Choice and Non-Determinism

Peter Welch (p.h.welch@kent.ac.uk)
Computing Laboratory, University of Kent at Canterbury

Co631 (Concurrency)

Choice and Non-Determinism

Non-determinism ...

The **ALT** and **PRI ALT** …

Control and real-time …

Resets and kills …

Memory cells ...

Pre-conditioned guards …

Serial FIFO *('ring')* buffer …

The replicated **ALT** …

Nested **ALT**s …

Deterministic Processes (CSP)

So far, our parallel systems have been ***deterministic***:

- the values in the output streams depend only on the values in the input streams;
- the semantics is scheduling independent;
- no race hazards are possible.

CSP parallelism, on its own, ***does not introduce non-determinism***.

This gives a firm foundation for exploring real-world models which cannot always behave so simply.

Non-Deterministic Processes (CSP)

In the real world, it is sometimes the case that things happen as a result of:

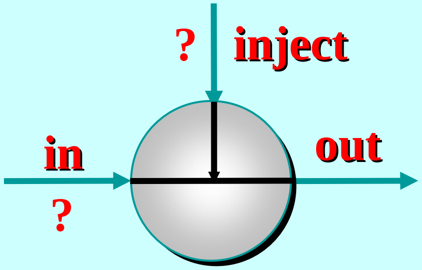
- what happened in the past;
- when (or, at least, in what order) things happened.

In this world, things are scheduling dependent.

CSP (and **occam-π**) addresses these issues ***explicitly***.

Non-determinism does not arise by default.

A Control Process



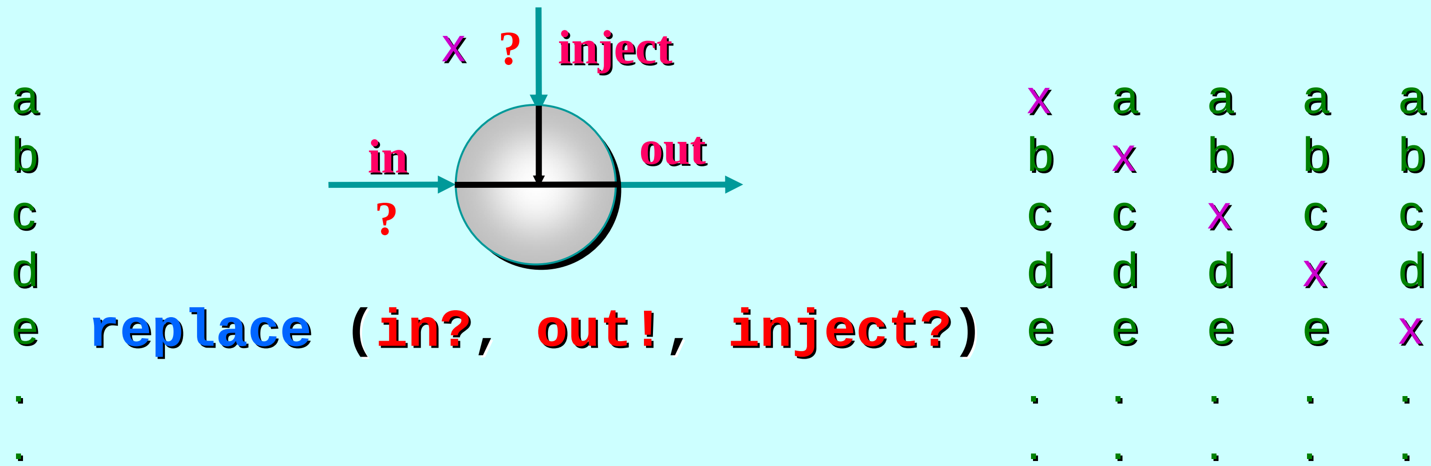
replace (in?, out!, inject?)

Coping with the real world - making choices …

In **replace**, data normally flows from **in?** to **out!** unchanged.

However, if something arrives on **inject?**, it is output on **out!** - *instead of* the next input from **in?**.

A Control Process

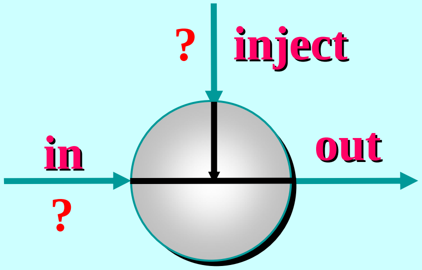


The **out!** stream depends upon:

- The values contained in the **in** and **inject** streams;
- the ***order*** in which those values arrive.

The **out!** stream is ***not*** determined just by the **in?** and **inject?** streams - it is ***non-deterministic***.

A Control Process

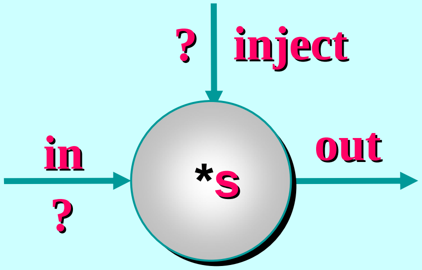



```
replace (in?, out!, inject?) =
  (inject?x --> ((in?a --> SKIP) || (out!x --> SKIP))
   [PRI]
   in?a --> out!a --> SKIP
  );
  replace (in?, out!, inject?)
```

for information only …

Note: `[]` is the (external) choice operator of CSP.
`[PRI]` is a prioritised version - giving priority to the event on its left.

Another Control Process



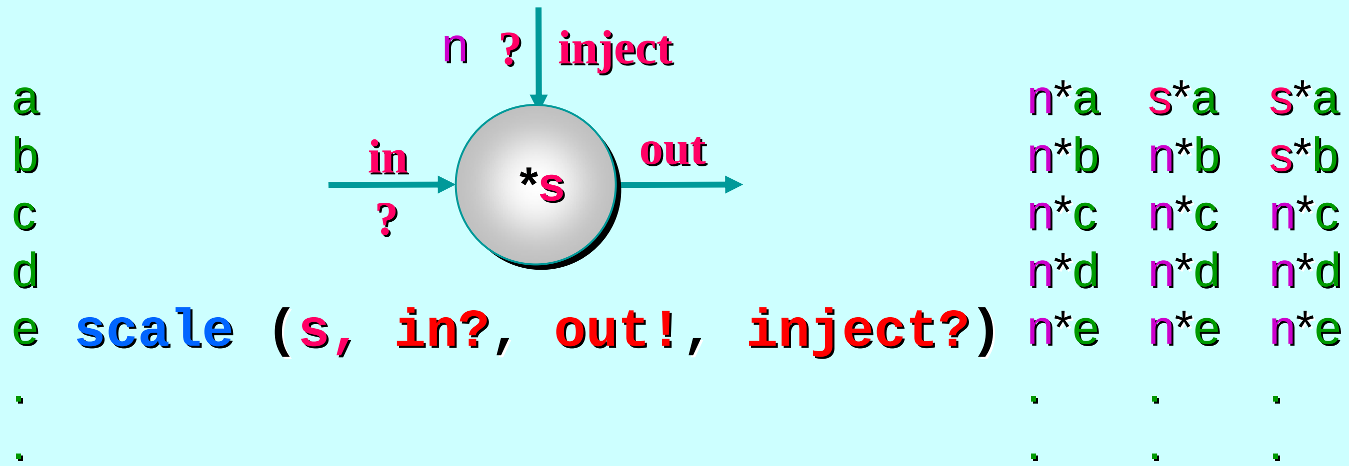
scale (s, in?, out!, inject?)

Coping with the real world - making choices …

In **scale**, data flows from **in?** to **out!**, getting scaled by a factor of **s** as it passes.

Values arriving on **inject?** reset the **s** factor.

Another Control Process

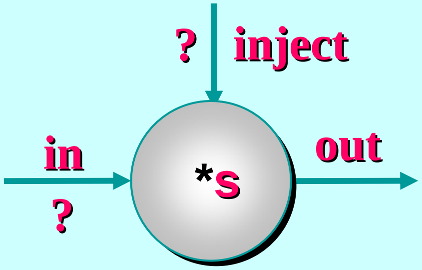


The **out!** stream depends upon:

- The values contained in the **in?** and **inject!** streams;
- the ***order*** in which those values arrive.

The **out!** stream is ***not*** determined just by the **in?** and **inject?** streams - it is ***non-deterministic***.

Another Control Process

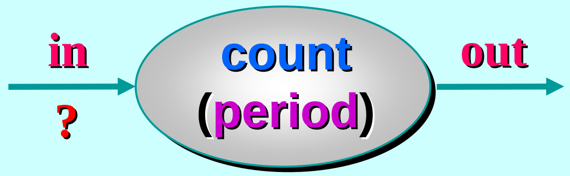



```
scale (s, in?, out!, inject?) =
  (inject?s --> SKIP
   [PRI]
   in?a --> out!s*a --> SKIP
  );
 scale (s, in?, out!, inject?)
```

for information only …

Note: **[]** is the (external) choice operator of CSP.
[PRI] is a prioritised version - giving priority to the event on its left.

A Real-Time Process

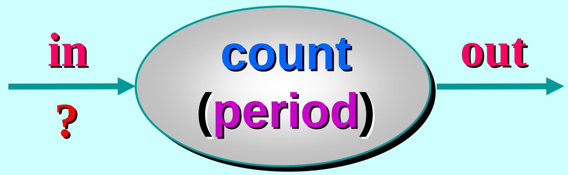



```
count (period, in?, out!)
```

Coping with the real world - making choices …

count observes passing time and messages arriving on **in?**. Every **period** microseconds, it outputs (on **out!**) the number of messages received during the previous **period**.

A Real-Time Process



a b c d e . .

`count (period, in?, out!)`

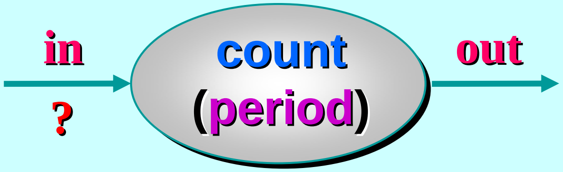
4	0	5
7	2	5
0	0	3
3	5	0
3	8	0
.	.	.
.	.	.

The **out!** stream depends upon:

- ***When*** values arrived on the **in?** stream (the values received are irrelevant).

The **out!** stream is ***not*** determined by the **in?** stream values - it is ***non-deterministic***.

A Real-Time Process



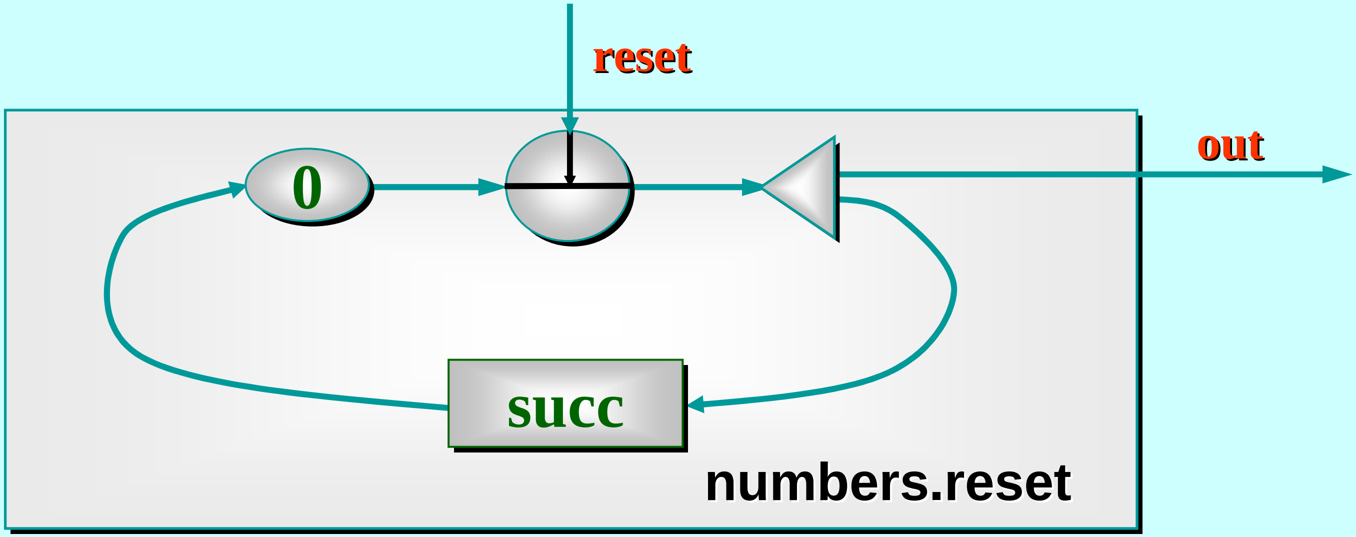
`count (period, in?, out!)`

```
count (period, in?, out!) =
```

standard CSP does not address time …

but occam-π does …

A Resettable Network



This is a *resettable* version of the **numbers** process.

If nothing is sent down **reset**, it behaves as before.

But it may be **reset** to continue counting from *any* number at *any* time.

Non-Deterministic Processes

To enable these, **occam-π** introduces a new programming structure: the **`ALT`** **...**

… which explicitly introduces *non-determinism*.

a very simple and elegant idea

will not frighten the horses …

Choice and Non-Determinism

Non-determinism ...

The **ALT** and **PRI ALT** …

Control and real-time …

Resets and kills …

Memory cells ...

