUE
      SEQ
        in ? data
        ...  process data
:
```

the whole array is received

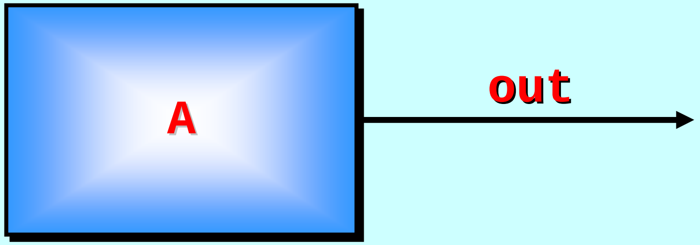
Primitive Communication




```
CHAN REAL64 c:
PAR
  A (c!)
  B (c?)
```

Here, the channel only carries one **REAL64** per message …

Primitive Communication

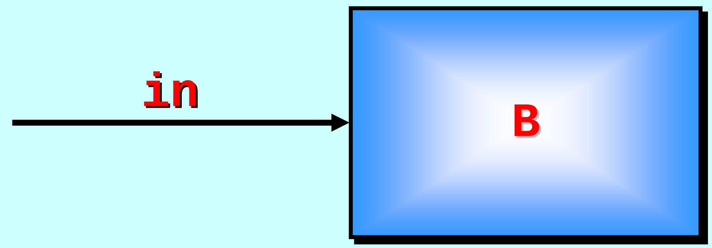



```
PROC A (CHAN [100]REAL64 out!)
  [100]REAL64 data:
  ...  other declarations
  SEQ
    ...  initialise stuff
    WHILE TRUE
      SEQ
        ...  modify data
        SEQ i = 0 FOR SIZE out!
          out ! data[i]
:
```

implies 100 context switches (back into **A**)
plus the loop overhead

100 separate messages

Primitive Communication




```
PROC B (CHAN [100]REAL64 in?)
  [100]REAL64 data:
  ...  other declarations
  SEQ
    ...  initialise stuff
    WHILE TRUE
      SEQ
        SEQ i = 0 FOR SIZE in?
          in ? data[i]
        ...  process data
:
```

implies 100 context switches (back into **B**)
plus the loop overhead

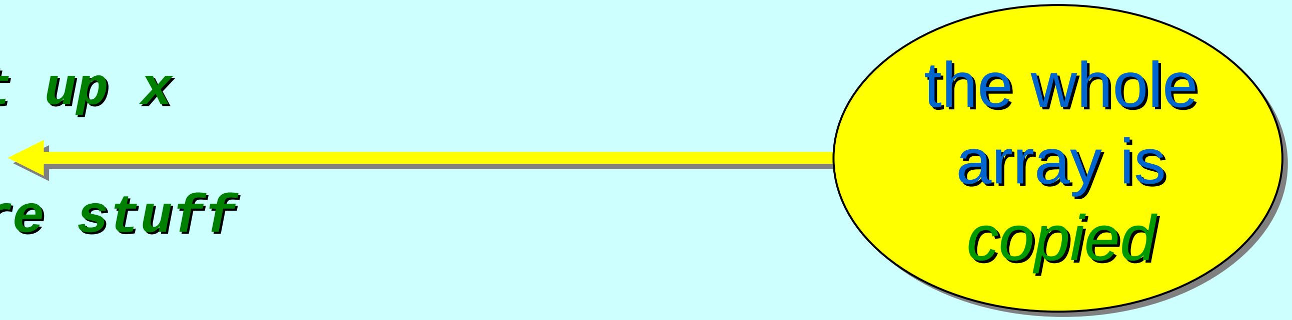
100 separate messages

Array Assignment

Till now, we have only shown assignments between variables having the same ***basic*** **occam-π** type: **INT**, **BYTE**, **REAL32**, …

However, we may assign between variables having the same (but ***any***) **occam-π** type: including ***arrays*** and ***records*** (which we have not yet introduced).

```
[100]REAL64 x, y:
SEQ
  ...  set up x
  y := x
  ...  more stuff
```



Primitive Assignment

Till now, we have only shown assignments between variables having the same ***basic*** **occam-π** type: **INT**, **BYTE**, **REAL32**, …

We *could* assign the elements one at a time …

```
[100]REAL64 x, y:
SEQ
  ...  set up x
  SEQ i = 0 FOR SIZE x
    y[i] := x[i]
  ...  more stuff
```

100 separate assignments

plus the loop overhead ...

Message Protocols

Primitive type protocols ...

Sequential protocols …

A more flexible multiplexer ...

Three monitors ...

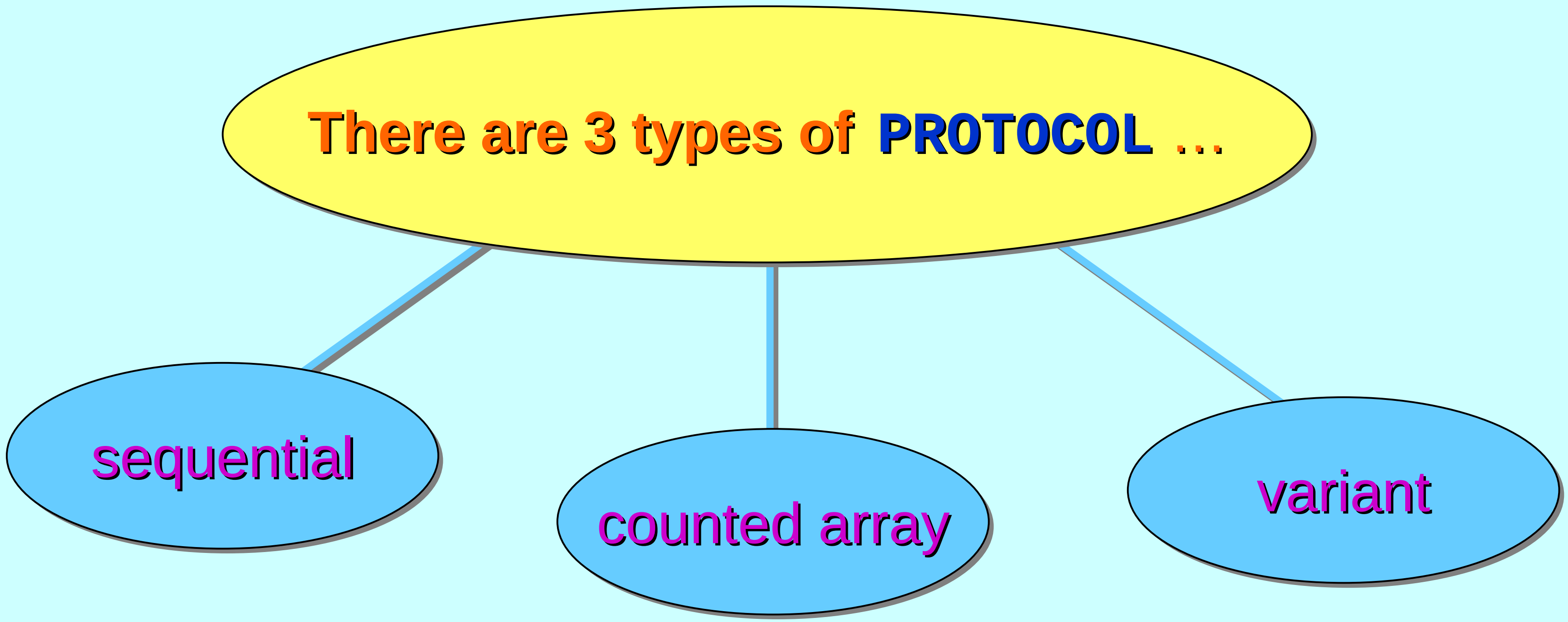
Counted array protocols …

A packet multiplexer ...

Variant protocols …

Message Protocols

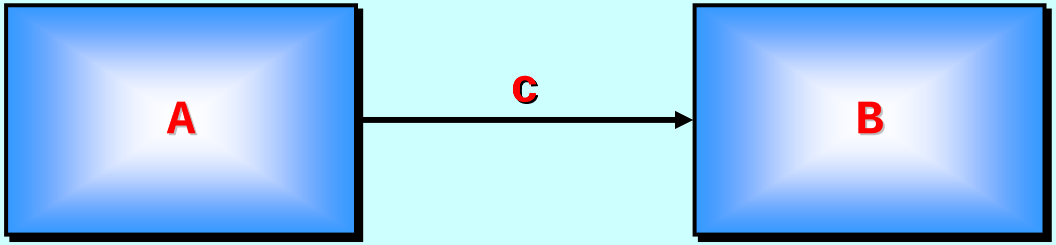
Channels may carry **occam-π** types or **PROTOCOL**s ***…***



An (**occam-π**) **'PROTOCOL'** only describes a *structure* for an *individual message*. It does not describe a *conversation pattern* (e.g. request-reply) of separate messages.

Sequential Protocol

```
PROTOCOL TRIPLE IS INT; BOOL; REAL32:
```

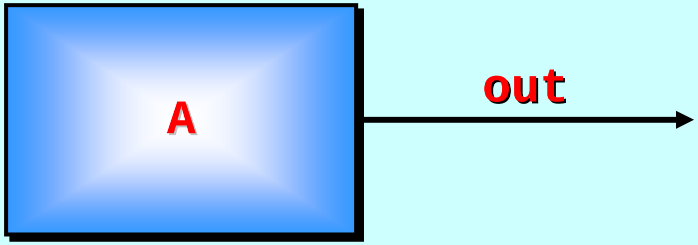



```
CHAN TRIPLE c:
PAR
  A (c!)
  B (c?)
```

The channel carries one **TRIPLE** per message …

Sequential Protocol

```
PROTOCOL TRIPLE IS INT; BOOL; REAL32:
```

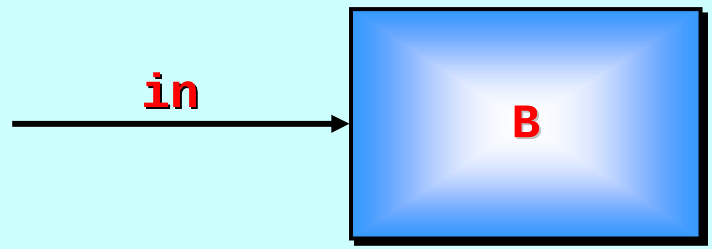



```
PROC A (CHAN TRIPLE out!)
  INITIAL INT i IS 42:
  INITIAL BOOL b IS FALSE:
  INITIAL REAL32 x IS 100.0:
  WHILE TRUE
    SEQ
      out ! i; b; x
      ...  modify i, b and x
:
```

compiler checks message conforms to the declared protocol

Sequential Protocol

```
PROTOCOL TRIPLE IS INT; BOOL; REAL32:
```




```
PROC B (CHAN TRIPLE in?)
  WHILE TRUE
    INT i:
    BOOL b:
    REAL32 x:
    SEQ
      in ? i; b; x
      ...  deal with them
:
```

compiler checks variables conform to the declared protocol

Sequential Protocol

A sequential **PROTOCOL** lists one or more (previously declared) **PROTOCOL**s separated by semi-colons.

```
PROTOCOL TRIPLE IS INT; BOOL; REAL32:
```

```
PROTOCOL DOUBLE.ARRAY IS [100]INT; [42]REAL32:
```

```
PROTOCOL TDA IS TRIPLE; DOUBLE.ARRAY:
```

The last is equivalent to …

```
PROTOCOL TDA IS INT; BOOL; REAL32;
                [100]INT; [42]REAL32:
```

Sequential Protocol

A sequential **PROTOCOL** lists one or more (previously declared) **PROTOCOL**s separated by semi-colons.

```
PROTOCOL TRIPLE IS INT; BOOL; REAL32:
```

```
PROTOCOL DOUBLE.ARRAY IS [100]INT; [42]REAL32:
```

```
PROTOCOL TDA IS TRIPLE; DOUBLE.ARRAY:
```

The sending process outputs a (semi-colon separated) list of ***values*** whose types conform to the **PROTOCOL**.

Sequential Protocol

A sequential **PROTOCOL** lists one or more (previously declared) **PROTOCOL**s separated by semi-colons.

```
PROTOCOL TRIPLE IS INT; BOOL; REAL32:
```

```
PROTOCOL DOUBLE.ARRAY IS [100]INT; [42]REAL32:
```

```
PROTOCOL TDA IS TRIPLE; DOUBLE.ARRAY:
```

The receiving process inputs to a (semi-colon separated) list of ***variables*** whose types conform to the **PROTOCOL**.

Message Protocols

Primitive type protocols ...

Sequential protocols …

A more flexible multiplexer ...

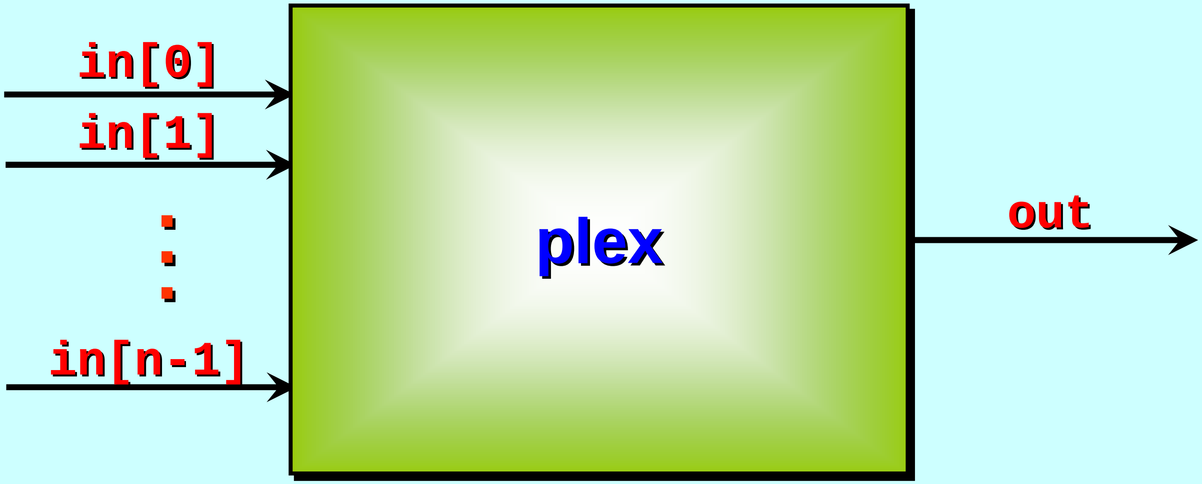
Three monitors ...

Counted array protocols …

A packet multiplexer ...

Variant protocols …

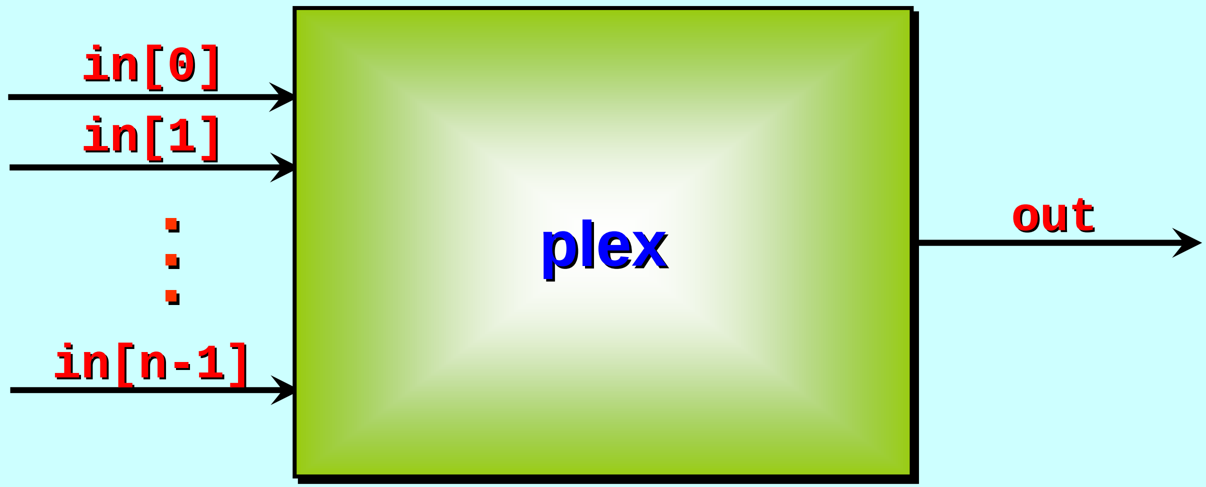
A Simple Multiplexor (seen before)



This process just forwards any message it receives …

… but prefixes the message with the index of the channel on which it had been received …

… which will allow subsequent *de-multiplexing*. ☺ ☺ ☺

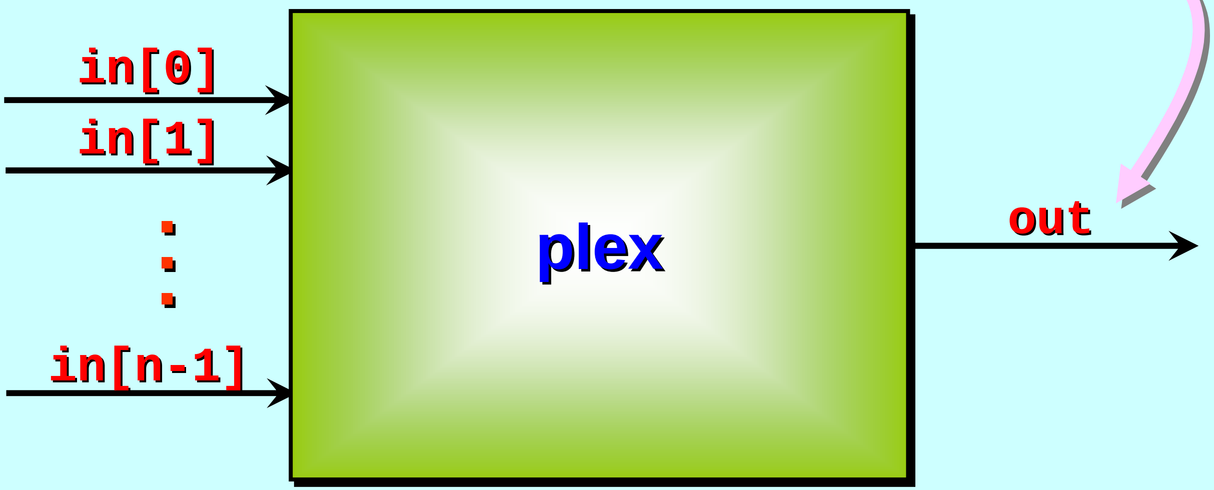



```
PROC plex ([]CHAN INT in?, CHAN INT out!)
  WHILE TRUE
    ALT i = 0 FOR SIZE in?
      INT x:
      in[i] ? x
        SEQ
          out ! i
          out ! x
:
```

the array size

This guarded process gets replicated ...

PROTOCOL INDEX.INT IS INT; INT:




```
PROC plex ([]CHAN INT in?, CHAN INDEX.INT out!)
  WHILE TRUE
    ALT i = 0 FOR SIZE in?
      INT x:
      in[i] ? x
        out ! i; x
:
```

the array size

This guarded process gets replicated ...

A Matching De-Multiplexor (*seen*)

This process recovers input messages to their correct output channels … and assumes each message is prefixed by the correct target channel index …

Each message must be a **`<index, data>`** pair, generated by a **`plex`** process (with the same number of inputs as this has outputs).


```
PROC de.plex (CHAN INT in?, []CHAN INT out!)
  WHILE TRUE
    INT i, x:
    SEQ
      in ? i
      in ? x
      out[i] ! x
:
```

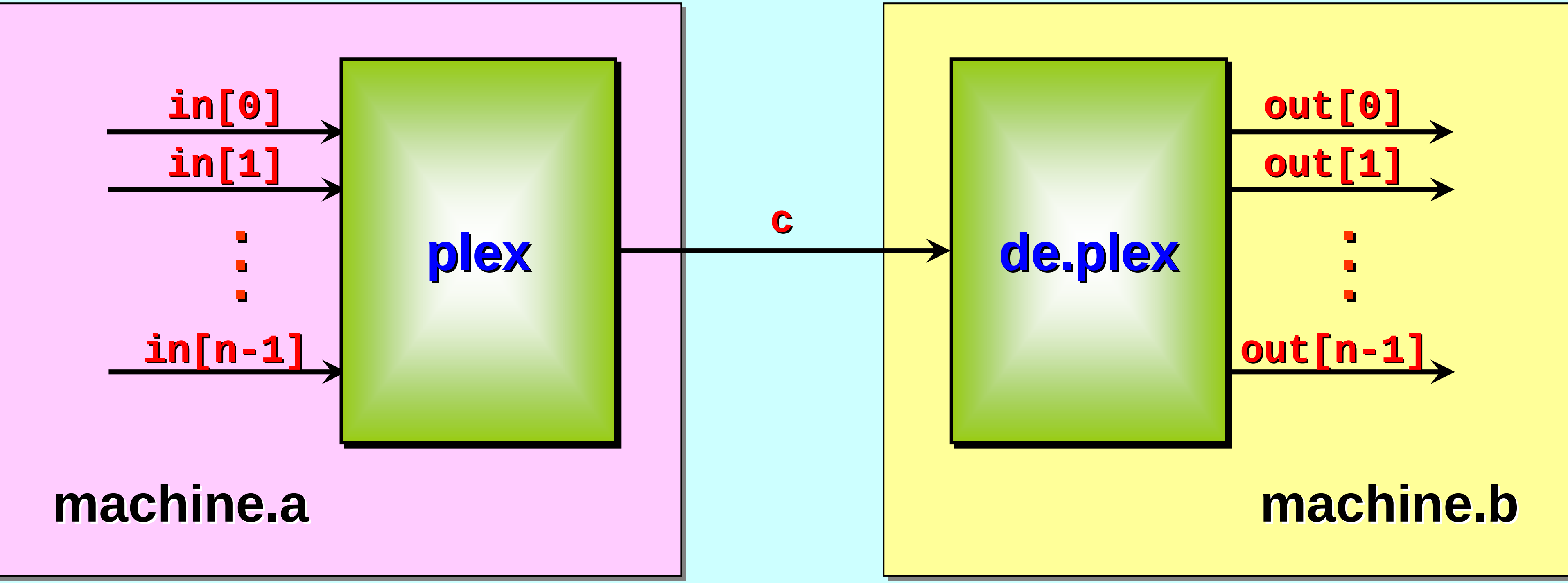
This must be a legal index of the **out** array!

```
PROTOCOL INDEX.INT IS INT; INT:
```


```
PROC de.plex (CHAN INDEX.INT in?, []CHAN INT out!)
  WHILE TRUE
    INT i, x:
    SEQ
      in ? i; x
      out[i] ! x
:
```

This must be a legal index of the **out** array!

Multiplexor Application (Example)



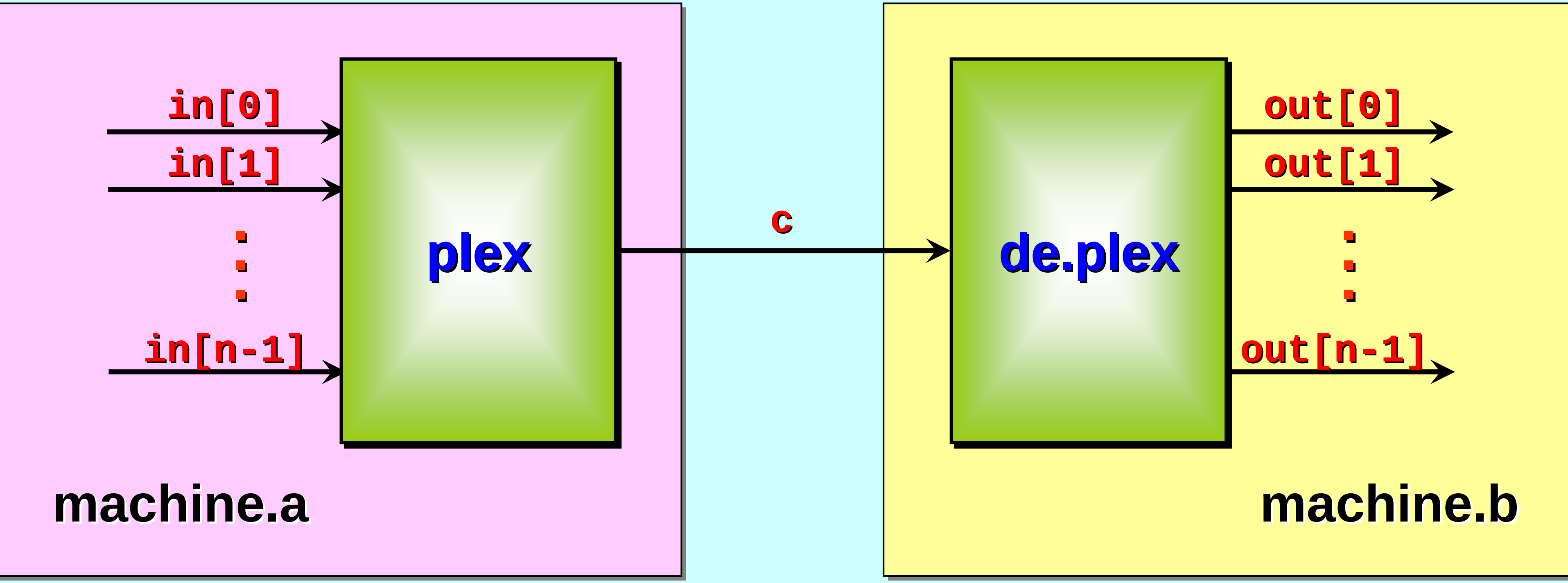
Message structures should be *documented* somewhere!