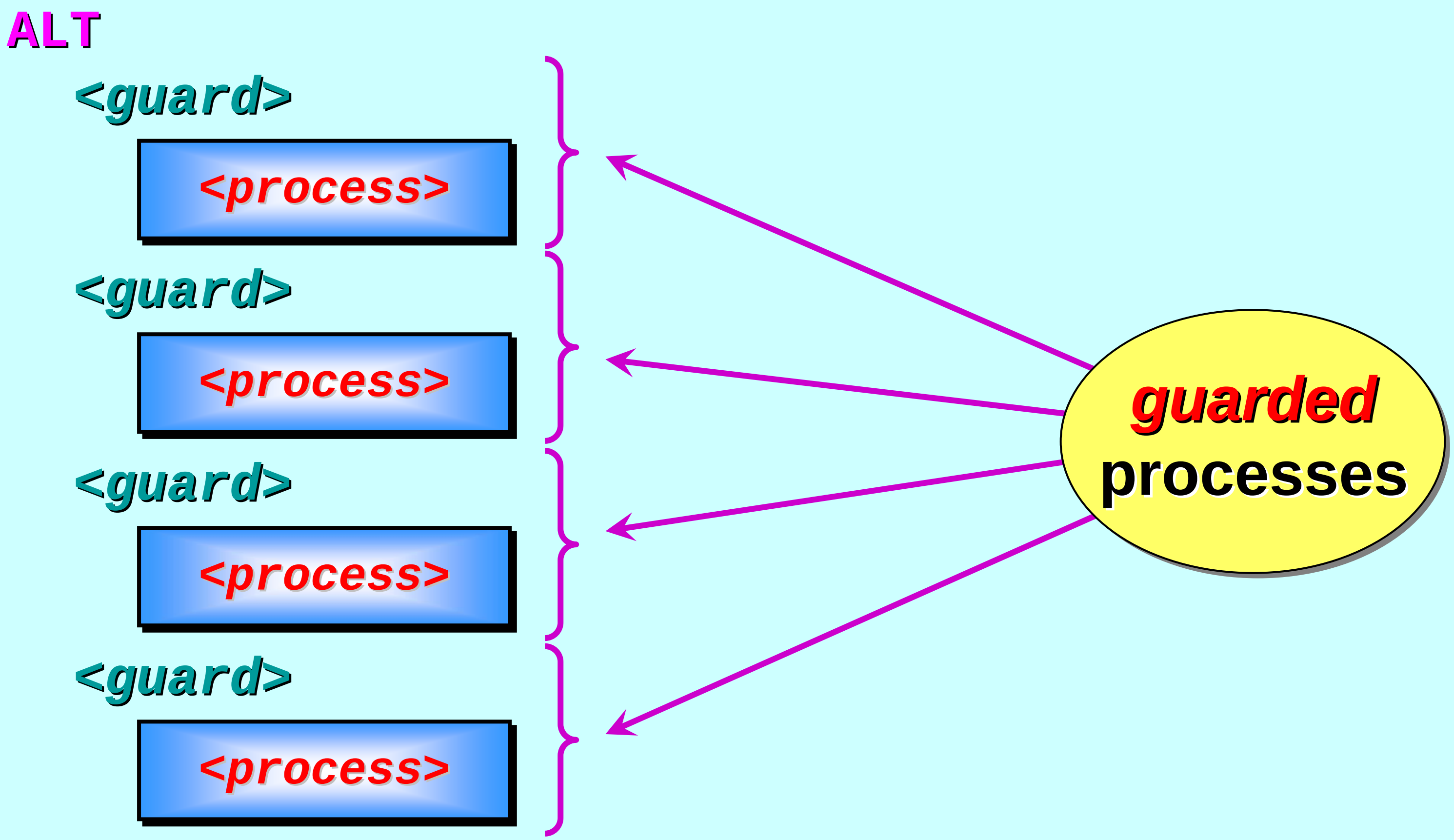
Pre-conditioned guards …

Serial FIFO *('ring')* buffer …

The replicated **ALT** …

Nested **ALT**s …

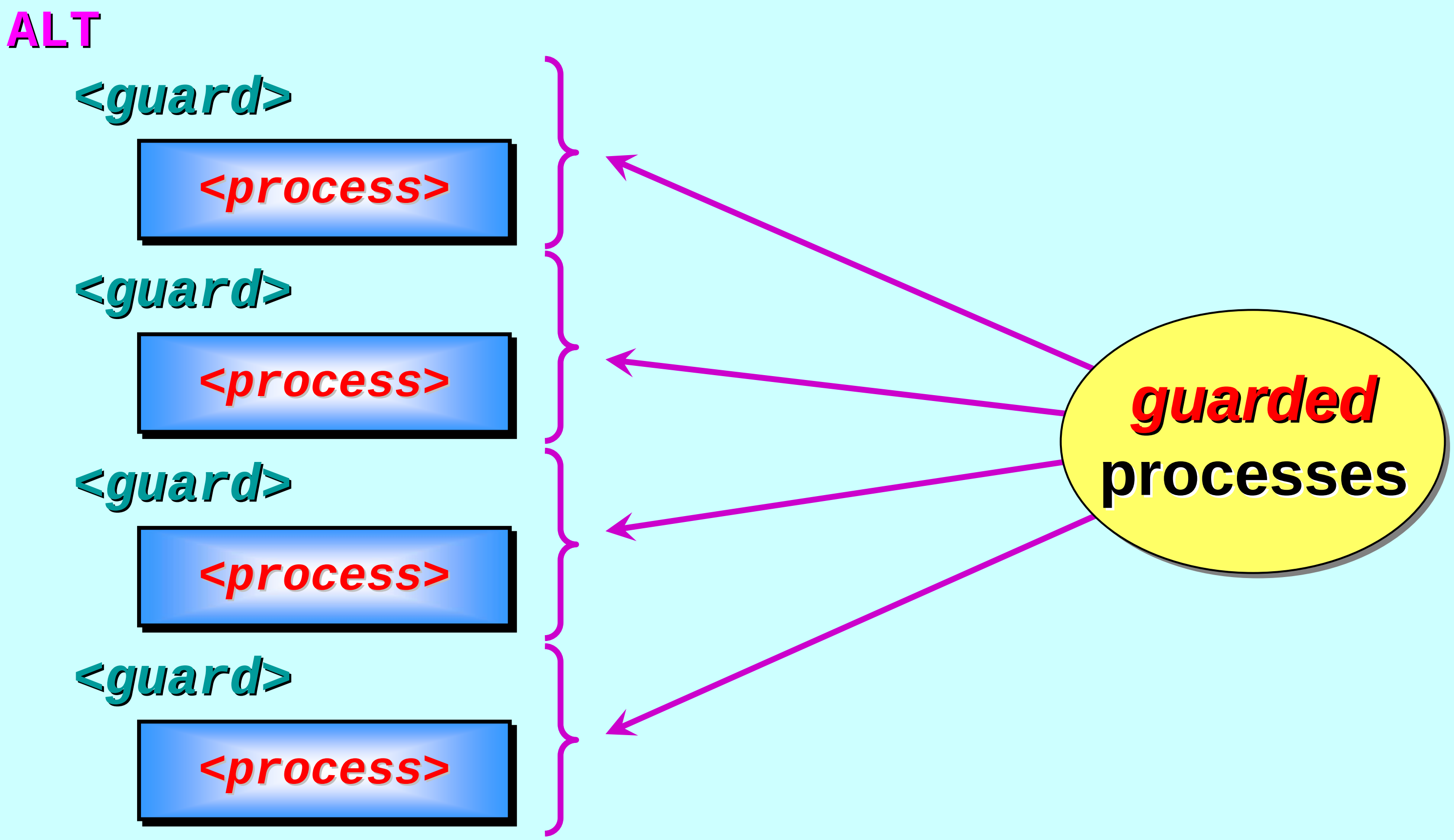
Non-Deterministic Choice



Non-Deterministic Choice

- A **<guard>** may be *ready* or *not-ready*.
- A *not-ready* **<guard>** may change to *ready* as a result of external activity.
- A *ready* **<guard>** may be executed.

Non-Deterministic Choice



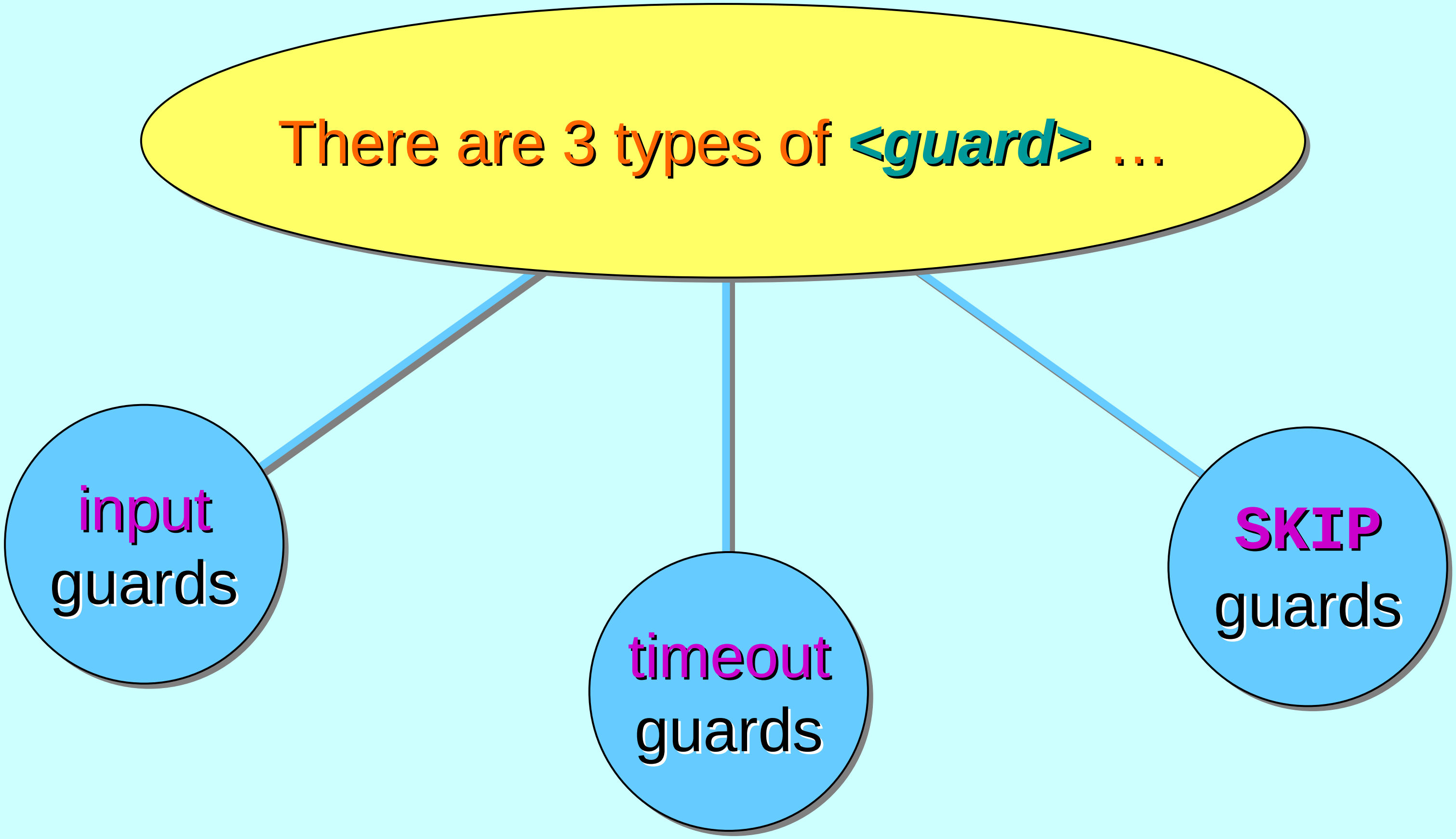
Non-Deterministic Choice

An **ALT** process executes as follows:

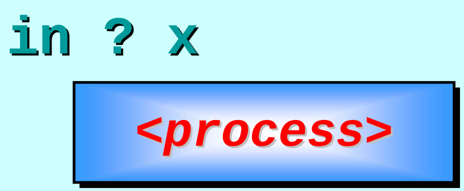
- if no guard is ready, the process is suspended until one, or more, become ready;
- if one guard is ready, execute it and then execute the process it was defending *(end of **ALT** process)*;
- if more than one guard is ready, one is *arbitrarily chosen* and executes, followed by the process it was defending *(end of **ALT** process)*.

Note: only *one* of the guarded processes is executed.

Non-Deterministic Choice



Non-Deterministic Choice



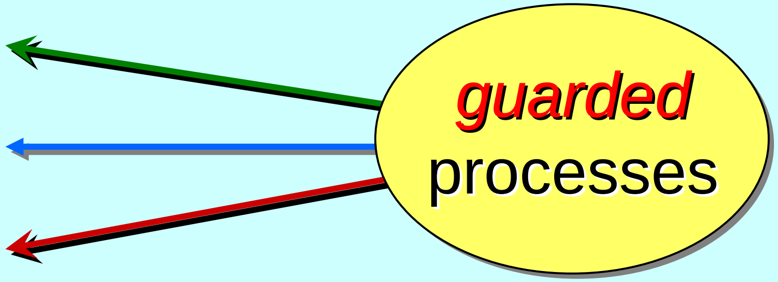
input guards

An input guard is *ready* if a process on the other end of the channel is trying to output to that channel and is waiting for its message to be taken.

Execution of this guard *(if chosen)* is just execution of the input process. Note that execution of this guard leaves it *not-ready* (until another process again outputs to the channel).

Non-Deterministic Choice


```
PROC crude.plex (CHAN INT in.0?, in.1?, in.2?, out!)
  WHILE TRUE
    INT x:
    ALT
      in.0 ? x
        out ! x
      in.1 ? x
        out ! x
      in.2 ? x
        out ! x
:
```



Non-Deterministic Choice

tim ? AFTER t

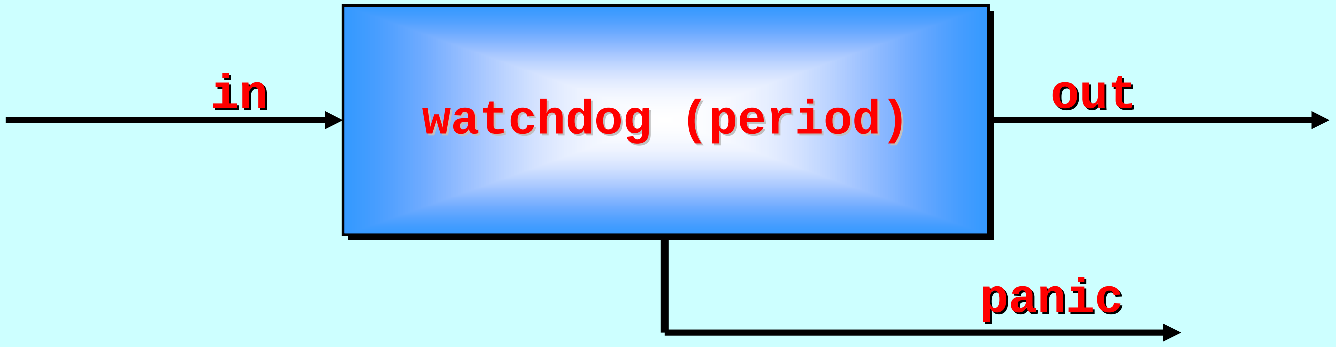
<process>

timeout guards

A timeout guard is *ready* if the time currently showing on the **TIMER** (**tim**) is **AFTER** the time indicated (**t**). Note that the time on a **TIMER** continually increments and that the time indicated cannot change while awaiting this timeout.

Execution of this guard *(if chosen)* is null. Note that execution of this guard leaves it *ready* (until the value of timeout is changed).

Non-Deterministic Choice




```
PROC watchdog (VAL INT period,
                CHAN INT in?, out!, CHAN BOOL panic!)
  WHILE TRUE
    TIMER tim:
    INT t, x:
    SEQ
      tim ? t
      ALT
        in ? x
          out ! x
        tim ? AFTER t PLUS period
          panic ! TRUE
:
```

Non-Deterministic Choice

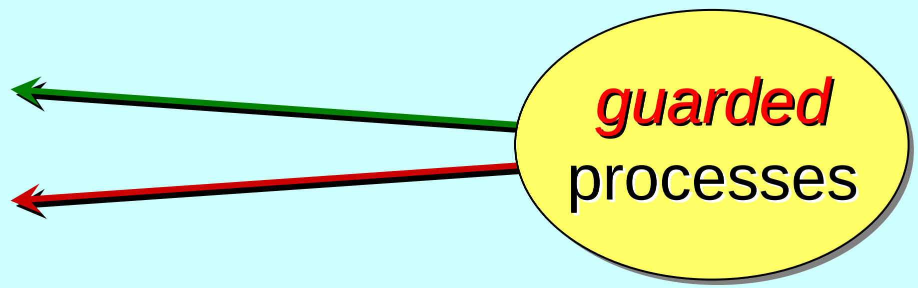
SKIP
guards

A **SKIP** guard is always ready.

Execution of this guard *(if chosen)* is null.

Non-Deterministic Choice

```
ALT
  SKIP
    x := 42
  SKIP
    x := 43
```

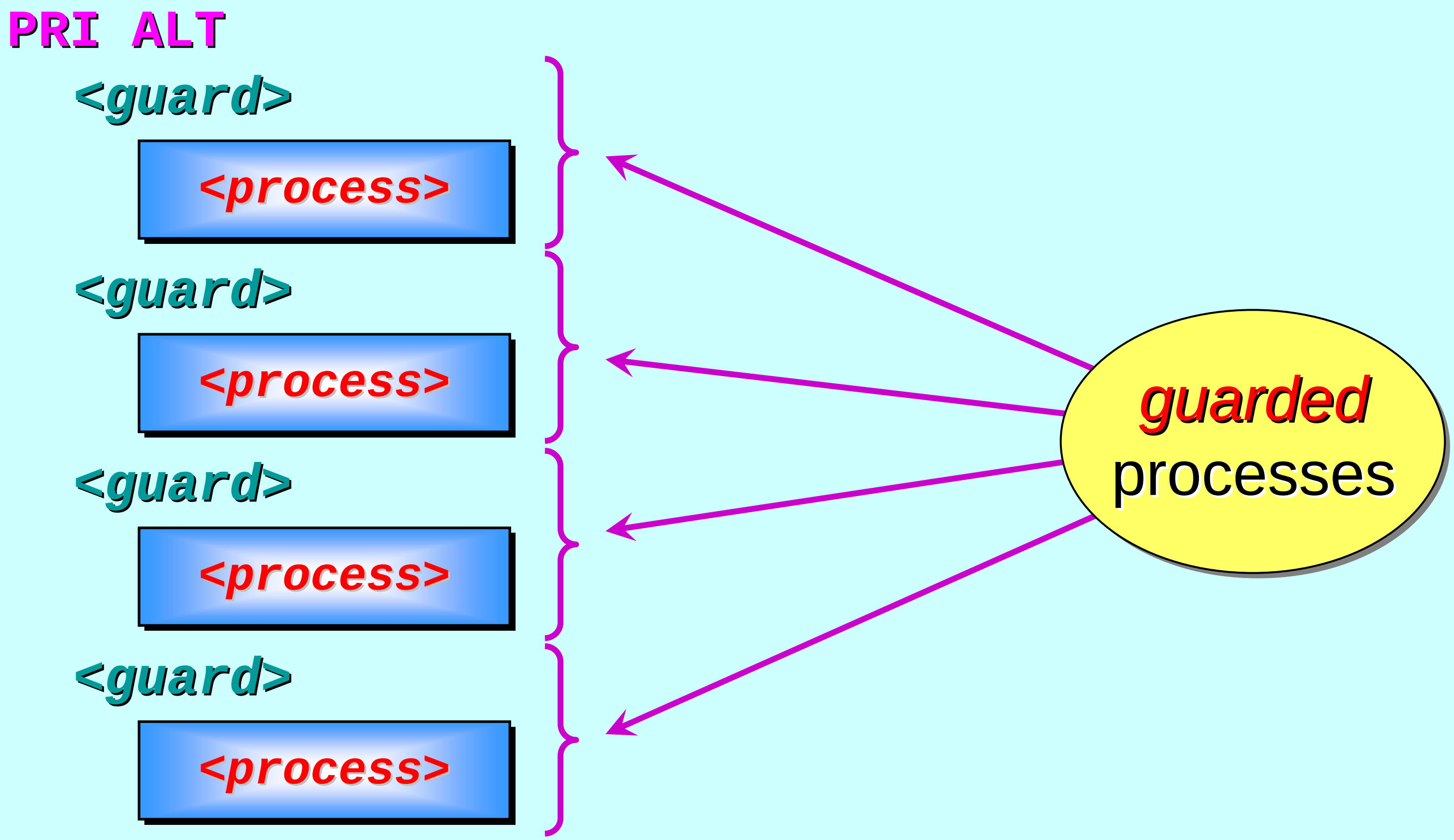


Both guards are ready – so an *arbitrary choice* is made!

Actually, such non-determinism is too much to be useful and the compiler issues warnings – *the programmer probably didn't mean to write this!*

SKIP guards only become useful with *prioritised choice*, which comes next.

Deterministic Choice



Deterministic Choice

A **`PRI ALT`** process executes as follows:

- if no guard is ready, the process is suspended until one, or more, become ready;
- if one guard is ready, execute it and then execute the process it was defending *(end of **`PRI ALT`** process)*;
- if more than one guard is ready, ***the first one listed is chosen*** and executes, followed by the process it was defending *(end of **`PRI ALT`** process)*.

Note: only *one* of the guarded processes is executed.

Example – Polling a Channel

```
PRI ALT
  in ? x
    ...  message was pending
  SKIP
    ...  message was not pending
```

guarded **processes**

If no message was pending on the channel, the first guard is *not-ready*. But the second guard is (always) *ready*, so that guarded process is executed.

If a message was pending on the channel, the first guard is *ready*. So (always) is the second guard – but the first has priority and is taken.

A **SKIP** guard lets us poll channels to test if a message is pending and, if so, deal with it. *Beware polling though – it can lead to inefficient and poor design …*

Choice and Non-Determinism

Non-determinism ...

The **ALT** and **PRI ALT** …

Control and real-time …

Resets and kills …

Memory cells ...

Pre-conditioned guards …

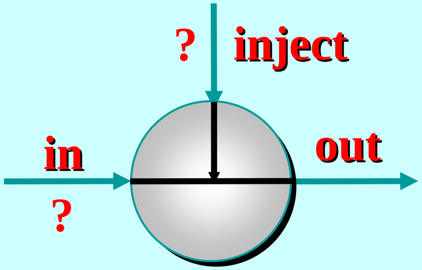
Serial FIFO *(‘ring’)* buffer …

The replicated **ALT** …

Nested **ALT**s …

Example – a Control Process

earlier example …



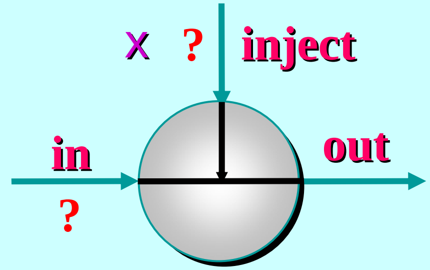
replace (in?, out!, inject?)

Coping with the real world - making choices …

In **replace**, data normally flows from **in?** to **out!** unchanged.

However, if something arrives on **inject?**, it is output on **out!** - *instead of* the next input from **in?**.

Example – a Control Process



a		x	a	a	a	a
b		b	x	b	b	b
c		c	c	x	c	c
d		d	d	d	x	d
e	**replace (in?, out!, inject?)**	e	e	e	e	x
.		.	.	.	.	.
.		.	.	.	.	.

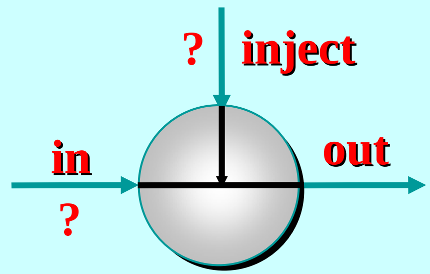
The **out!** stream depends upon:

- The *values* contained in the **in** and **inject** streams;
- the *order* in which those values arrive.

The **out!** stream is *not* determined just by the **in?** and **inject?** streams - it is *non-deterministic*.

Example – a Control Process

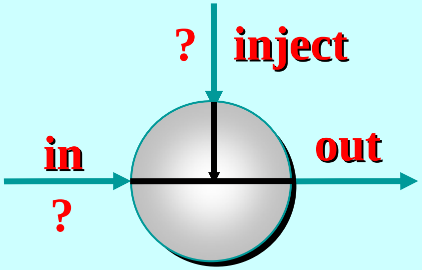
coding ...




```
PROC replace (CHAN INT in?, out!, inject?)
  WHILE TRUE
    INT x, any:
    PRI ALT
      inject ? x       -- replace the
        PAR            -- next ‘in’
          in ? any     -- with the
          out ! x      -- ‘inject’ value
      in ? x           -- normally
        out ! x        -- just copy through
:
```

Example – a Control Process

better coding ...




```
PROC replace (CHAN INT in?, out!, inject?)
  WHILE TRUE
    PRI ALT
      INT x, any:
      inject ? x        -- replace the
        PAR             -- next 'in'
          in ? any      -- with the
          out ! x       -- 'reset' value
      INT x:
      in ? x            -- normally
        out ! x         -- just copy through
:
```

local declaration

local declaration

Locals + Guarded Processes

```
ALT              -- or PRI ALT
  <local declarations>
  <guard>
    <process>

  <local declarations>
  <guard>
    <process>

  <local declarations>
  <guard>
    <process>
```

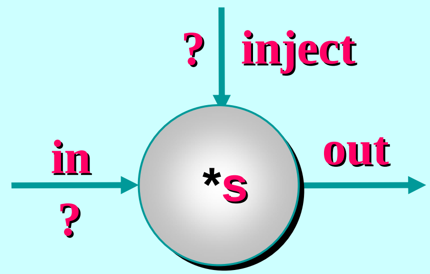
local declarations are optional

guarded **processes**

local declarations have scope only for the following guarded process

Example – another Control Process

earlier example ...



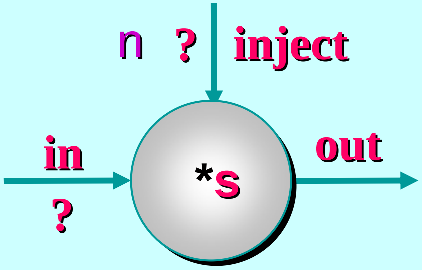

```
scale (s, in?, out!, inject?)
```

Coping with the real world - making choices …

In **`scale`**, data flows from **`in?`** to **`out!`**, getting scaled by a factor of **`s`** as it passes.

Values arriving on **`inject?`** reset the **`s`** factor.

Example – another Control Process




```
a                                      n*a  s*a  s*a
b                                      n*b  n*b  s*b
c                                      n*c  n*c  n*c
d                                      n*d  n*d  n*d
e scale (s, in?, out!, inject?)        n*e  n*e  n*e
.                                      .    .    .
.                                      .    .    .
```

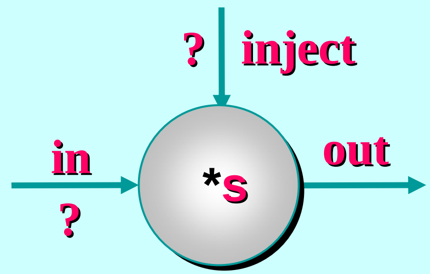
The **`out!`** stream depends upon:

- The *values* contained in the **`in?`** and **`inject!`** streams;
- the *order* in which those values arrive.

The **`out!`** stream is *not* determined just by the **`in?`** and **`inject?`** streams - it is *non-deterministic*.