```
PROTOCOL INDEX.INT IS INT; INT:
```

```
CHAN INDEX.INT c:
```

done
☺ ☺ ☺

Multiplexor Application (Example)

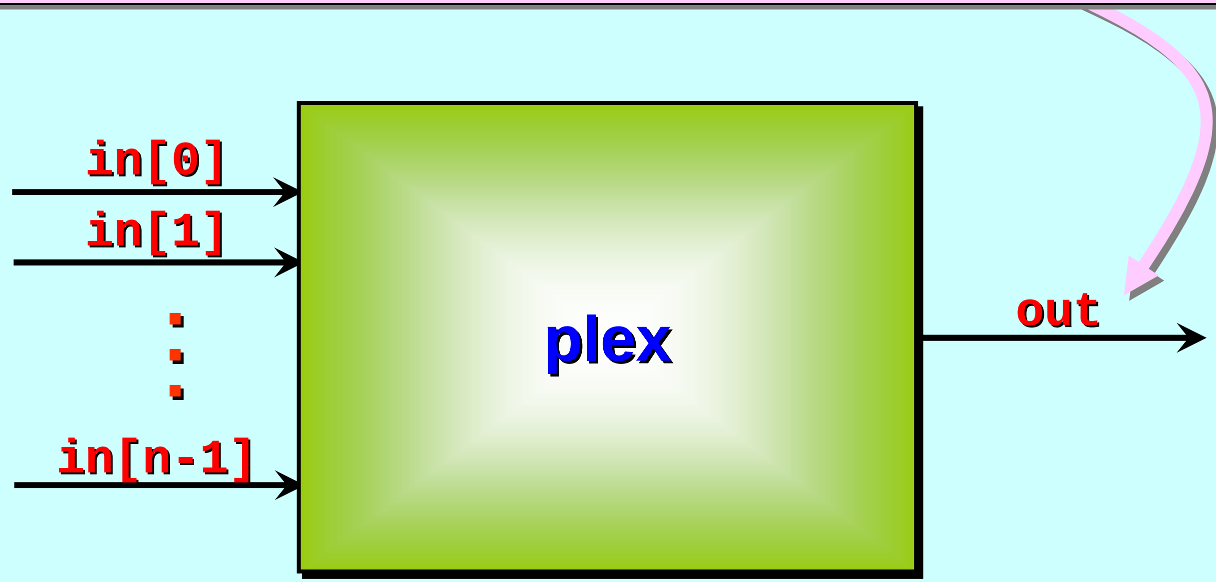


However, suppose that the *messages* to be multiplexed were type **REAL64** …

Now, messages on **c** have form: **INT; REAL64**

How do we type the *multiplexed* channel: **CHAN *???* c:**

```
PROTOCOL INDEX.INT IS INT; INT:
```

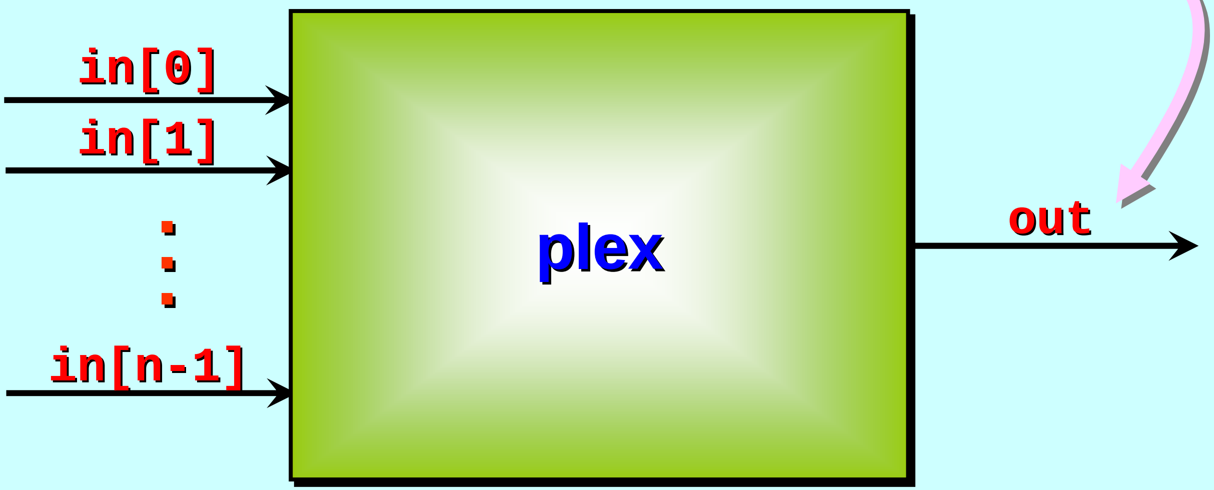



```
PROC plex ([]CHAN INT in?, CHAN INDEX.INT out!)
  WHILE TRUE
    ALT i = 0 FOR SIZE in?
      INT x:
      in[i] ? x
        out ! i; x
:
```

the array size

This guarded process gets replicated ...

PROTOCOL INDEX.REAL64 IS INT; REAL64:

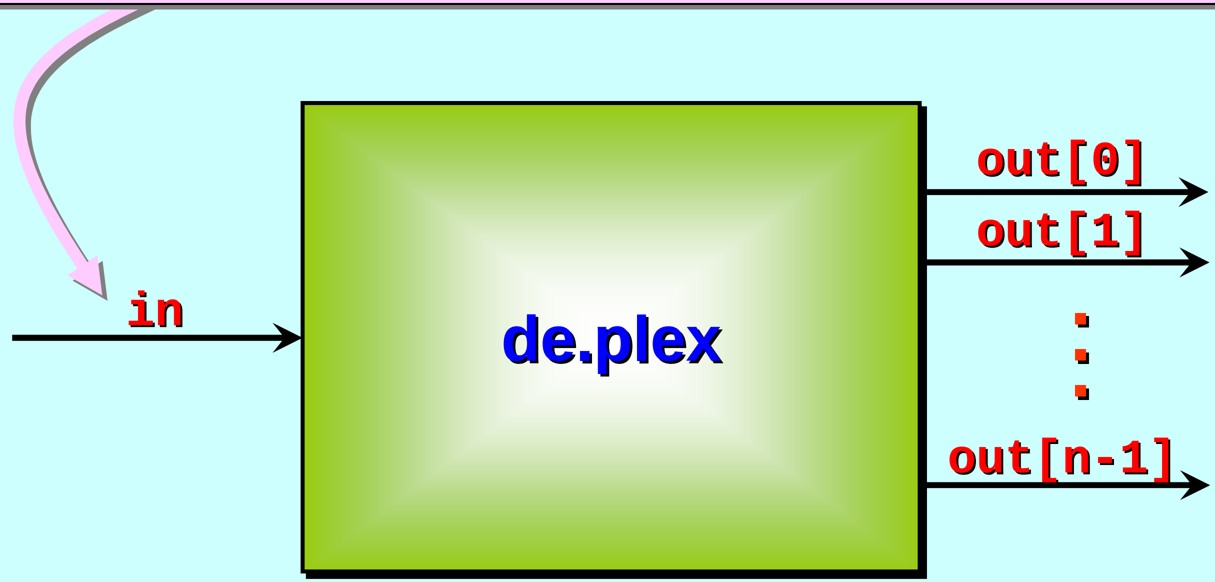



```
PROC plex ([]CHAN REAL64 in?, CHAN INDEX.REAL64 out!)
  WHILE TRUE
    ALT i = 0 FOR SIZE in?
      REAL64 x:
      in[i] ? x
        out ! i; x
:
```

the array size

This guarded process gets replicated ...

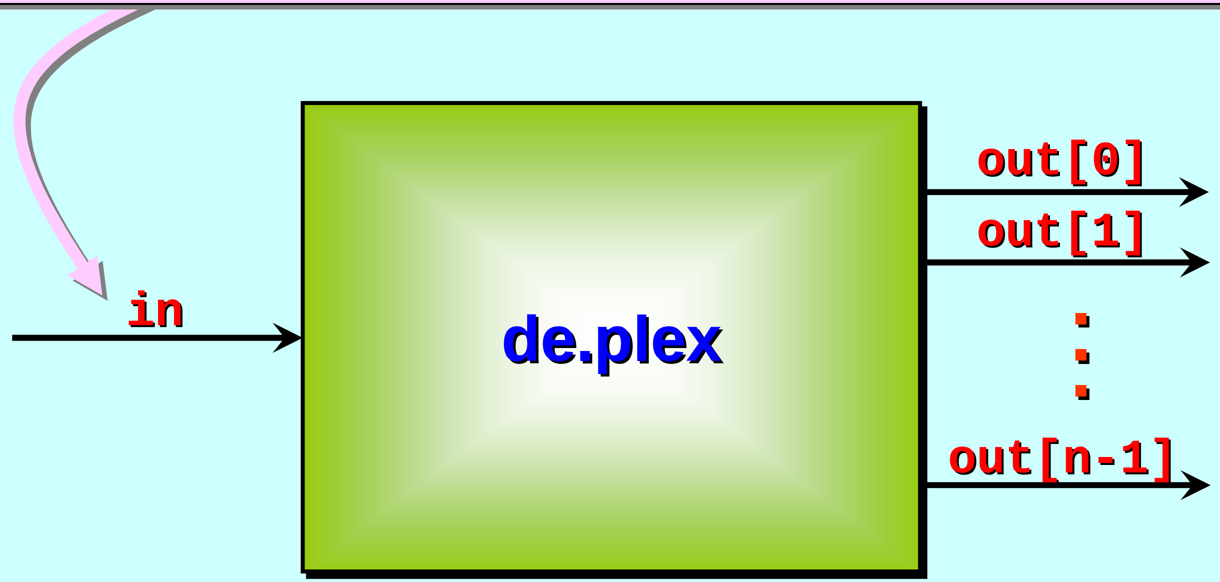
```
PROTOCOL INDEX.INT IS INT; INT:
```




```
PROC de.plex (CHAN INDEX.INT in?, []CHAN INT out!)
  WHILE TRUE
    INT i, x:
    SEQ
      in ? i; x
      out[i] ! x
:
```

This must be a legal index of the **out** array!

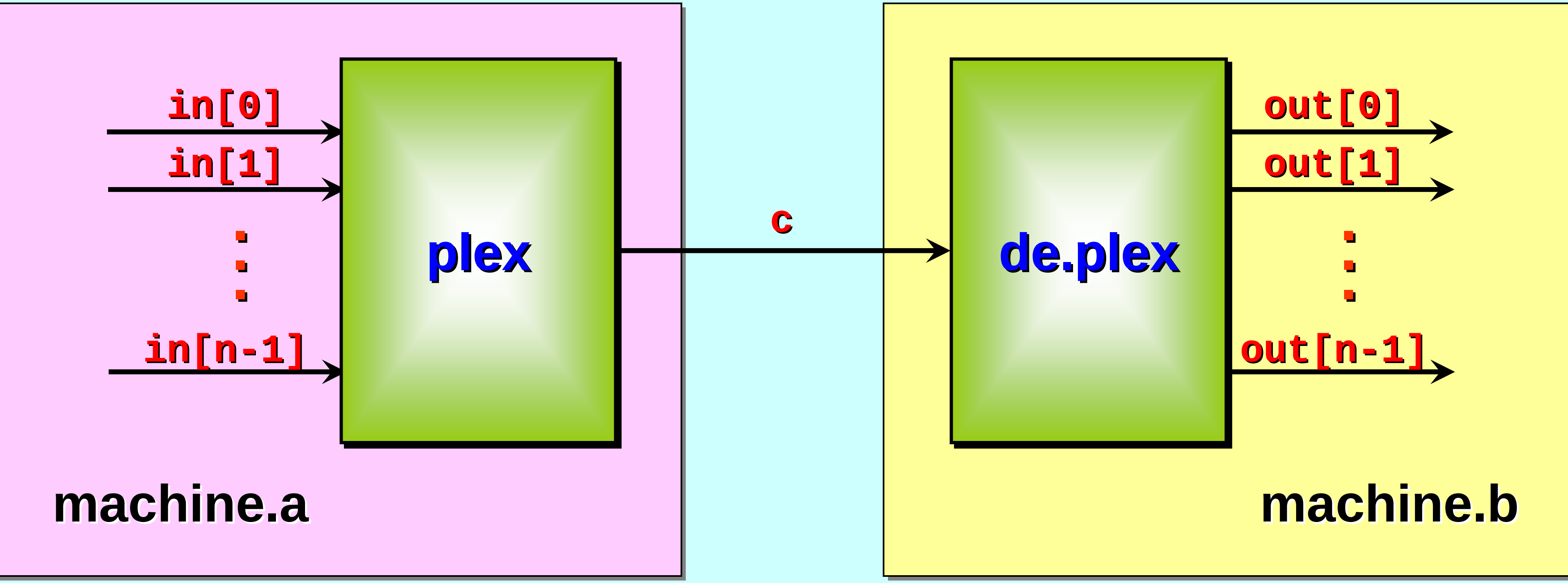
PROTOCOL INDEX.REAL64 IS INT; REAL64:




```
PROC de.plex (CHAN INDEX.REAL64 in?, []CHAN REAL64 out!)
  WHILE TRUE
    INT i:
    REAL64 x:
    SEQ
      in ? i; x
      out[i] ! x
:
```

This must be a legal index of the **out** array!

Multiplexor Application (Example)



Message structures should be *documented* somewhere!