Example – another Control Process

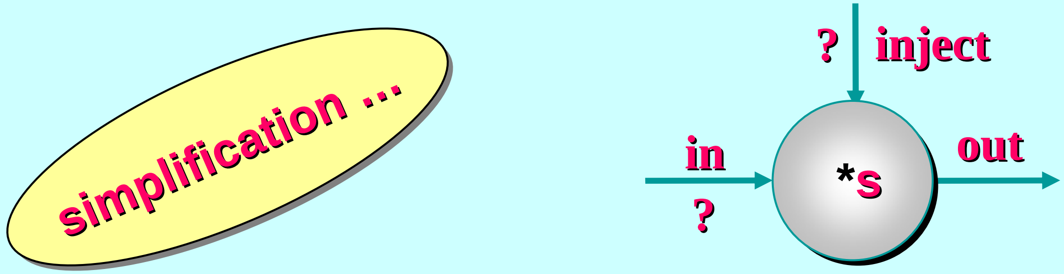
coding …




```
PROC scale (VAL INT s, CHAN INT in?, out!, inject?)
  INT scale:
  SEQ
    scale := s
    WHILE TRUE
      PRI ALT
        inject ? scale            -- get new scale
          SKIP
        INT x:
        in ? x                    -- data
          out ! scale*x           -- scale it up
:
```

local declaration

Example – another Control Process

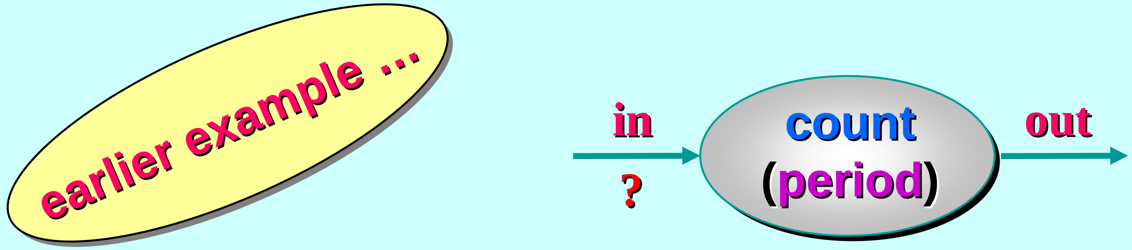



```
PROC scale (VAL INT s, CHAN INT in?, out!, inject?)
  INITIAL INT scale IS s:
  WHILE TRUE
    PRI ALT
      inject ? scale        -- get new scale
        SKIP
      INT x:
      in ? x                -- data
        out ! scale*x       -- scale it up
:
```

initialising declaration

local declaration

Example – a Real-Time Process

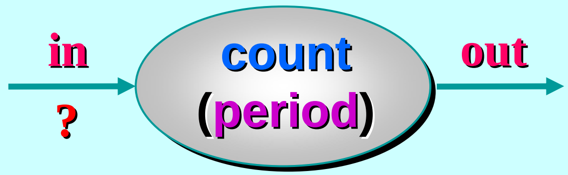


count (period, in?, out!)

Coping with the real world - making choices …

count observes passing time and messages arriving on **in?**. Every **period** microseconds, it outputs (on **out!**) the number of messages received during the previous **period**.

Example – a Real-Time Process



a b c d e . .

`count (period, in?, out!)`

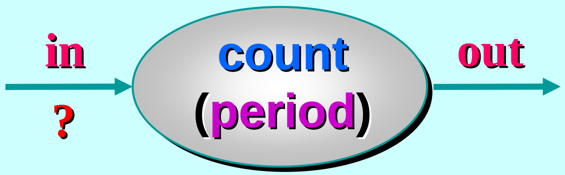
4	0	5
7	2	5
0	0	3
3	5	0
3	8	0
.	.	.
.	.	.

The **`out!`** stream depends upon:

- *When* values arrived on the **`in?`** stream (the values received are irrelevant).

The **`out!`** stream is *not* determined by the **`in?`** stream values - it is *non-deterministic*.

Example – a Real-Time Process




```
PROC count (VAL INT period, CHAN INT in?, out!)
  INITIAL INT seen IS 0:
  TIMER tim:
  INT timeout:
  SEQ
    tim ? timeout
    timeout := timeout PLUS period
    WHILE TRUE
      PRI ALT
        tim ? AFTER timeout     -- timeout
          SEQ
            out ! seen
            seen := 0
            timeout := timeout PLUS period
        INT any:
        in ? any                -- data
          seen := seen + 1
:
```

coding ...

Choice and Non-Determinism

Non-determinism ...

The **ALT** and **PRI ALT** …

Control and real-time …

Resets and kills …

Memory cells ...

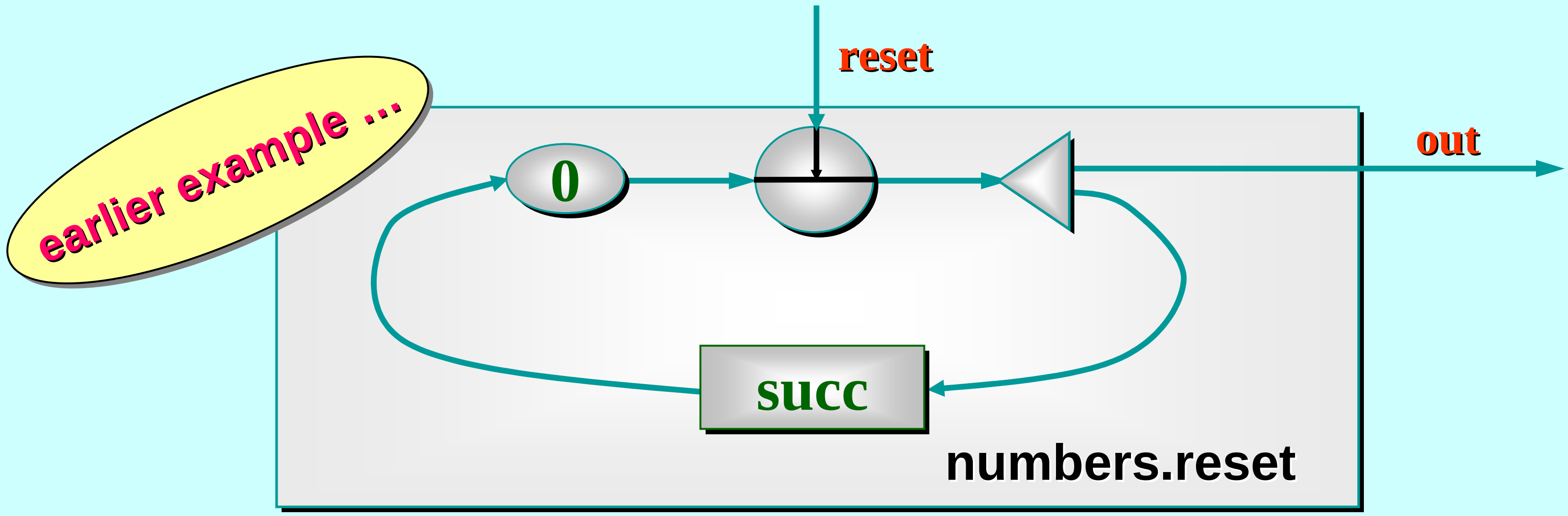
Pre-conditioned guards …

Serial FIFO *(‘ring’)* buffer …

The replicated **ALT** …

Nested **ALT**s …

Example – a Resettable Network

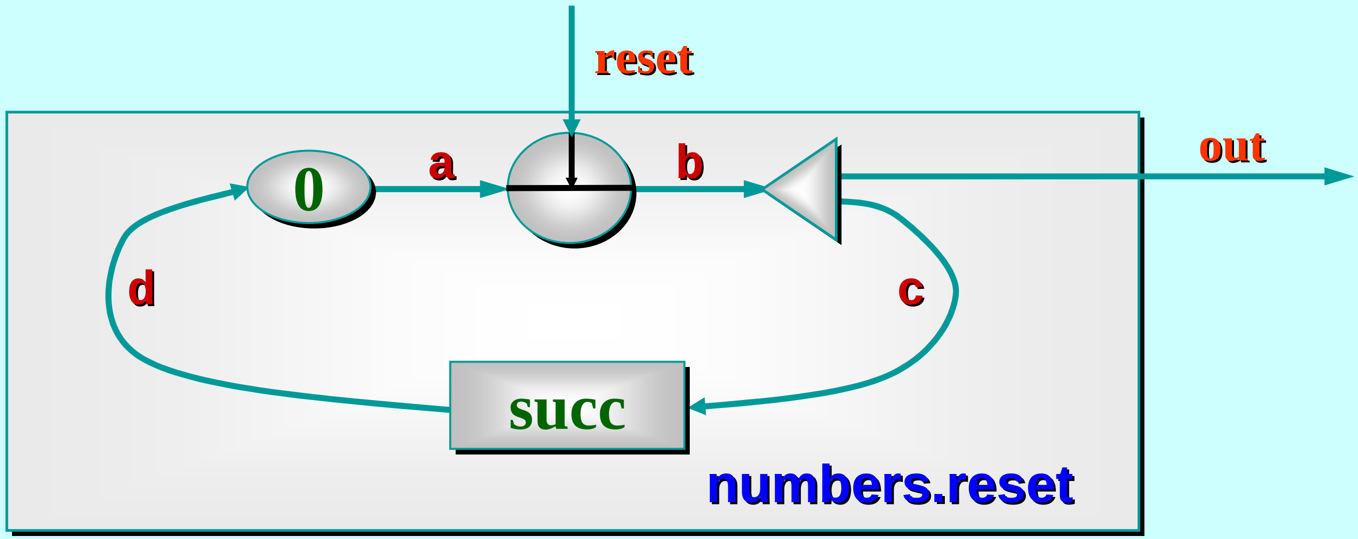


This is a *resettable* version of the **numbers** process.

If nothing is sent down **reset**, it behaves as before.

But it may be **reset** to continue counting from *any* number at *any* time.

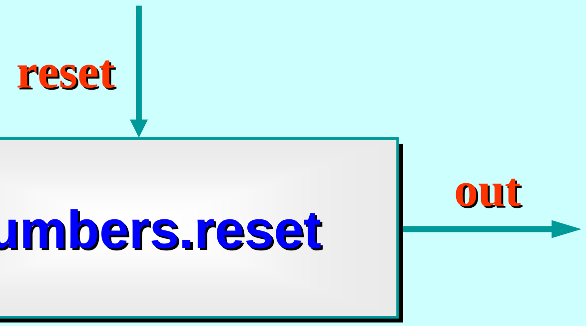
Example – a Resettable Network




```
PROC numbers.reset (CHAN INT reset?, out!)
  CHAN INT a, b, c, d:
  PAR
    prefix (0, d?, a!)
    replace (a?, b!, reset?)
    delta (b?, out!, c!)
    succ (c?, d!)
:
```

parallel implementation

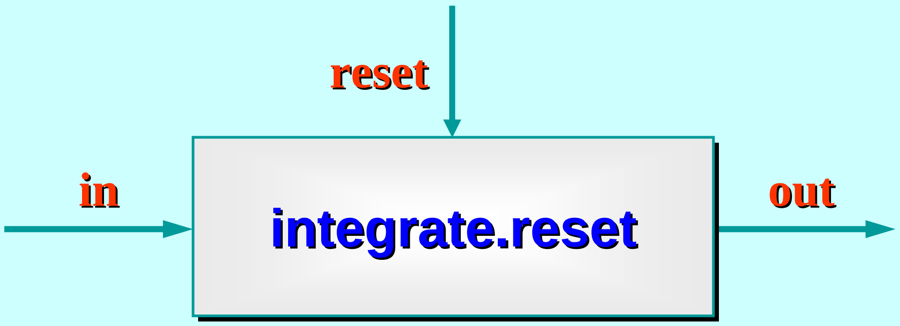
Example – a Resettable Network




```
PROC numbers.reset (CHAN INT reset?, out!)
  INITIAL INT n IS 0:
  WHILE TRUE
    SEQ
      PRI ALT     -- poll reset channel
        reset ? n
          SKIP
        SKIP
          SKIP
      out ! n
      n := n PLUS 1
:
```

serial implementation

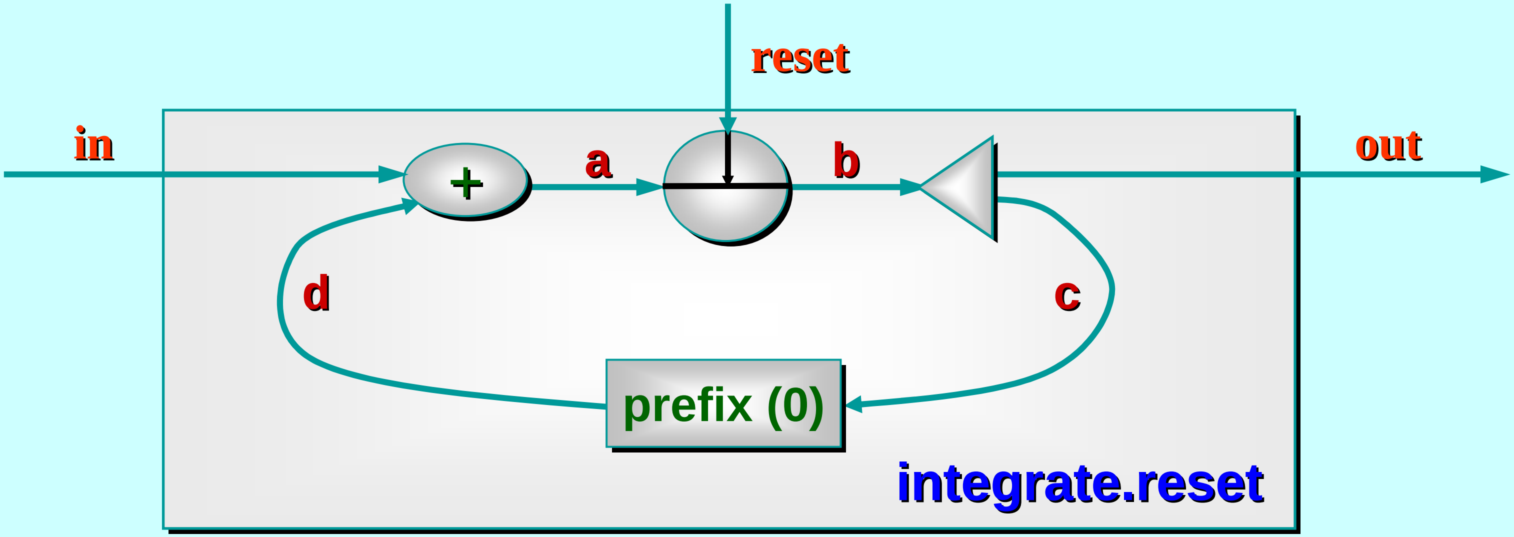
Example – Resettable Integrator




```
PROC integrate.reset (CHAN INT in?, reset?, out!)
  INITIAL INT total IS 0:
  WHILE TRUE
    SEQ
      PRI ALT
        reset ? total
          SKIP
        INT x:
        in ? x
          total := total + x
      out ! total
:
```

serial implementation

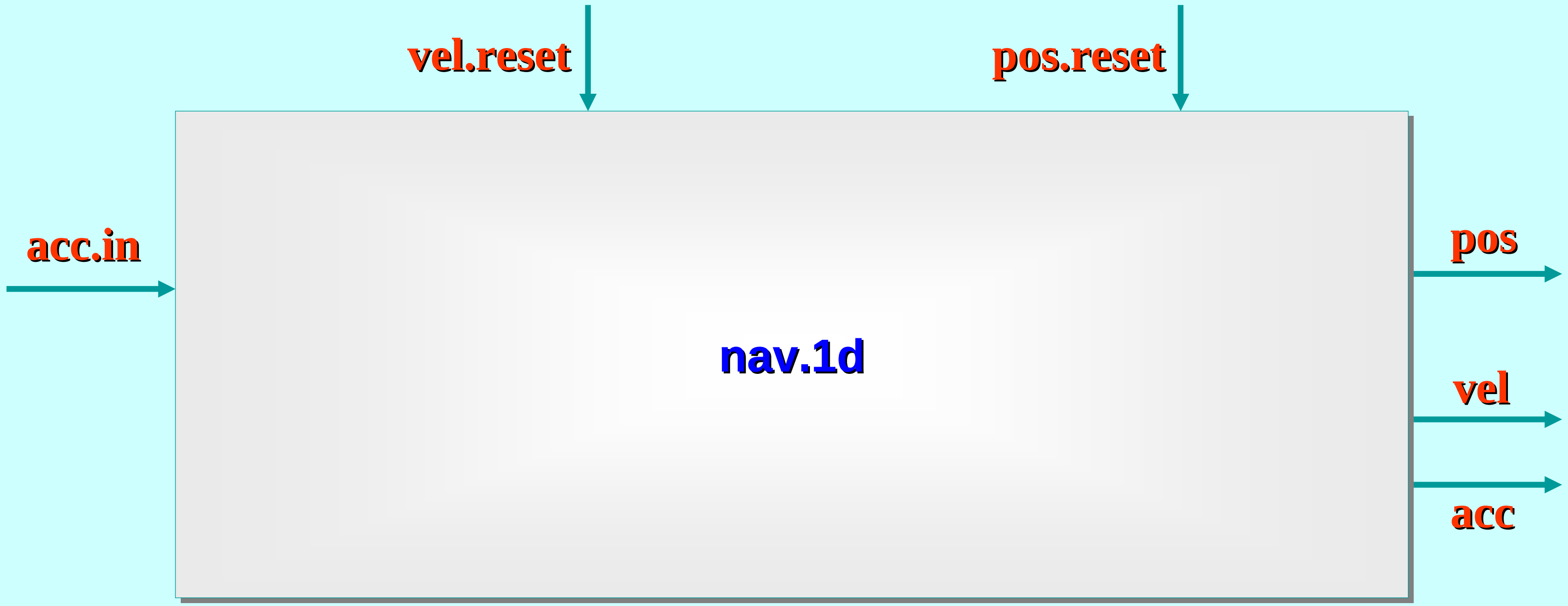
Example – Resettable Integrator




```
PROC integrate.reset (CHAN INT in?, reset?, out!)
  CHAN INT a, b, c, d:
  PAR
    plus (in?, d?, a!)
    replace (a?, b!, reset?)
    delta (b?, out!, c!)
    prefix (0, c?, d!)
:
```

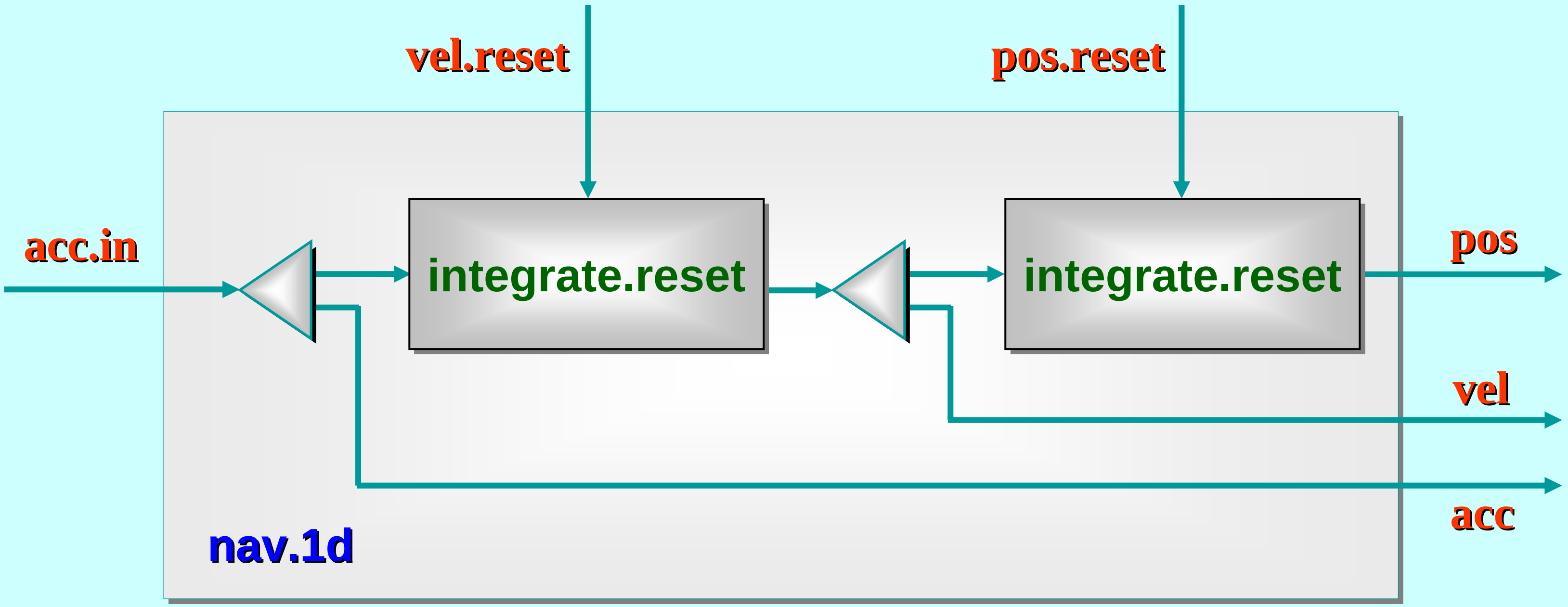
parallel implementation

An Inertial Navigation Component



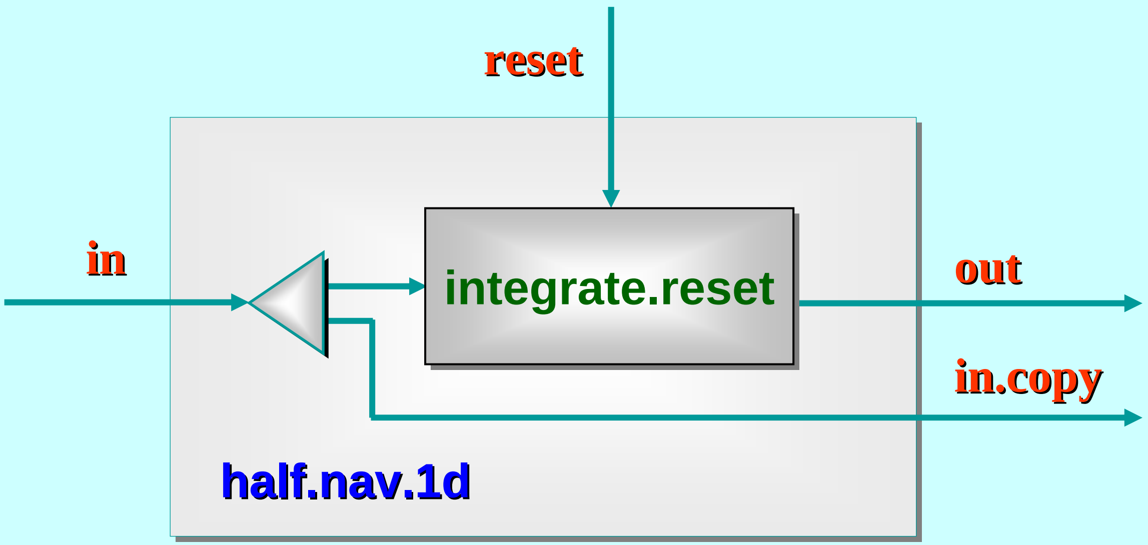
- **`acc.in`**: carries *regular* accelerometer samples;
- **`vel.reset`**: velocity *initialisation* and *corrections*;
- **`pos.reset`**: position *initialisation* and *corrections*;
- **`pos/vel/acc`**: *regular* outputs.

An Inertial Navigation Component



- **`acc.in`**: carries *regular* accelerometer samples;
- **`vel.reset`**: velocity *initialisation* and *corrections*;
- **`pos.reset`**: position *initialisation* and *corrections*;
- **`pos/vel/acc`**: *regular* outputs.

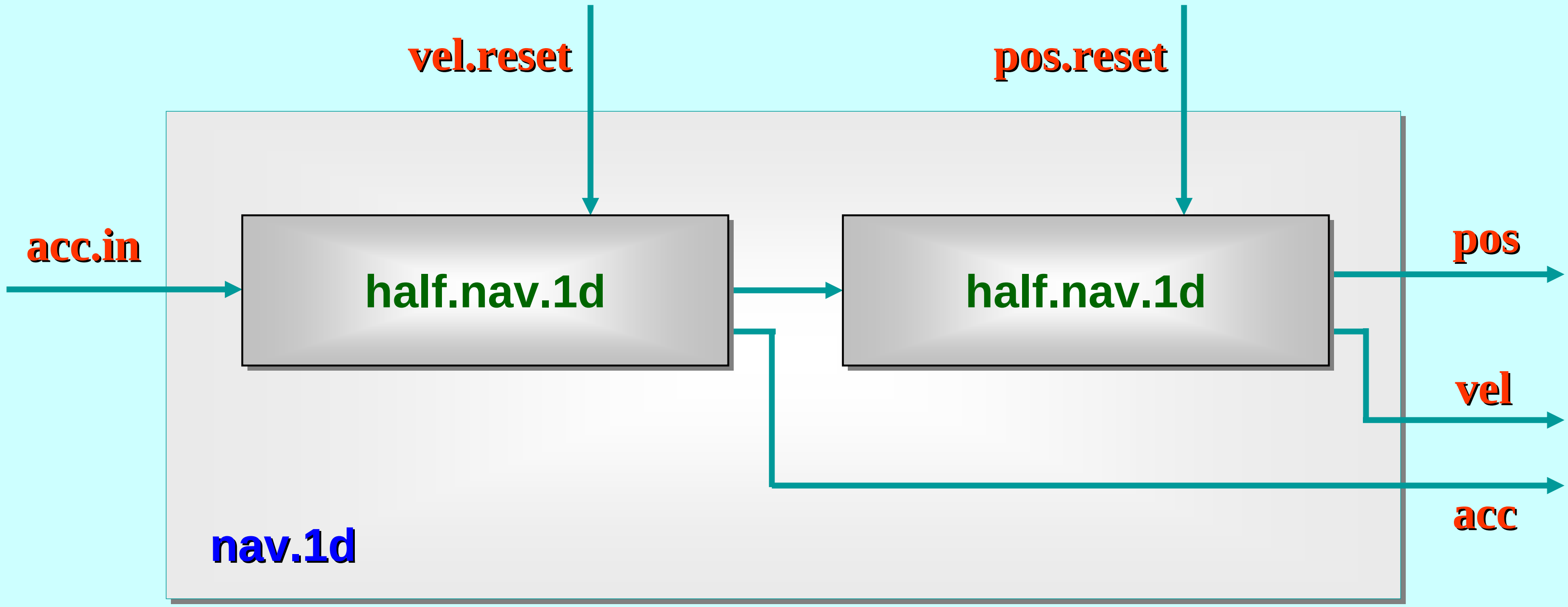
Half Inertial Navigation Component



Build it from two components

- **in**: carries *regular* samples;
- **in.copy**: copy of the **in** stream;
- **out**: *regular* outputs *(sample running sums)*;
- **reset**: running sum *initialisation* and *corrections*.

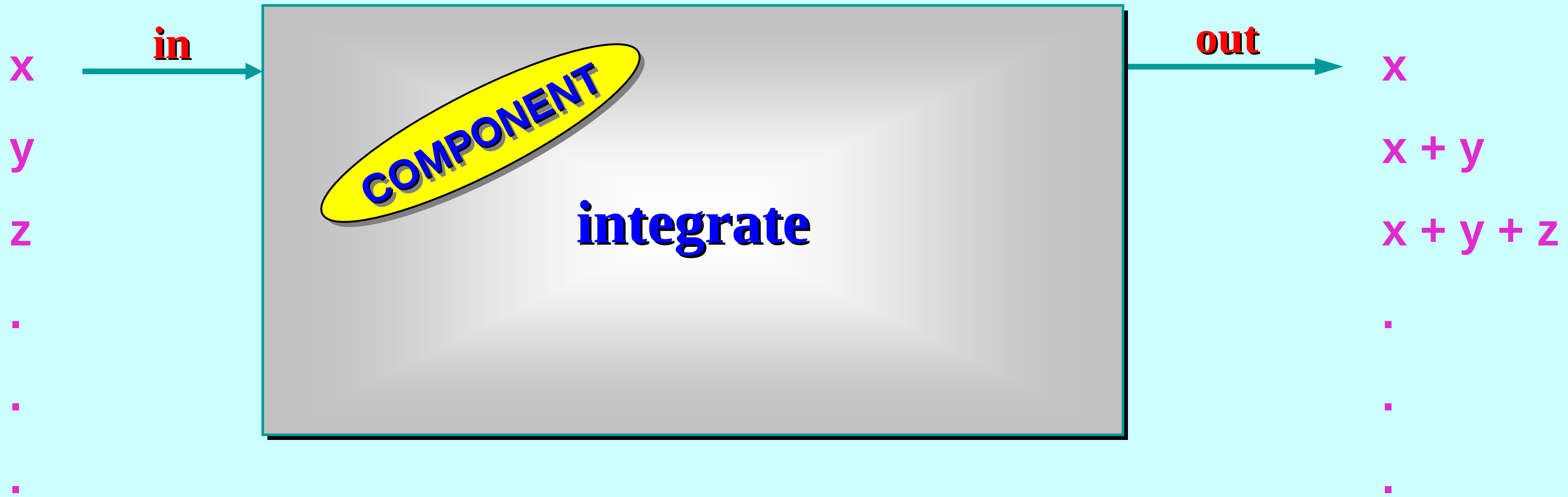
An Inertial Navigation Component



- **acc.in**: carries *regular* accelerometer samples;
- **vel.reset**: velocity *initialisation* and *corrections*;
- **pos.reset**: position *initialisation* and *corrections*;
- **pos/vel/acc**: *regular* outputs.