```
PROTOCOL INDEX.REAL64 IS INT; REAL64:
```

```
CHAN INDEX.REAL64 c:
```

done ☺ ☺ ☺

Message Protocols

Primitive type protocols ...

Sequential protocols …

A more flexible multiplexer ...

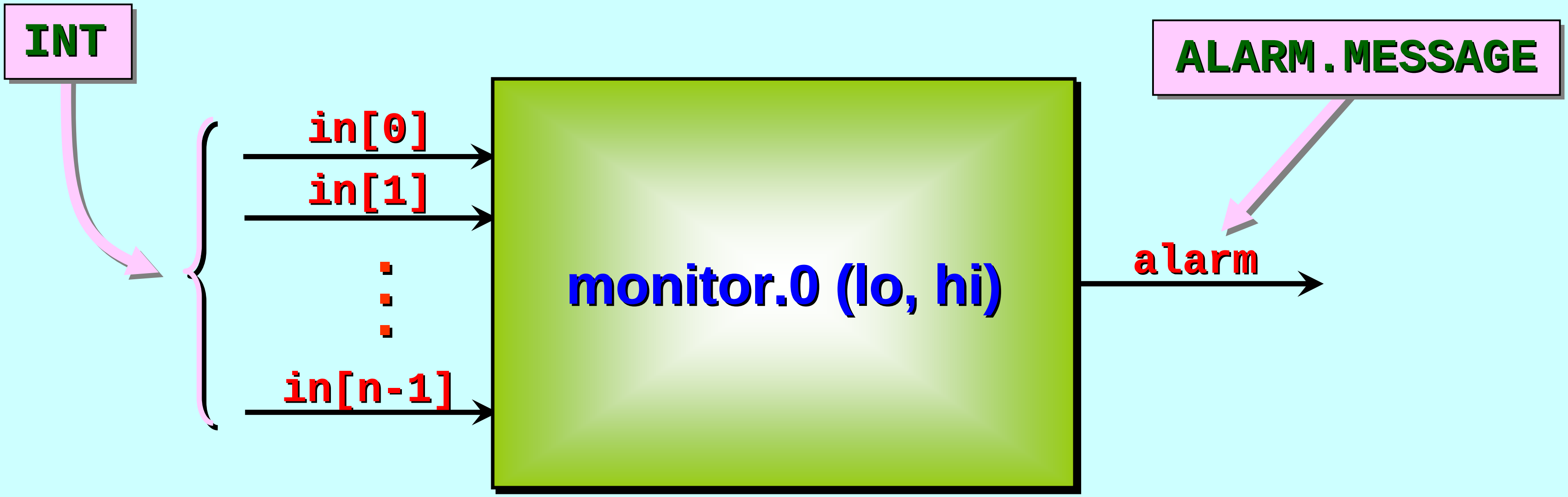
Three monitors ...

Counted array protocols …

A packet multiplexer ...

Variant protocols …

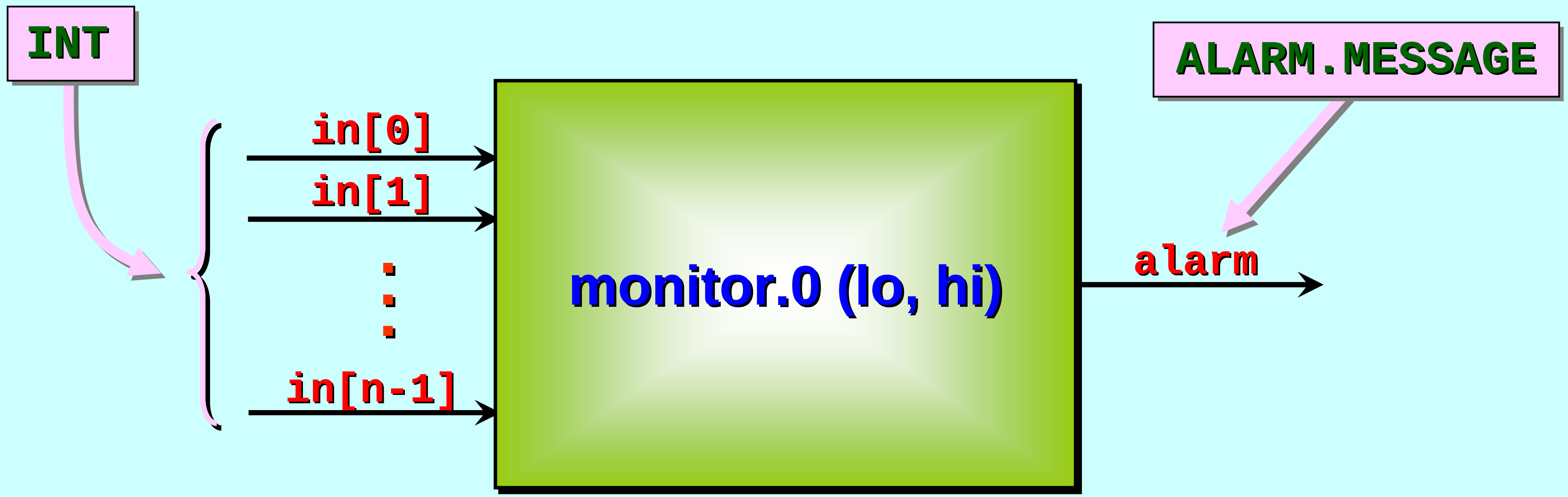
A Simple Data Monitor



The input channels deliver raw sensor data (such as *temperature/pressure* measurements from a machine). The rate of supply of this data is *irregular*.

This process *monitors* that data, raising an **`alarm`** should any lie outside the range **`lo..hi`** (defined by its parameters).

A Simple Data Monitor

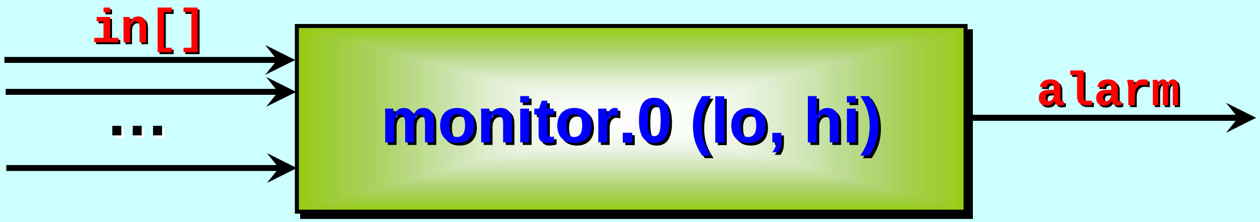



```
PROTOCOL ALARM.MESSAGE IS INT; INT:
```

offending channel index

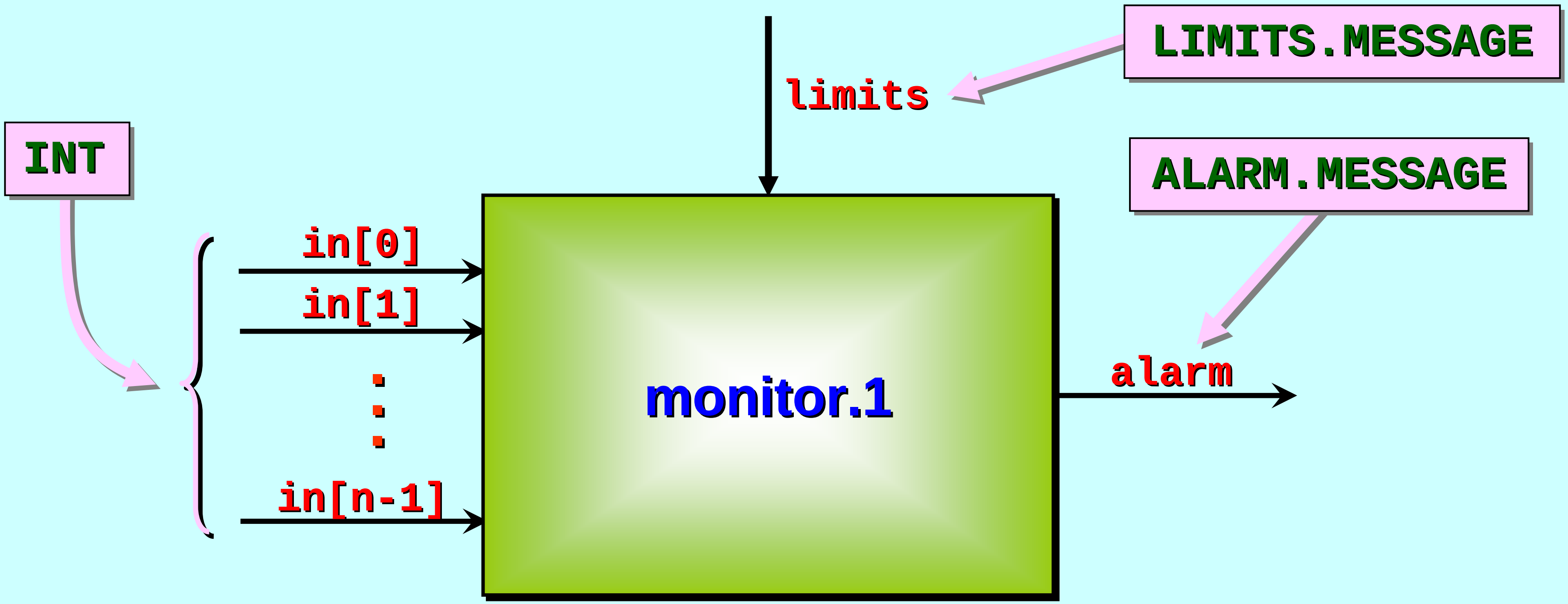
offending data

A Simple Data Monitor



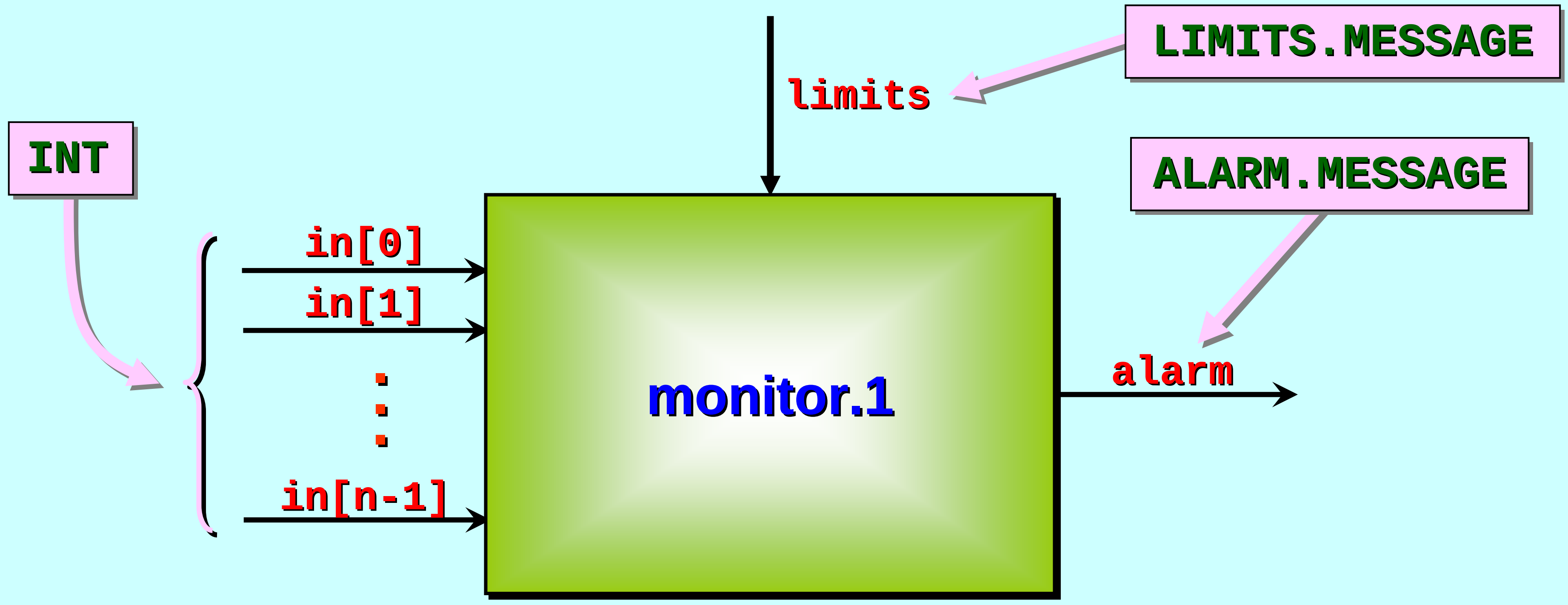