Build it from two components

Example – Integrator (*again*)




```
PROC integrate (CHAN INT in?, out!)
  INITIAL INT total IS 0:
  WHILE TRUE
    INT x:
    SEQ
      in ? x
      total := total + x
      out ! total
:
```

serial implementation

With an Added Kill Channel

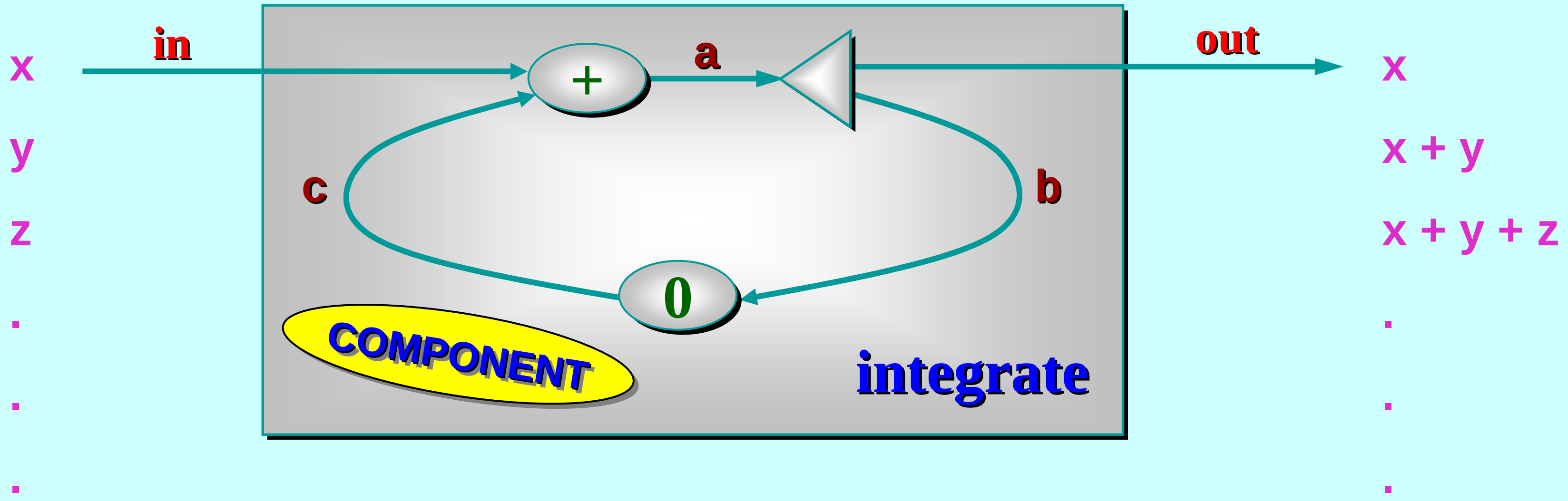

```
PROC integrate.kill (CHAN INT in?, out!, kill?)
  INITIAL INT total IS 0:
  INITIAL BOOL running IS TRUE:
  ...  main loop
:
```

With an Added Kill Channel


```
WHILE running          -- main loop
  PRI ALT
    INT any:
    kill ? any
      running := FALSE
    INT x:
    in ? x
      SEQ
        total := total + x
        out ! total
```

serial implementation

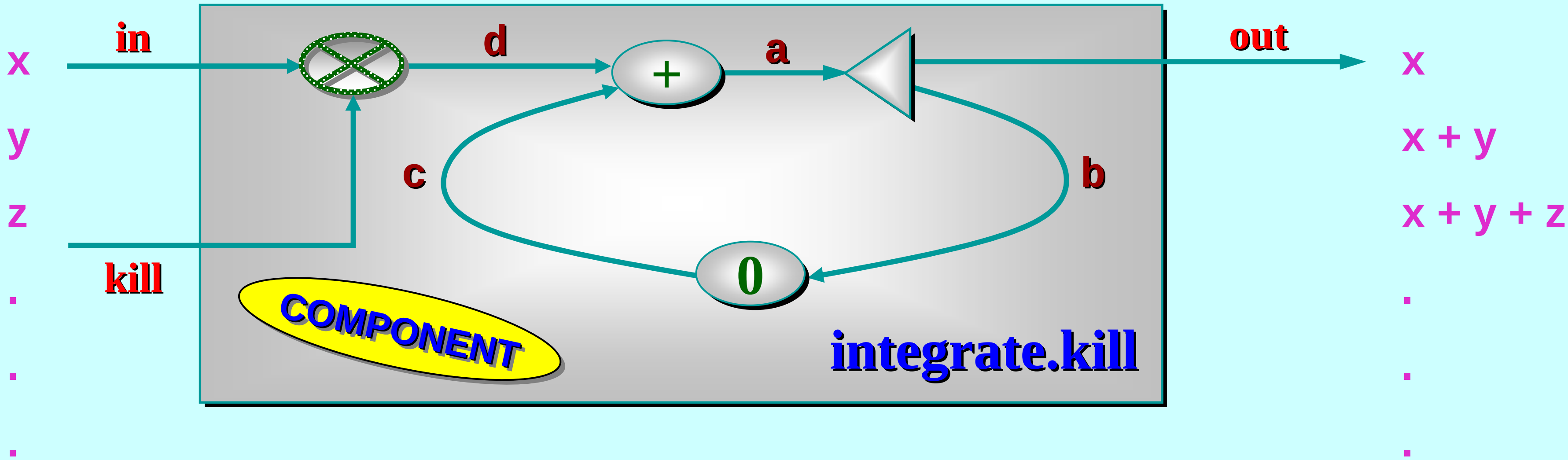
Example – Integrator (*again*)




```
PROC integrate (CHAN INT in?, out!)
  CHAN INT a, b, c:
  PAR
    plus (in?, c?, a!)
    delta (a?, out!, b!)
    prefix (0, b?, c!)
:
```

parallel implementation

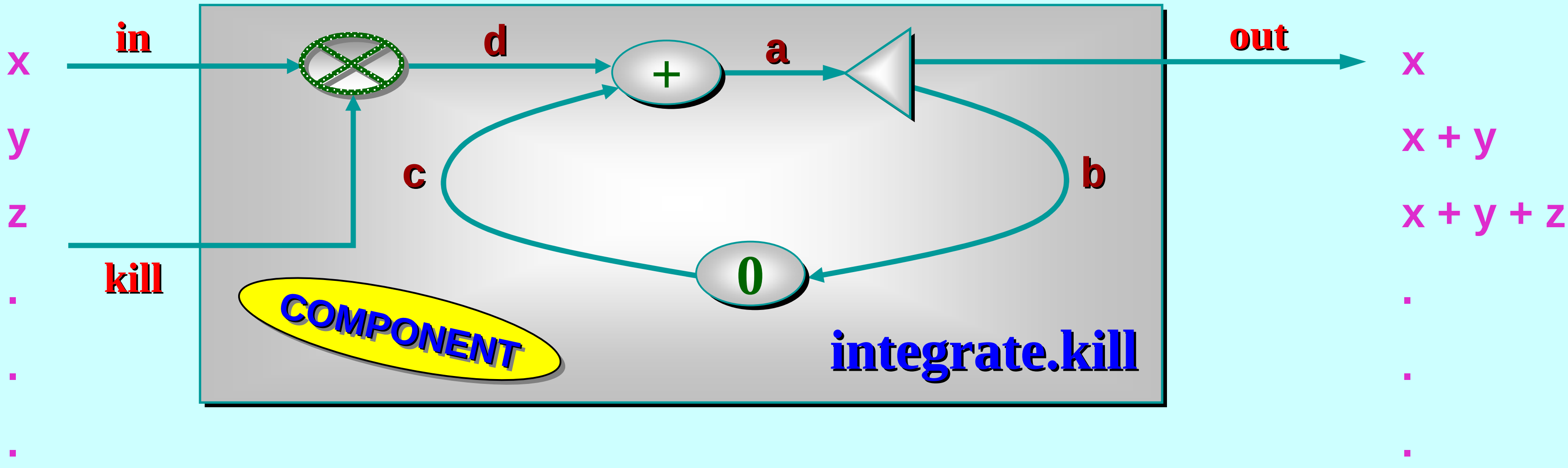
With an Added Kill Channel




```
PROC integrate.kill (CHAN INT in?, out !, kill?)
  CHAN INT a, b, c, d:
  PAR
    killer (in?, kill?, d!)
    plus (d?, c?, a!)
    delta (a?, out!, b!)
    prefix (0, b?, c!)
:
```

parallel implementation

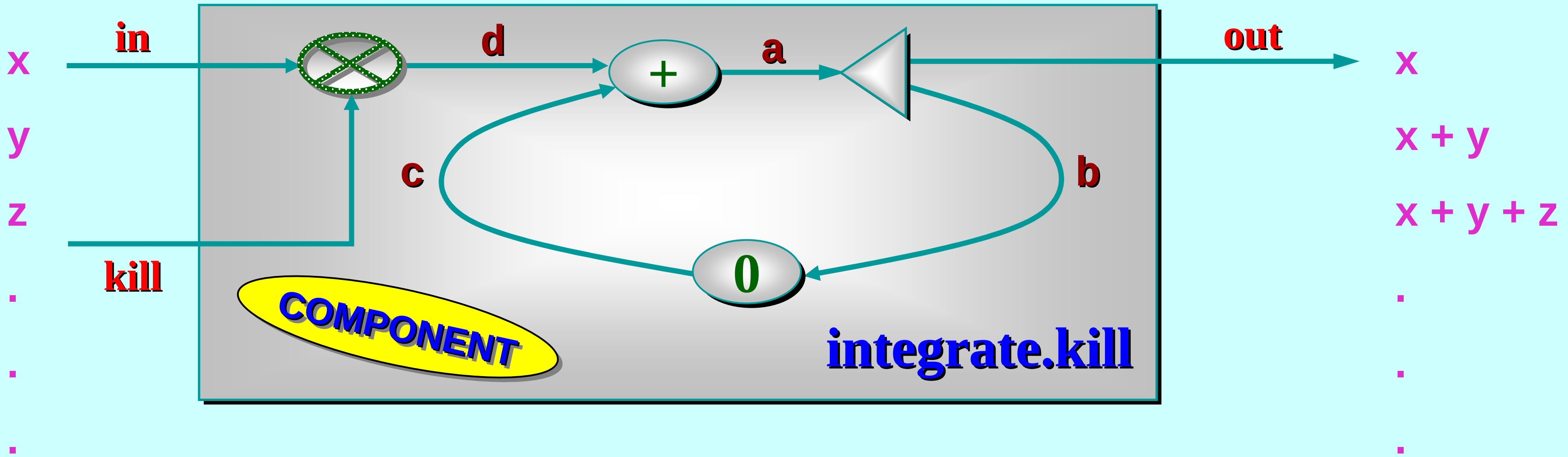
With an Added Kill Channel



To shut down a network *gracefully* (without leaving some processes stranded – i.e. deadlocked), we *poison* all the components. The poison spreads through the normal dataflow.

For **`integrate.kill`**, the **`killer`** process injects poison upon receiving a **`kill`** signal, and then shuts down.

With an Added Kill Channel

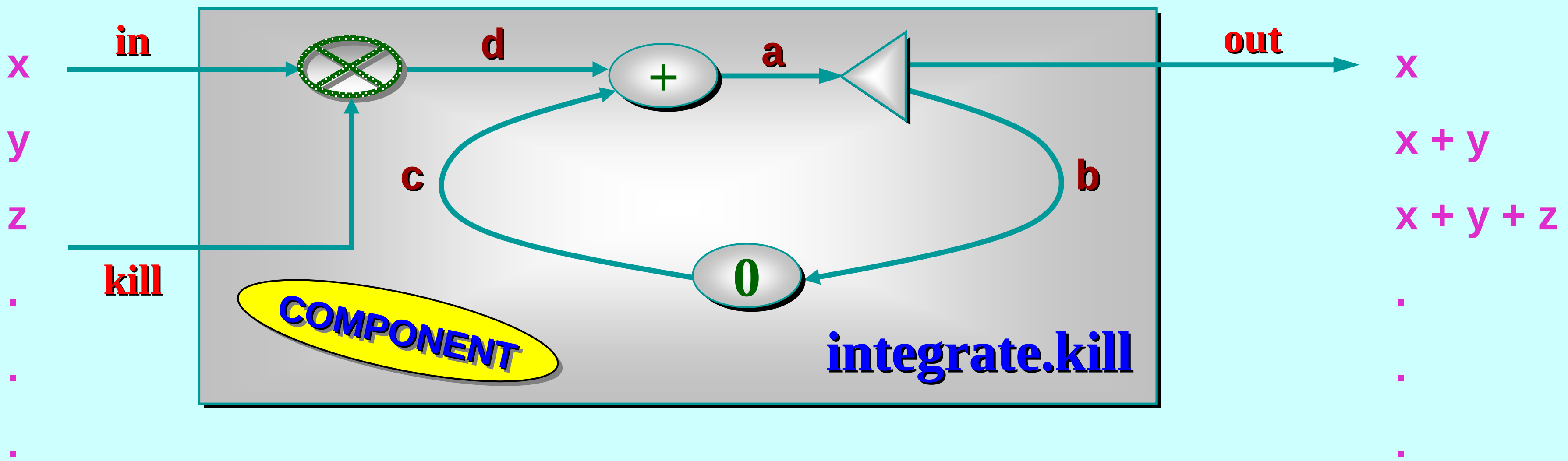


This shutdown protocol generalises to work for any process network – see the paper:

"Graceful Termination, Graceful Resetting"

… background reading

With an Added Kill Channel



The other processes check for poisonous input data – if found, they pass it on and die.