```
PROC monitor.0 (VAL INT lo, hi, []CHAN INT in?,
                CHAN ALARM.MESSAGE alarm!)
  WHILE TRUE
    ALT i = 0 FOR SIZE in?
      INT x:
      in[i] ? x
        IF
          (x < lo) OR (x > hi)
            alarm ! i; x
          TRUE
            SKIP
:
```

A Better Data Monitor



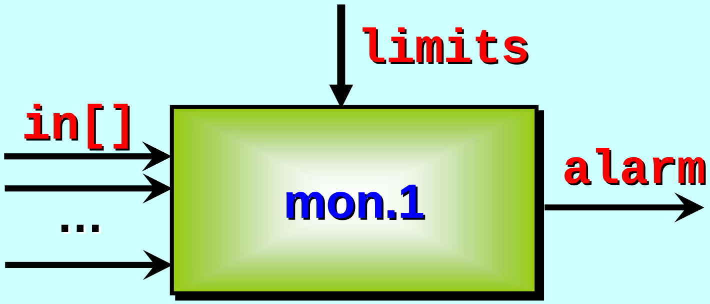
This version allows the *'safe'* limits of the monitored range to be changed at run-time. It also refuses to start monitoring until those limits have been set.

A Better Data Monitor



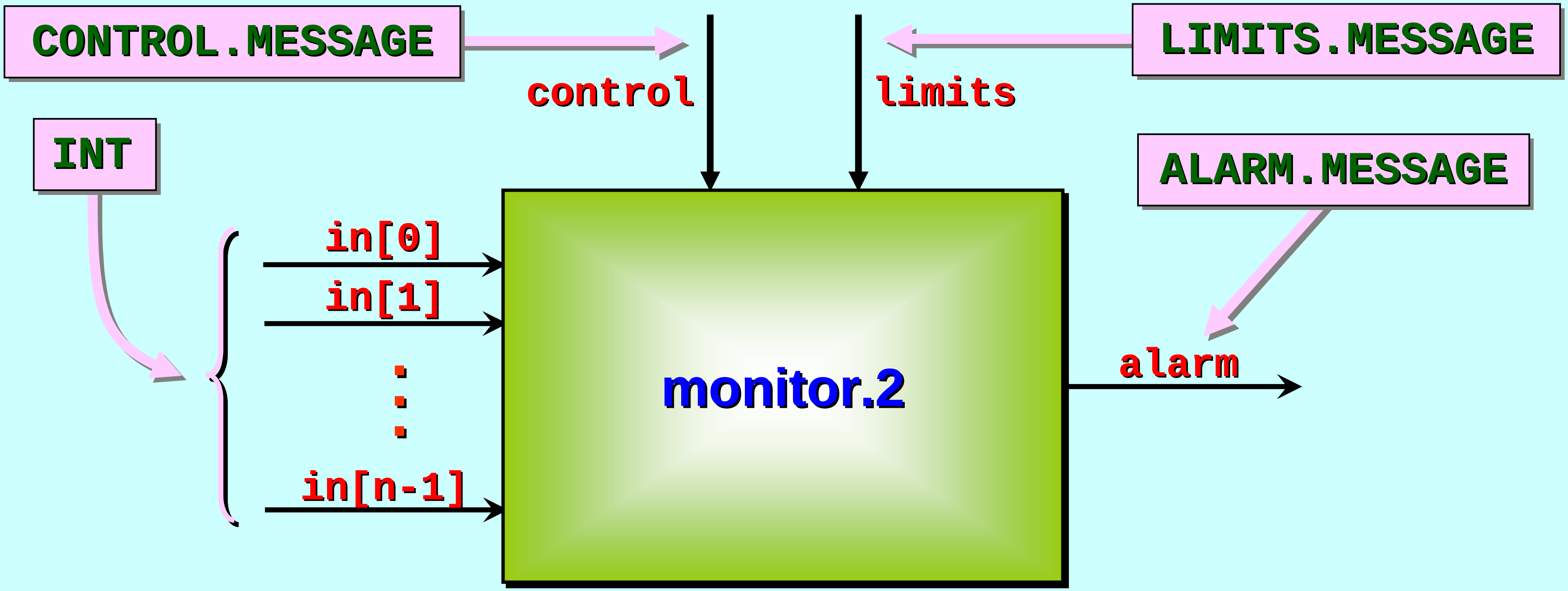

```
PROTOCOL LIMITS.MESSAGE IS INT; INT:
```

```
PROC monitor.1 ([]CHAN INT in?,
                CHAN LIMITS.MESSAGE limits?,
                CHAN ALARM.MESSAGE alarm!)
  INT lo, hi:
  SEQ
    limits ? lo; hi
    WHILE TRUE
      PRI ALT
        limits ? lo; hi
          SKIP
        ALT i = 0 FOR SIZE in?
          INT x:
          in[i] ? x
            IF
              (x < lo) OR (x > hi)
                alarm ! i; x
              TRUE
                SKIP
:
```



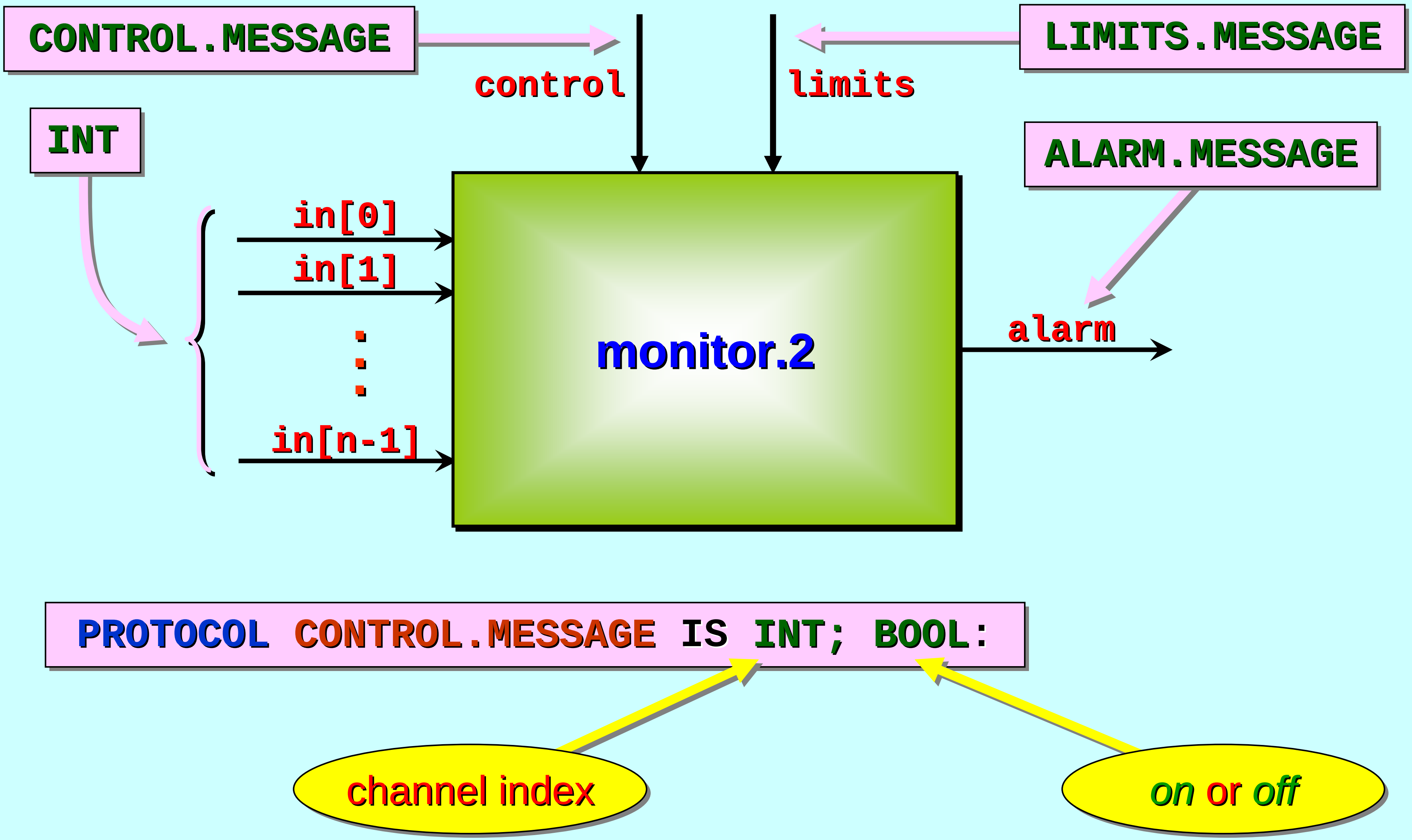
This **ALT** is nested ...

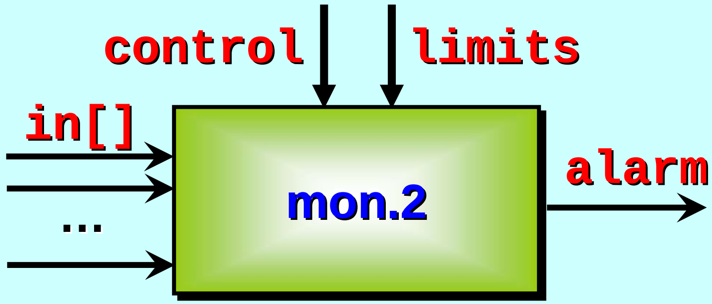
An Even Better Data Monitor



This version allows *switching off* listening to some (or all) of the data channels at run-time. Initially, it is set to listen to *all* channels.

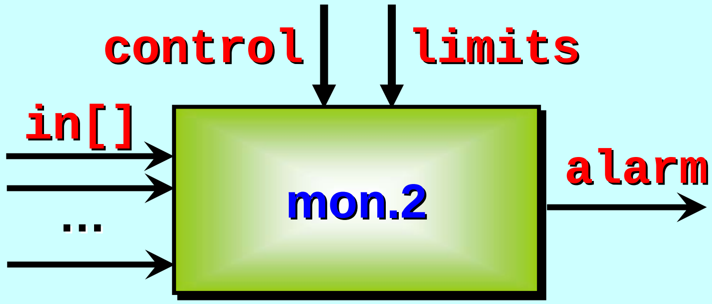
An Even Better Data Monitor





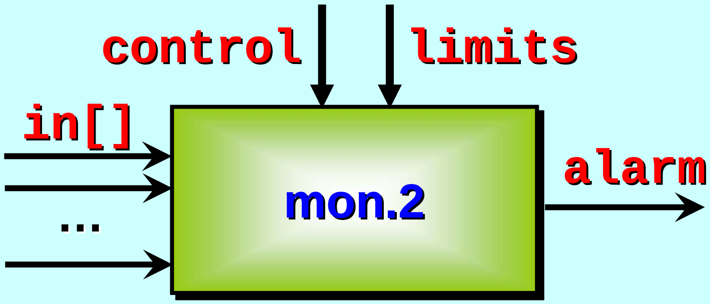

```
PROC monitor.2 ([]CHAN INT in?,
                CHAN CONTROL.MESSAGE control?,
                CHAN LIMITS.MESSAGE limits?,
                CHAN ALARM.MESSAGE alarm!,
                []BOOL ok)
:
```

User supplies an array of **BOOL** *flags* for this process to use …



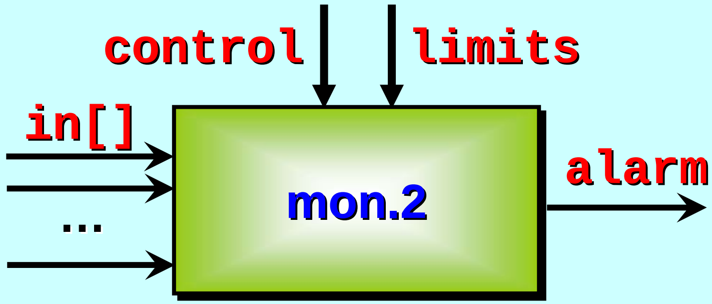

```
PROC monitor.2 ([]CHAN INT in?,
                CHAN CONTROL.MESSAGE control?,
                CHAN LIMITS.MESSAGE limits?,
                CHAN ALARM.MESSAGE alarm!,
                []BOOL ok)
  -- assume: (SIZE in?) = (SIZE ok)
  INT lo, hi:
  SEQ
    ...  initialise
    ...  main cycle
:
```

We need one *flag* for each monitored channel …




```
--{{{  initialise
SEQ
  limits ? lo; hi
  SEQ i = 0 FOR SIZE ok
    ok[i] := TRUE
--}}}
```

Initially, listen to *all* inputs …




```
--{{{  main cycle
WHILE TRUE
  PRI ALT
    limits ? lo; hi
      SKIP
    INT line:
    control ? line; ok[line]
      SKIP
    ALT i = 0 FOR SIZE in?
      INT x:
      ok[i] & in[i] ? x
        IF
          (x < lo) OR (x > hi)
            alarm ! i; x
          TRUE
            SKIP
--}}}
```

This guard is *pre-conditioned* …

Message Protocols

Primitive type protocols ...

Sequential protocols …

A more flexible multiplexer ...

Three monitors ...

Counted array protocols …

A packet multiplexer ...

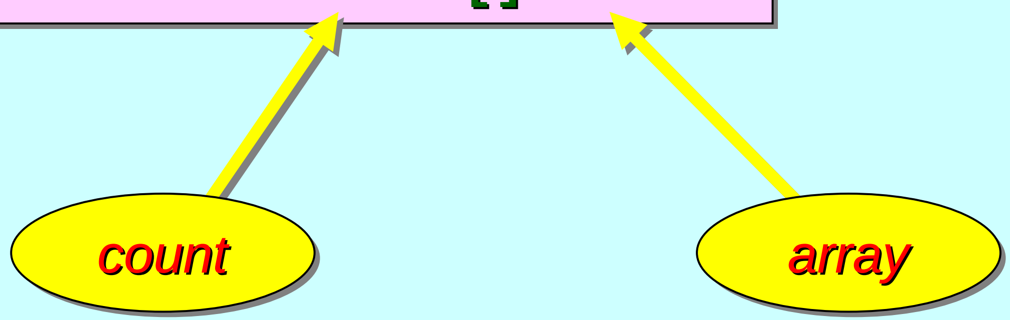
Variant protocols …

Counted Array Protocol

Previous message structures (**PROTOCOL**s) have always had a known and fixed size.

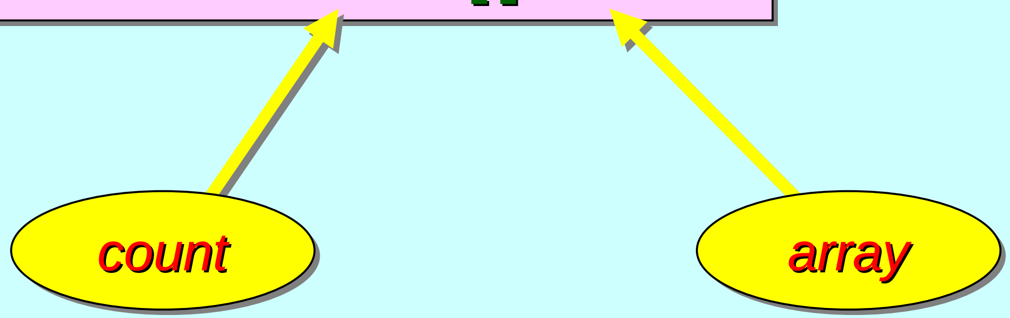
We now describe messages whose components all have the same type (in fact, they are an *array*) but whose *size* is part of the message …

```
PROTOCOL PACKET IS INT::[]REAL64:
```



Counted Array Protocol

```
PROTOCOL PACKET IS INT::[]REAL64:
```



The sending process outputs a ***count*** (which must be >= 0) followed by (::) an ***array*** (whose size must be >= ***count***).

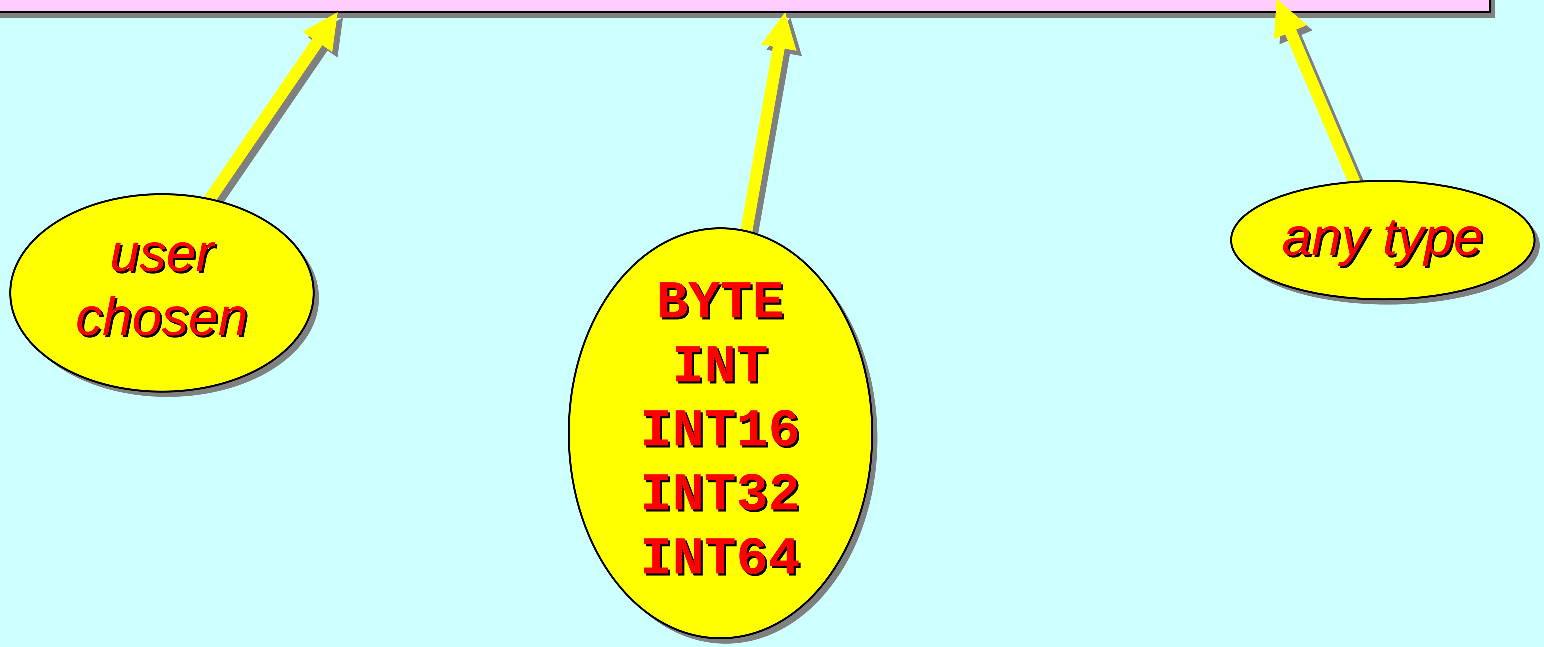
The types of the ***count*** and the ***array*** must conform to the **PROTOCOL**.

Only the first ***count*** elements of the ***array*** are sent (***copied***).

Counted Array Protocol

In general …

```
PROTOCOL <name> IS <discrete-type>::[]<type>:
```



The *count* value must be *non-negative* and only the first *count* elements of the *array* are communicated.

Message Protocols

Primitive type protocols ...

Sequential protocols …

A more flexible multiplexer ...

Three monitors ...

Counted array protocols …

A packet multiplexer ...

Variant protocols …

Counted Array Protocol (*Example*)

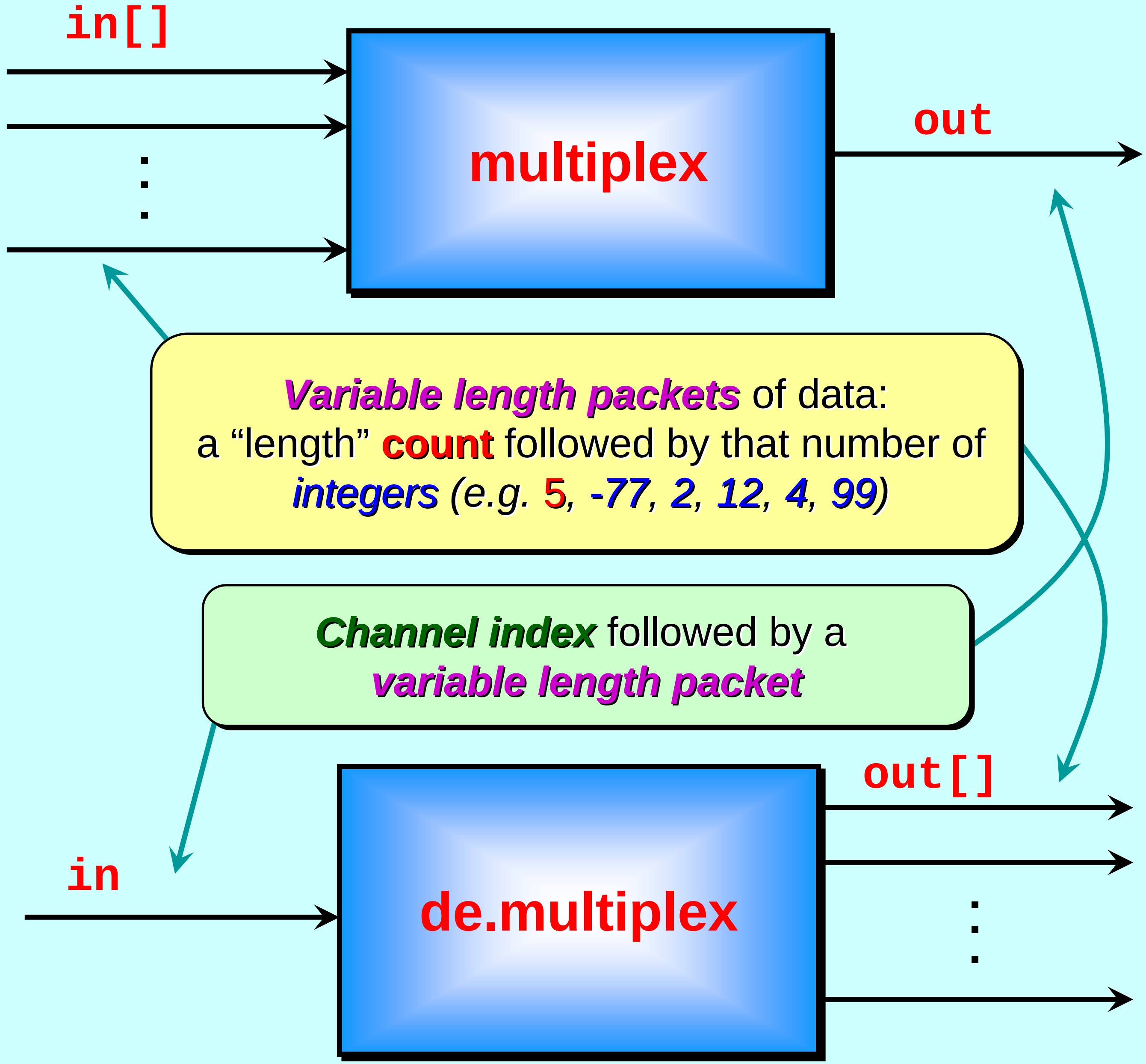
This is another multiplexor / de-multiplexor example.

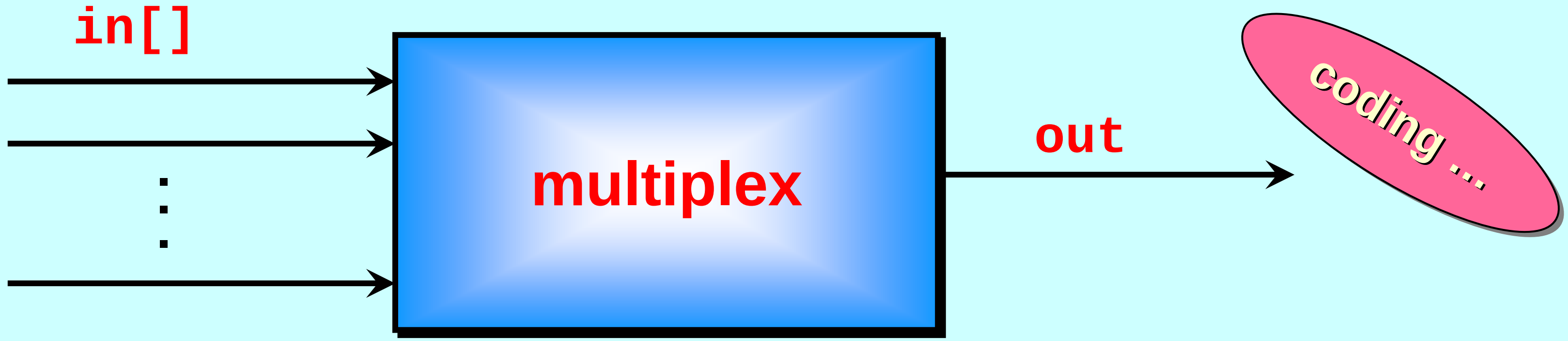
This time the messages being multiplexed are *'packets'* of data, whose size is given at run-time.

Initially, the elements within these *'packets'* are all **`INT`**s.

And we will program them at a low-level, using **`CHAN INT`**s.

Afterwards, we will program them using a *counted array*.


in[]
multiplex
out
Variable length packets of data:
a “length” count followed by that number of
integers (e.g. 5, -77, 2, 12, 4, 99)
Channel index followed by a
variable length packet
out[]
in
de.multiplex