The **`plus`** process must wait for the poison to return from the feedback loop before dying.

The **`delta`** process only forwards the poison internally – unless it really wants to bring down the next component!

Choice and Non-Determinism

Non-determinism ...

The **ALT** and **PRI ALT** …

Control and real-time …

Resets and kills …

Memory cells ...

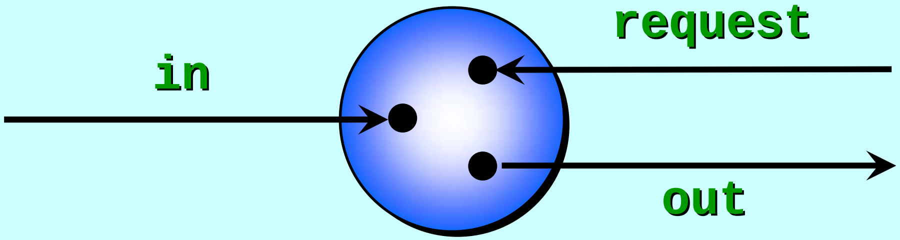
Pre-conditioned guards …

Serial FIFO *('ring')* buffer …

The replicated **ALT** …

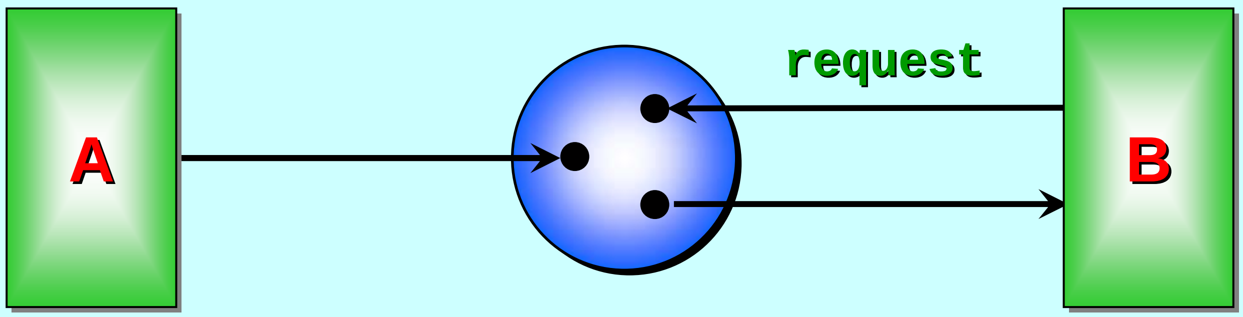
Nested **ALT**s …

A Memory Cell



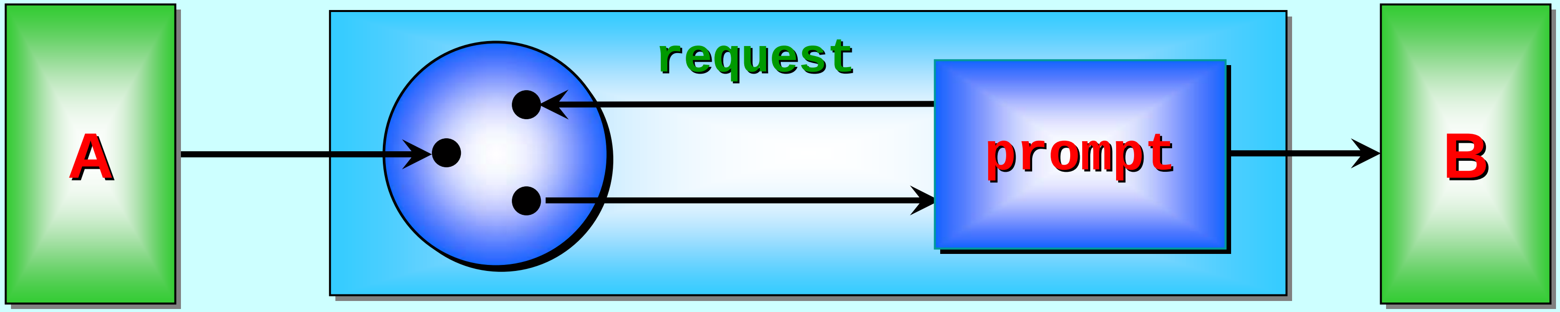

```
PROC mem.cell (CHAN INT in?,
               CHAN BOOL request?, CHAN INT out!)
  -- WARNING: write before reading!
  INT x:
  WHILE TRUE
    ALT
      in ? x
        SKIP
      BOOL any:
      request ? any
        out ! x
:
```

Asynchronous Communication



- A sends information to B.
- A can send at any time (it will never be blocked by B not being ready to receive).
- B can receive data at any time but, first, it has to **`request`** some (it will never be blocked by A not being able to send).
- The memory cell acts as a *common pool* of information.

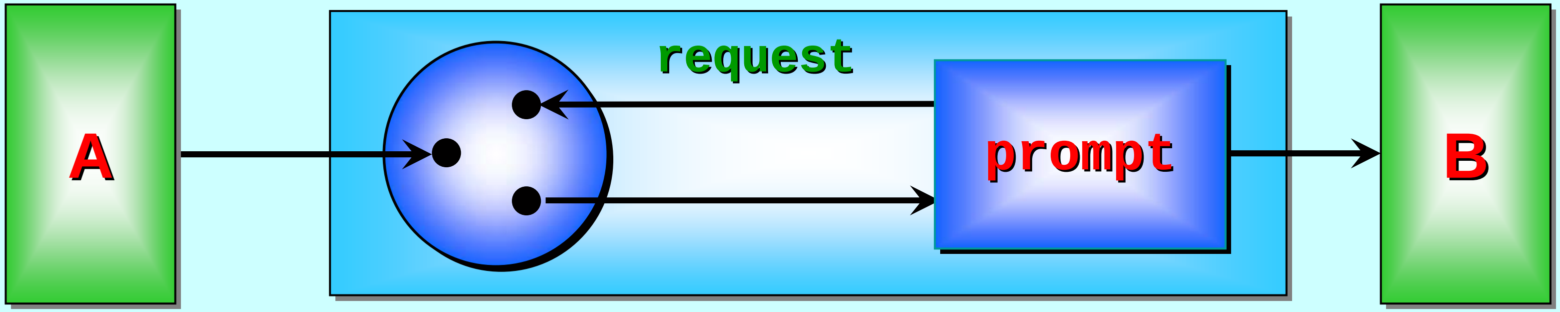
Asynchronous Communication



- We *could* relieve B from having to make requests by combining an *auto-prompter* with the memory cell.

```
PROC prompt (CHAN BOOL request!, CHAN INT in?, out!)
  WHILE TRUE
    INT x:
    SEQ
      request ! TRUE
      in ? x
      out ! x
:
```

Asynchronous Communication

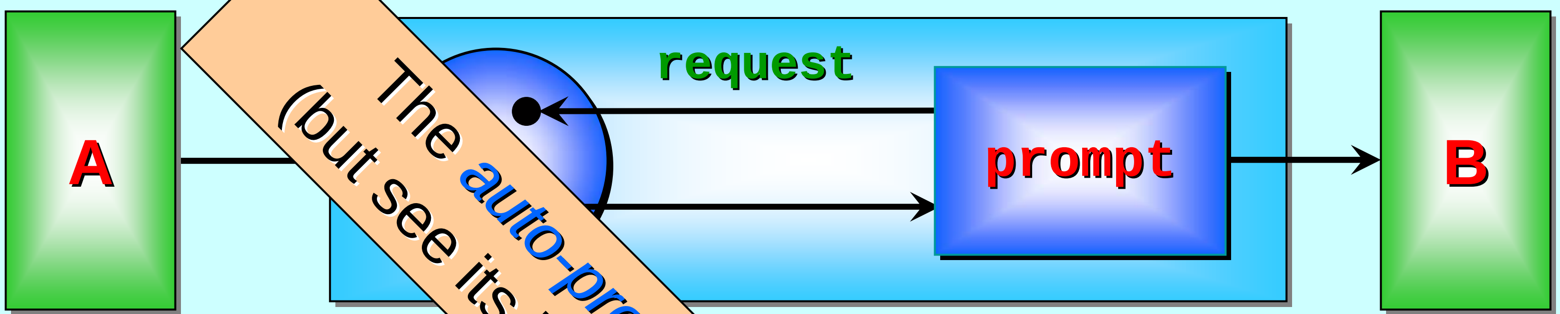


- We *could* relieve B from having to make requests by combining an *auto-prompter* with the memory cell.

- But if *auto-prompter* gets its first **`request`** in before A sends anything, it will pick up garbage from the cell.

- Also, if B is not taking data, *auto-prompter* stores old *(stale)* data, while the *memory-cell* holds anything new that arrives. *This is probably a bad thing*. When B takes data, it wants the *latest* item that A has sent.

Asynchronous Communication



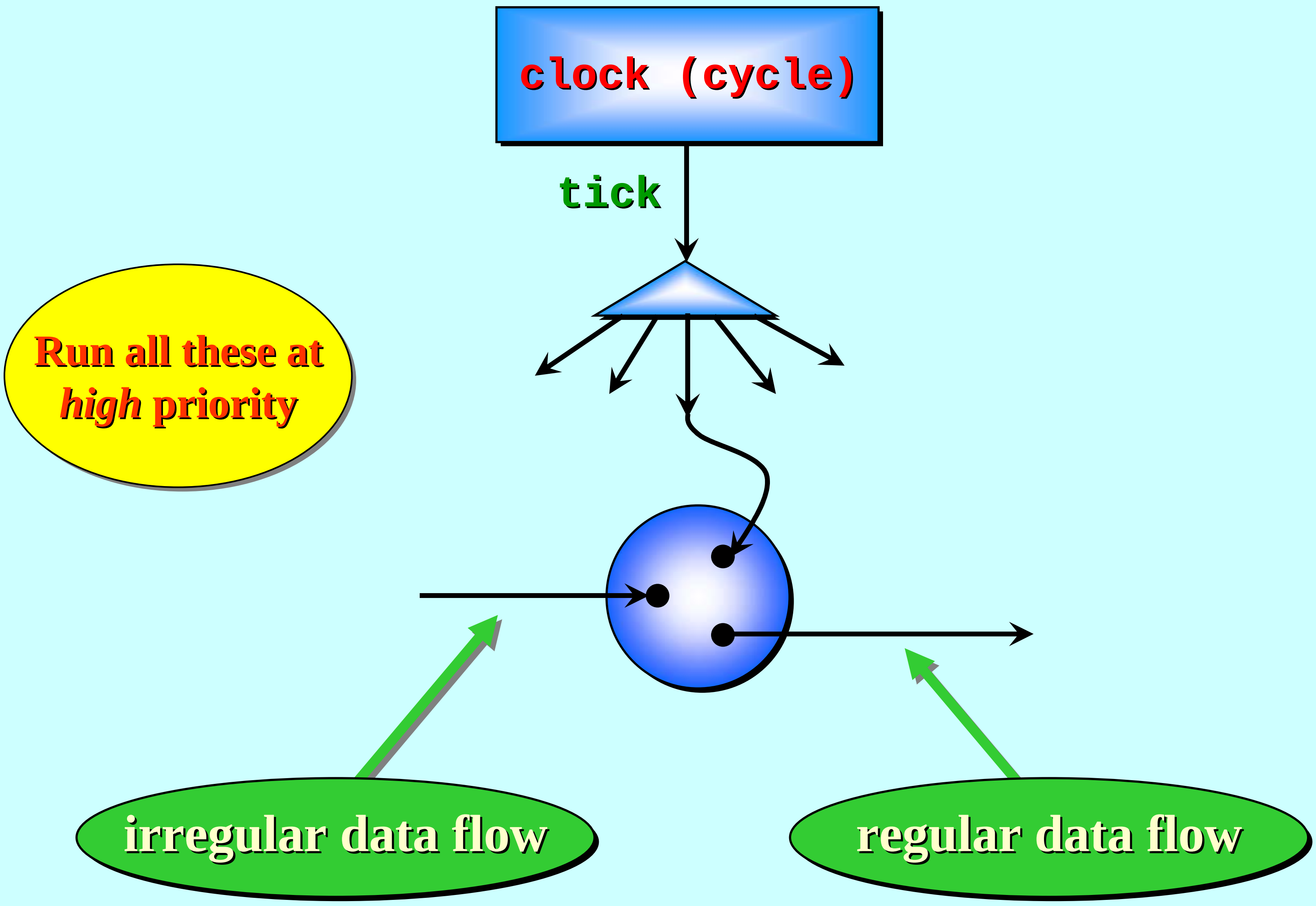
The *auto-prompter* not a good idea here …
(but see its later use the *blocking-FIFO* buffer)

- We *could* relieve B from … make requests by combining an *auto-prompter* with the me…
- But if *auto-prompter* gets its first … in before A sends anything, it will pick up garbage from …
- Also, if B is not taking data, *auto-prompte*… ld *(stale)* data, while the *memory-cell* holds anything new t… *This is probably a bad thing*. When B takes data, it w… *test* item that A has sent.