```
PROC multiplex ([]CHAN INT in?, CHAN INT out!)
  WHILE TRUE
    ALT i = 0 FOR SIZE in?
      INT length:
      in[i] ? length
        SEQ
          out ! i
          out ! length
          SEQ j = 0 FOR length
            INT x:
            SEQ
              in[i] ? x
              out ! x
:
```

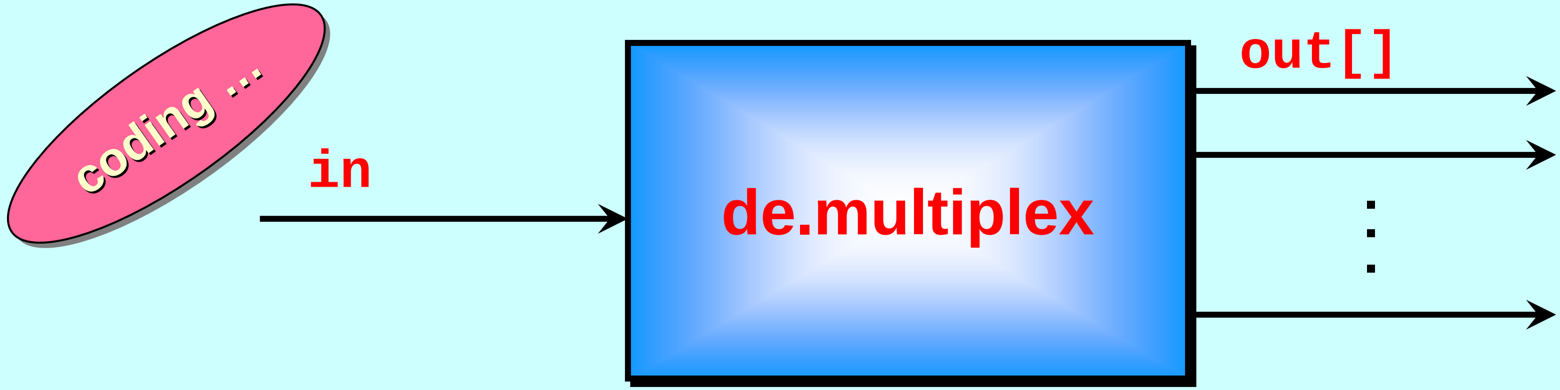
replicated **ALT**

guard

index

count

rest of 'packet'




```
PROC de.multiplex (CHAN INT in?, []CHAN INT out!)
  WHILE TRUE
    INT i, length:
    SEQ
      in ? i
      in ? length
      out[i] ! length
      SEQ j = 0 FOR length
        INT x:
        SEQ
          in ? x
          out[i] ! x
:
```

index

count

rest of 'packet'

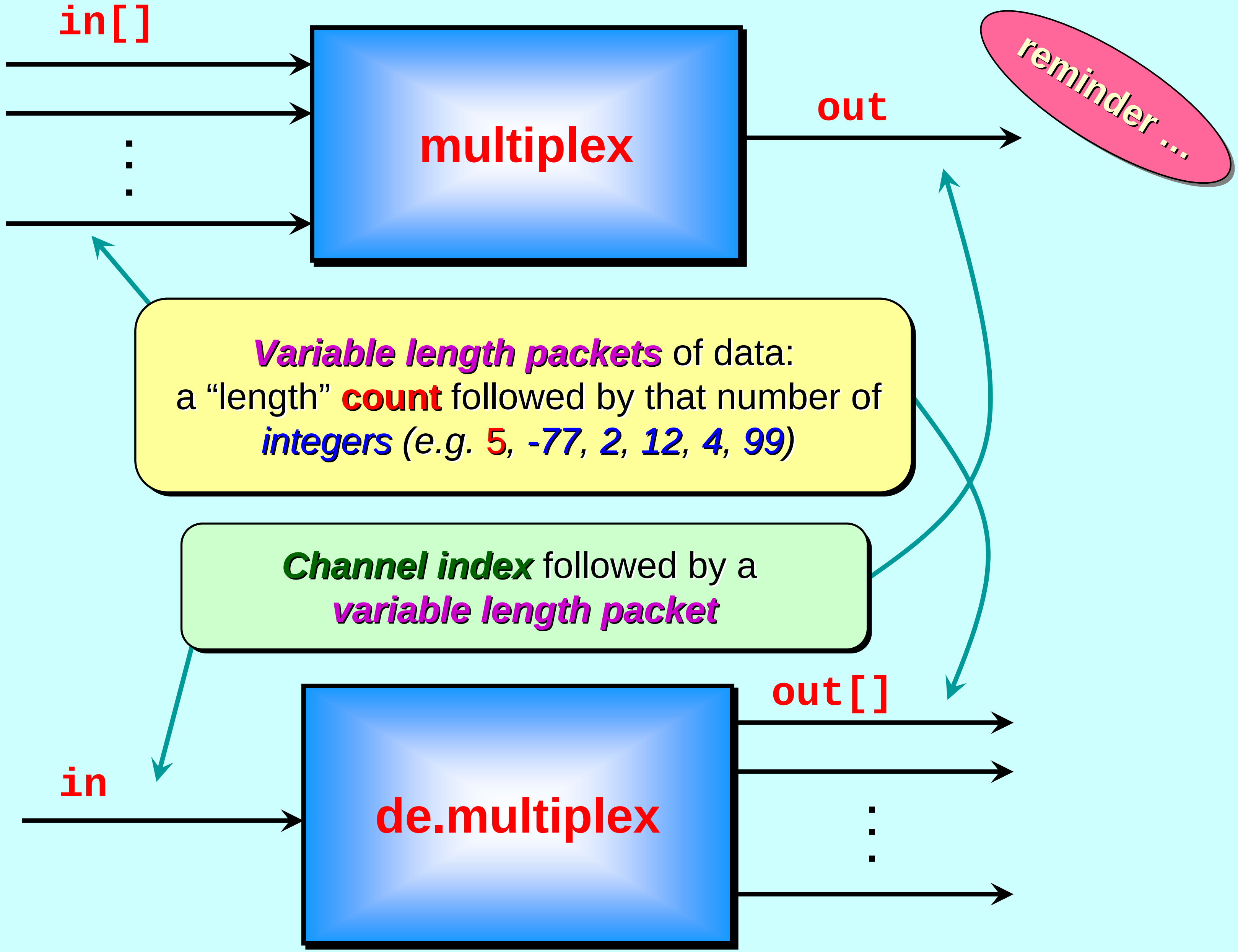
Counted Array Protocol (*Example*)

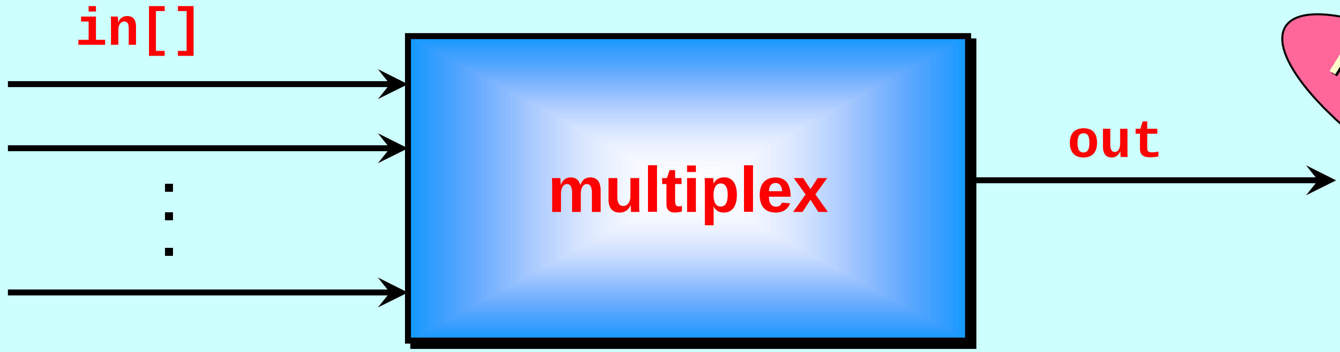
```
PROTOCOL PACKET IS INT::[]REAL64:
```

```
PROTOCOL INDEX.PACKET IS INT; PACKET:
```

The last is equivalent to …

```
PROTOCOL INDEX.PACKET IS INT; INT::[]REAL64:
```


in[]
multiplex
out
reminder …
Variable length packets of data:
a “length” count followed by that number of
integers (e.g. 5, -77, 2, 12, 4, 99)
Channel index followed by a
variable length packet
out[]
in
de.multiplex



reminder ...

```
PROC multiplex ([]CHAN INT in?, CHAN INT out!)
  WHILE TRUE
    ALT i = 0 FOR SIZE in?
      INT length:
      in[i] ? length
        SEQ
          out ! i
          out ! length
          SEQ j = 0 FOR length
            INT x:
            SEQ
              in[i] ? x
              out ! x
:
```

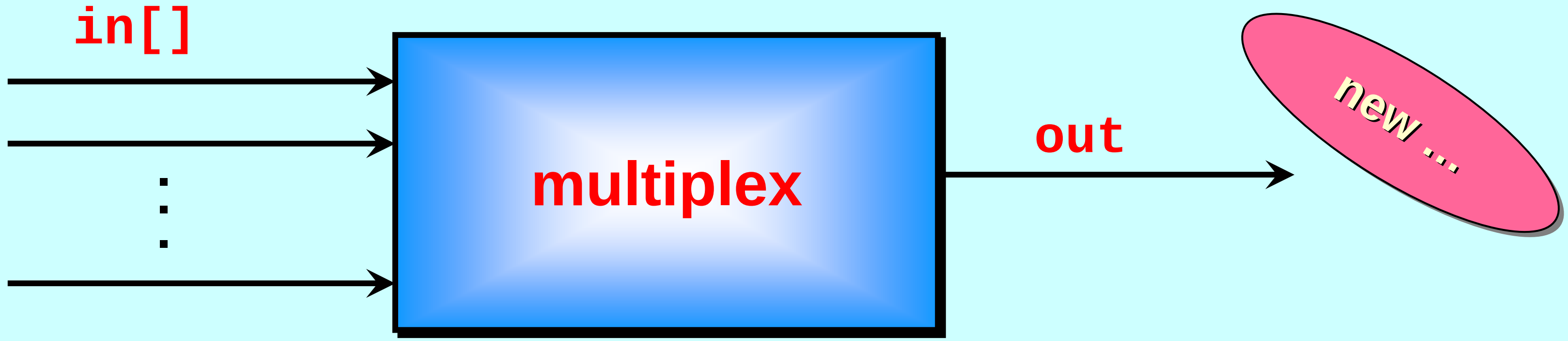
replicated **ALT**

guard

index

count

rest of ‘packet’



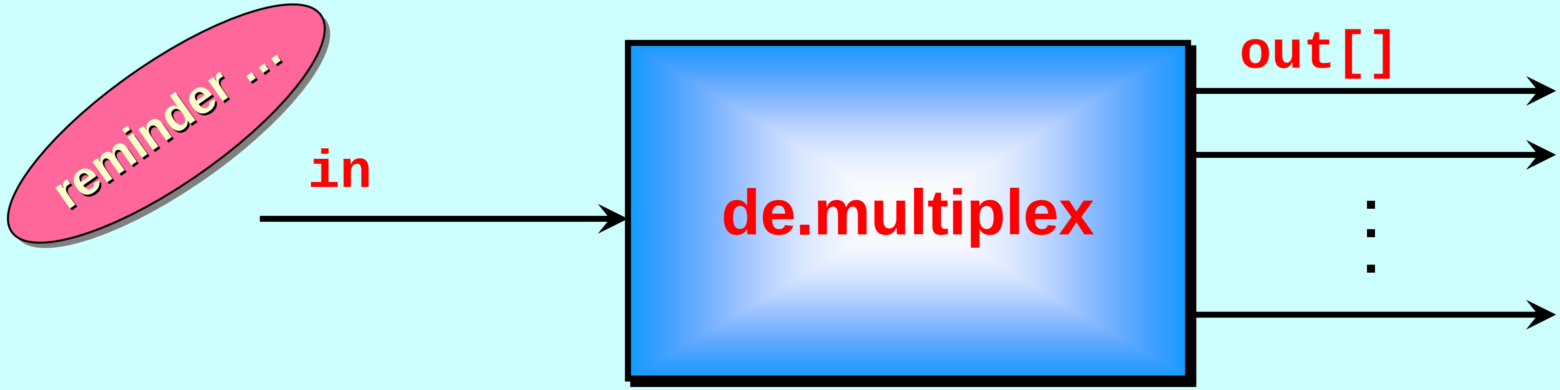

```
PROC multiplex ([]REAL64 buffer,
                []CHAN PACKET in?,
                CHAN INDEX.PACKET out!)
  WHILE TRUE
    ALT i = 0 FOR SIZE in?
      INT length:
      in[i] ? length::buffer
        out ! i; length::buffer
:
```

replicated **ALT**

guard

guarded process

User supplies **multiplex** with a **buffer** sufficiently large for all messages that will be passed through this component.

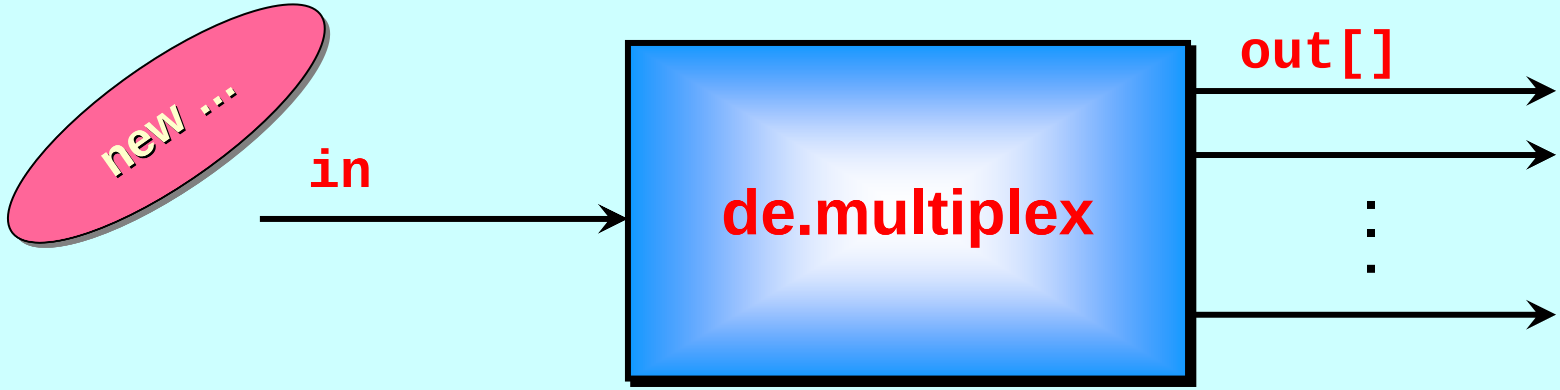



```
PROC de.multiplex (CHAN INT in?, []CHAN INT out!)
  WHILE TRUE
    INT i, length:
    SEQ
      in ? i
      in ? length
      out[i] ! length
      SEQ j = 0 FOR length
        INT x:
        SEQ
          in ? x
          out[i] ! x
:
```

index

count

rest of ‘packet’




```
PROC de.multiplex ([]REAL64 buffer,
                   CHAN INDEX.PACKET in?,
                   []CHAN PACKET out!)
  WHILE TRUE
    INT i, length:
    SEQ
      in ? i; length::buffer
      out[i] ! length::buffer
:
```

- index
- count
- rest of ‘packet’

User supplies **de.multiplex** with a **buffer** sufficiently large for all messages that will be passed through this component.

Counted Array Protocol

Gives us a *higher level* expression for this communication structure: a *count* followed by (an array of) *count* items.

Gives us *shorter* and *easier to write and understand* code.

Allows array components of *any* **occam-π** type (not just **INT**).

Yields *much faster code* (than directly programming the loops).

But *requires buffer space* to hold a complete message – whereas the low-level loop code *worm-holed* the message through, needing just two **INT**s (one for the **length** and one for each **INT** *element* of the message).

Message Protocols

Primitive type protocols ...

Sequential protocols …

A more flexible multiplexer ...

Three monitors ...

Counted array protocols …

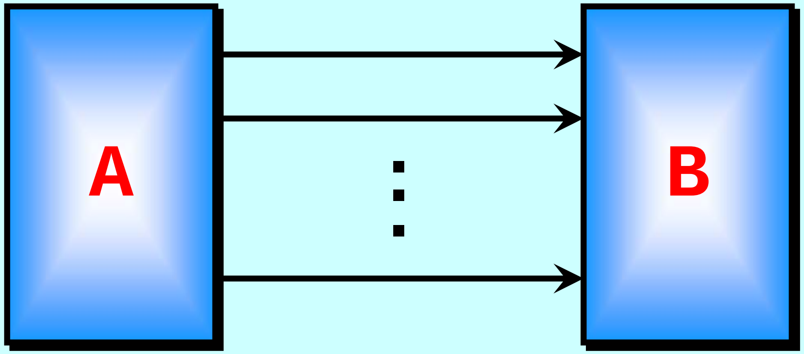
A packet multiplexer ...

Variant protocols …

Variant Protocol

Often we need to send *different* kinds (*i.e. protocols*) of message between a pair of processes.

One way would be to connect them with a set of different channels – each carrying *one* of the *different protocols*:

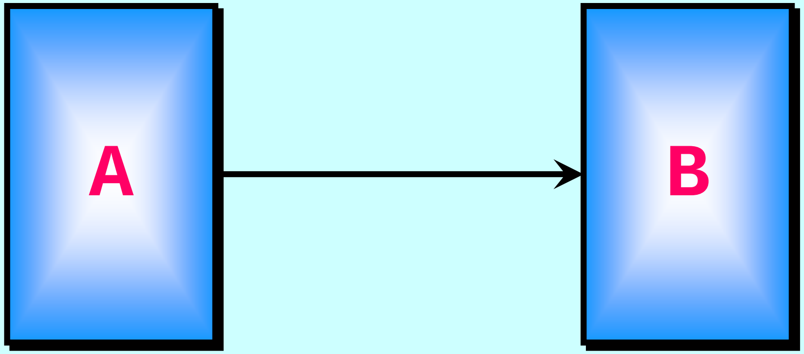


The sender process just uses the appropriate channel for the message it wants to send.

The receiver process listens on all channels (using an **ALT**) from the sender.

Variant Protocol

occam-π provides a direct *(and more efficient)* mechanism – the *variant* (or **CASE**) **PROTOCOL** – that allows *different* kinds of message to be sent along a single channel:



The sender process prefixes each message with a *tag* **BYTE** that identifies its structure. Each *variant* has a unique *tag*.

The receiver process listens on the one channel for the *tag* **BYTE**, using that in a **CASE** *(switching)* mechanism to input the rest of the message.

Example Protocols

```
PROTOCOL STRING IS BYTE::[]BYTE:

PROTOCOL PACKET IS INT::[]REAL32:
```

counted array

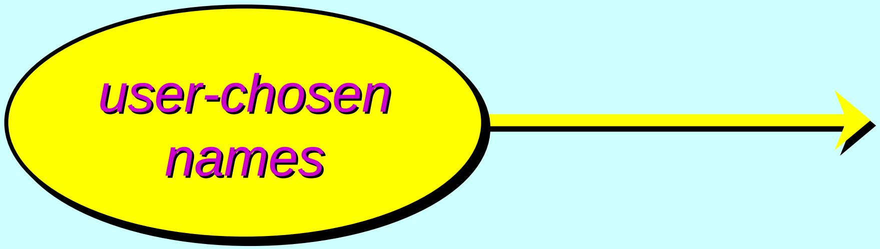
```
PROTOCOL MESSAGE IS STRING; PACKET:
```

sequential

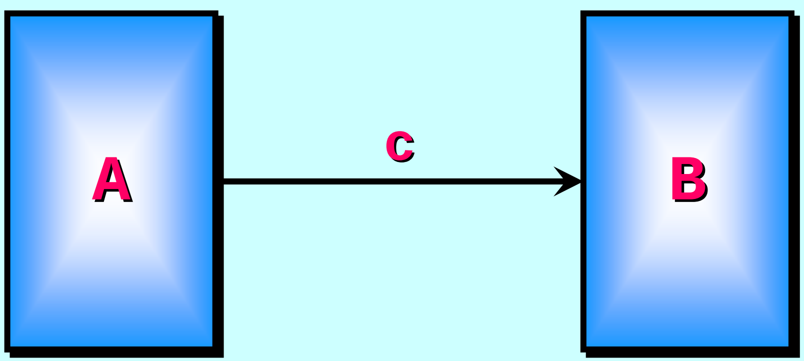
```
PROTOCOL ALTERNATIVES
  CASE
    dog; INT
    cat; STRING
    pig; PACKET
    canary; MESSAGE
    poison
:
```

variant

Variant Protocol



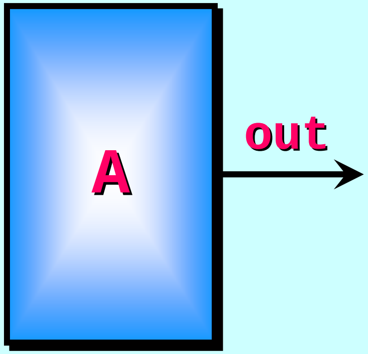

```
PROTOCOL ALTERNATIVES
  CASE
    dog; INT
    cat; STRING
    pig; PACKET
    canary; MESSAGE
    poison
:
```




```
CHAN ALTERNATIVES c:
PAR
  A (c!)
  B (c?)
```

Variant Protocol

```
PROC A (CHAN ALTERNATIVES out!)

  VAL []BYTE s IS "sat on the mat":
  [255]BYTE a:
  [1000]REAL32 b, c:

  SEQ
    ...  initalise a, b and c
    out ! dog; 42
    out ! cat; 6::s
    out ! pig; (SIZE b)::b
    out ! canary; (BYTE (SIZE a))::a; (SIZE b)::b
    ...  more stuff
    out ! poison
:
```

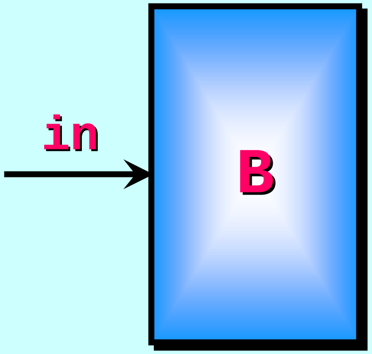


Only the first **6 BYTE**s of **s** (**"sat on")** are sent.

The **SIZE** of an array is always an **INT**. So, this must be *cast* into the **BYTE** needed for the first *counted array* component of the **canary** variant.

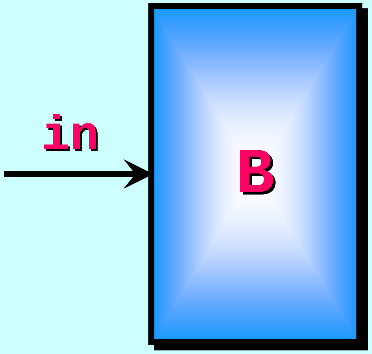
Variant Protocol

```
PROC B (CHAN ALTERNATIVES in?)

  [255]BYTE s:
  [1024]REAL32 x:
  INT state:
  INITIAL BOOL running IS TRUE:

  WHILE running
    ...  process input alternatives
:
```



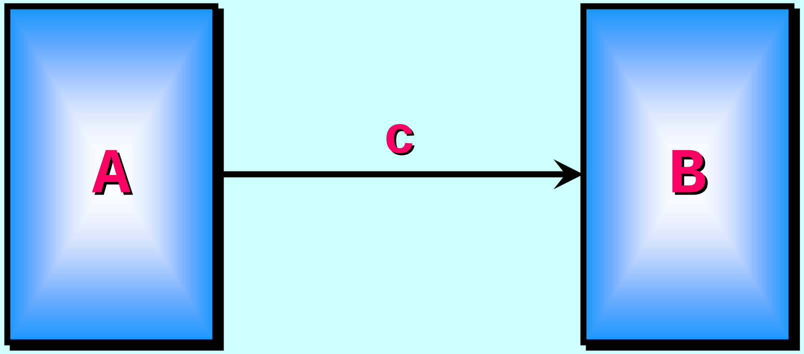
Variant Protocol

```
WHILE running
  in ? CASE
    dog; state
      ...  deal with this variant
    BYTE size:
    cat; size::s
      ...  deal with this variant
    INT size:
    pig; size::x
      ...  deal with this variant
    BYTE size.s:
    INT size.x:
    canary; size.s::s; size.x::x
      ...  deal with this variant
    poison
      running := FALSE
```




```
[255]BYTE s:
[1024]REAL32 x:
INT state:
BOOL running:
```

Variant Protocol




```
CHAN ALTERNATIVES c:
PAR
  A (c!)
  B (c?)
```

```
PROTOCOL ALTERNATIVES
  CASE
    dog; INT
    cat; STRING
    pig; PACKET
    canary; MESSAGE
    poison
:
```

Notice the *higher-level protocol* employed here …

If the **poison** variant is sent, it is the *last message* sent on the channel.