Regular Events


```
PROC clock (VAL INT cycle, CHAN BOOL tick!)
  TIMER tim:
  INT t:
  SEQ
    tim ? t
    WHILE TRUE
      SEQ
        t := t PLUS cycle
        tim ? AFTER t
        tick ! TRUE
:
```

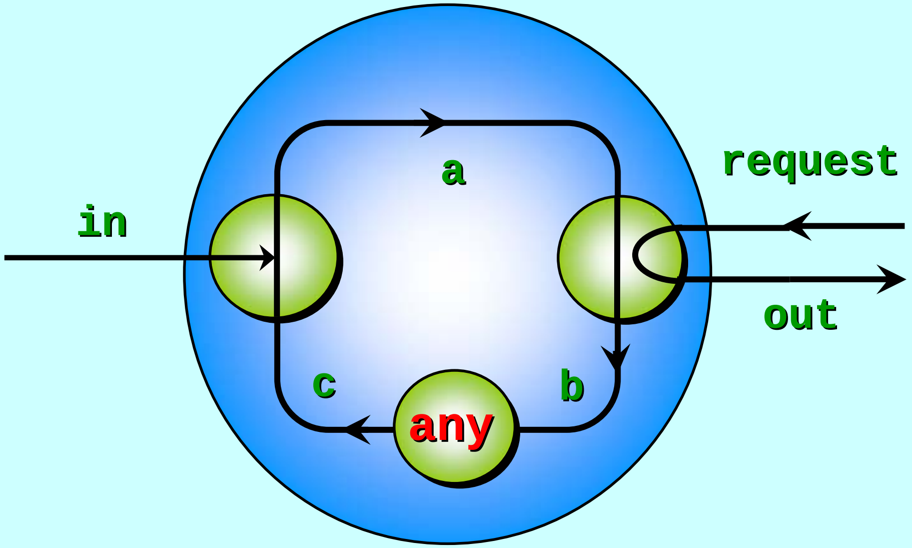
Run this at *high* priority!!


clock (cycle)
tick
Run all these at *high* priority
irregular data flow
regular data flow

Another Memory Cell

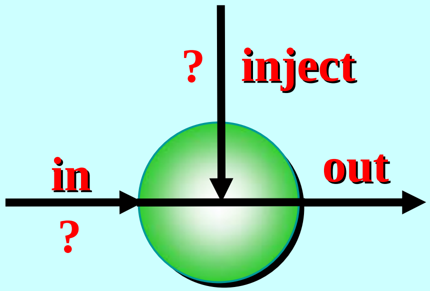
- The implementation of **`mem.cell`** captured state information *(the memory)* with a variable. This is OK for the demonstrated application *(asynchronous communication)* … but a bit of a cheat if we want to *model* a variable.

- The following implementation retains state information just by the topology *(feedback loops)* of the internal connections. The internal components do not themselves retain state. They give a design for hardware implementation.



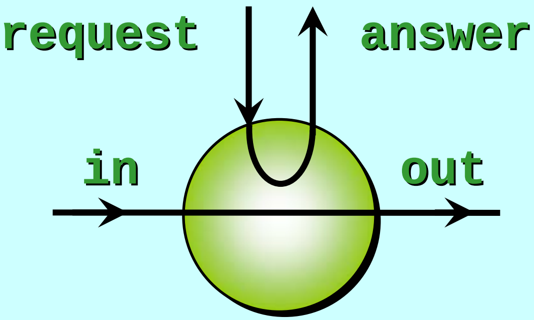

```
PROC mem.cell (CHAN INT in?,
               CHAN BOOL request?, CHAN INT out!)
  -- WARNING: write before reading!!!
  CHAN INT a, b, c:
  PAR
    replace (c?, a!, in?)
    sample (a?, b!, request?, out!)
    INT any:
    prefix (any, b?, c!)
:
```

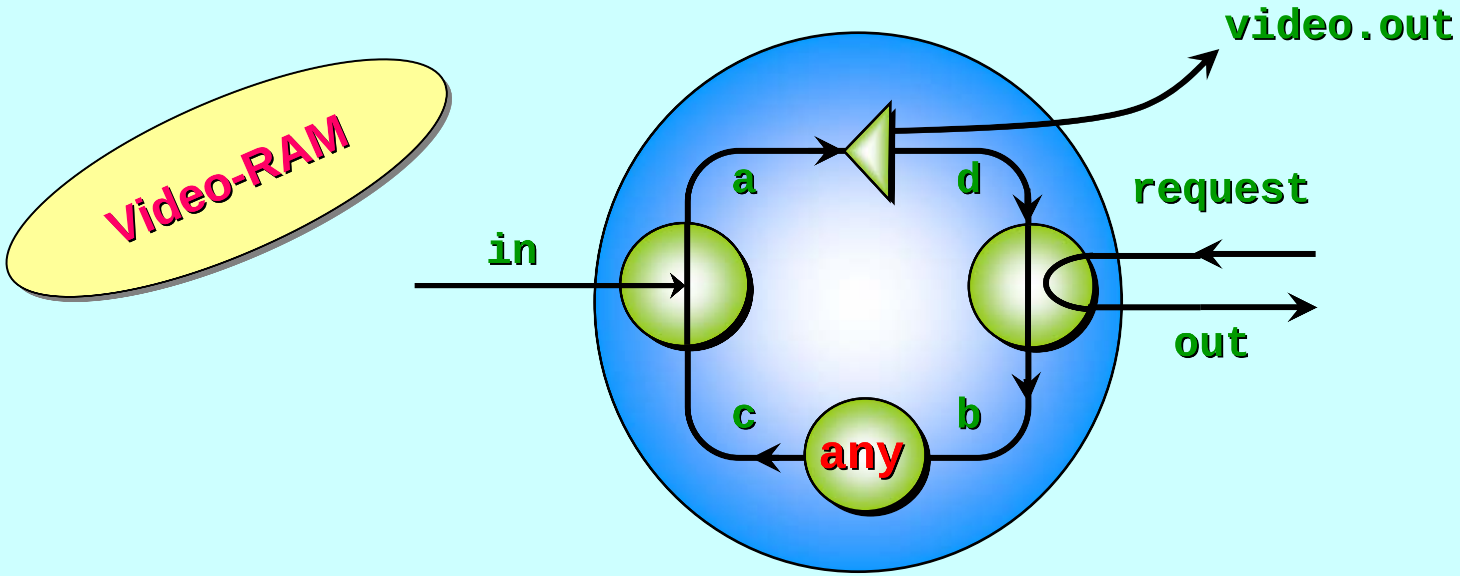
Seen before




```
PROC replace (CHAN INT in?, out!, inject?)
  WHILE TRUE
    PRI ALT
      INT x, any:
      inject ? x         -- replace the
        PAR              -- next ‘in’
          in ? any       -- with the
          out ! x        -- ‘inject’ value
      INT x:
      in ? x             -- normally
        out ! x          -- just copy through
:
```

```
PROC sample (CHAN INT in?, out!,
             CHAN BOOL request?, CHAN INT answer!)
  WHILE TRUE
    PRI ALT
      BOOL any:
      request ? any
        INT x:
        SEQ
          in ? x
          PAR
            answer ! x     -- duplicate
            out ! x        -- output
       INT x:
       in ? x              -- normally
         out ! x           -- just copy through
:
```






```
PROC vid.cell (CHAN INT in?, CHAN INT video.out!,
               CHAN BOOL request?, CHAN INT out!)
  -- WARNING: write before reading or viewing!!!
  CHAN INT a, b, c, d:
  PAR
    replace (c?, a!, in?)
    delta (a?, video.out!, d!)
    sample (d?, b!, request?, out!)
    INT any:
    prefix (any, b?, c!)
:
```

Choice and Non-Determinism

Non-determinism ...

The **ALT** and **PRI ALT** …

Control and real-time …

Resets and kills …

Memory cells ...

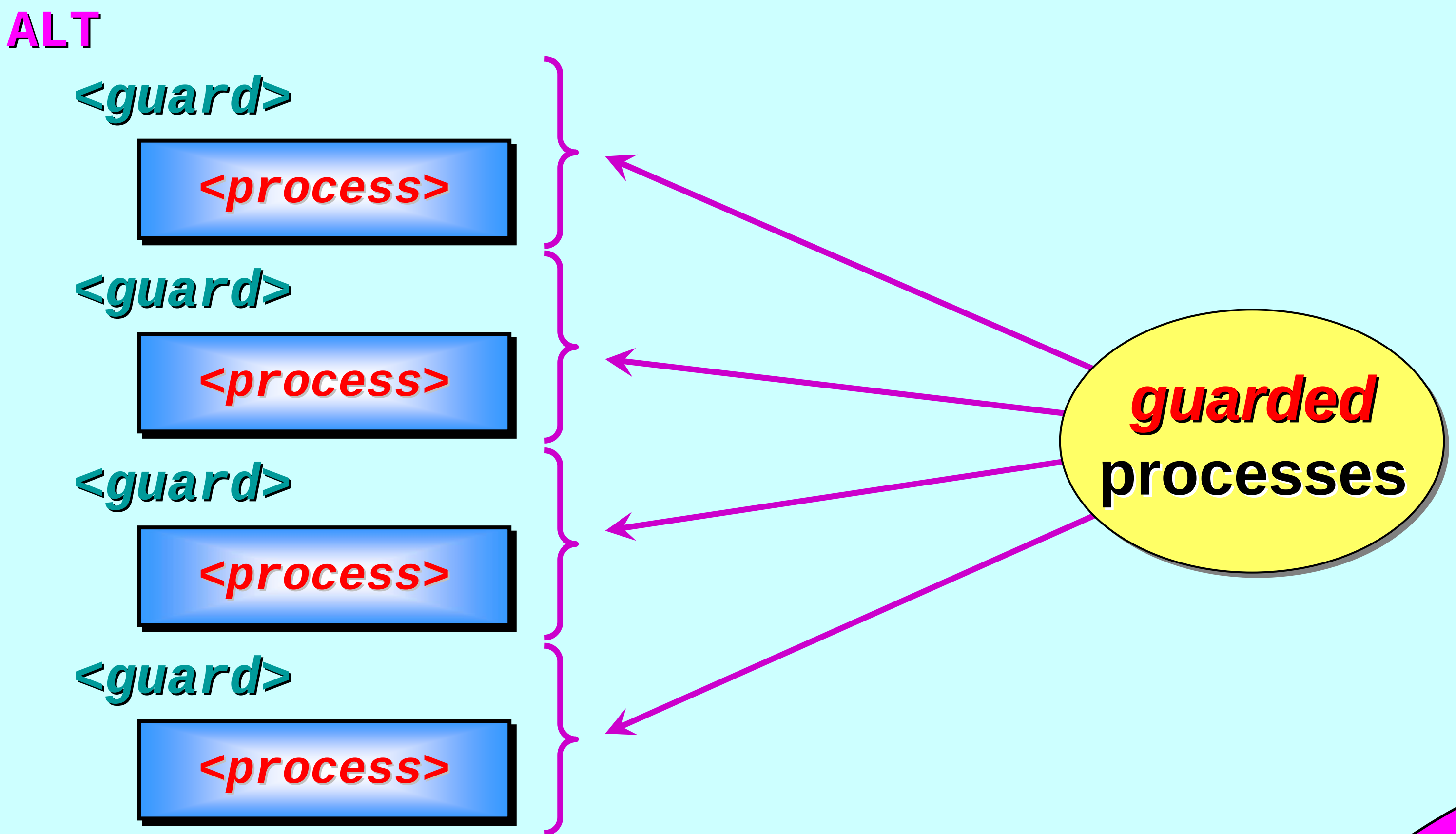
Pre-conditioned guards …

Serial FIFO *('ring')* buffer …

The replicated **ALT** …

Nested **ALT**s …

Non-Deterministic Choice



Revision :

Non-Deterministic Choice

An **ALT** process executes as follows:

- if no guard is ready, the process is suspended until one, or more, become ready;
- if one guard is ready, execute it and then execute the process it was defending *(end of* ***ALT*** *process)*;
- if more than one guard is ready, one is ***arbitrarily chosen*** and executes, followed by the process it was defending *(end of* ***ALT*** *process)*.

Note: only *one* of the guarded processes is executed.

Revision :

Deterministic Choice

A **PRI ALT** process executes as follows:

- if no guard is ready, the process is suspended until one, or more, become ready;

- if one guard is ready, execute it and then execute the process it was defending *(end of **PRI ALT** process)*;

- if more than one guard is ready, ***the first one listed is chosen*** and executes, followed by the process it was defending *(end of **PRI ALT** process)*.

Note: only *one* of the guarded processes is executed.

Revision :

Pre-Conditioned Guards

Any guard may be prefixed by a **BOOL** *pre-condition*:

<pre-condition> & <guard>

<process>

When the **ALT** (or **PRI ALT**) starts execution, any *pre-conditions* on the guards are evaluated.

If a *pre-condition* turns out to be **FALSE**, *that guarded process is not chosen for execution* – even if the guard is (or becomes) ready.

Pre-Conditioned Guards

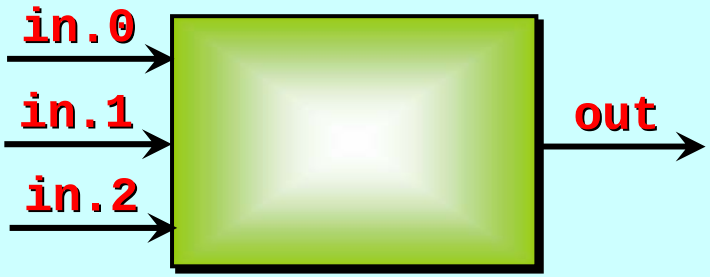
Any guard may be prefixed by a **BOOL** *pre-condition*:

```
<pre-condition> & <guard>
    <process>
```

For each execution of an **ALT** (or **PRI ALT**), any *pre-conditions* only need evaluating *once* – no rechecks are necessary.

A *pre-condition* is a **BOOL** expression, *whose variables cannot change* whilst waiting for a guard to become ready. No other process can change those variables *(simply because this process is observing them)*.

```
INT a:
BOOL timing:
SEQ
  ...  set a and timing
  TIMER tim:
  INT time.out, x:
  SEQ
    tim ? time.out
    time.out := time.out PLUS 1000
    ALT
      in.0 ? x
        out ! x
      in.1 ? x
        out ! x
      (a = 42) & in.2 ? x
        out ! x
      timing & tim ? AFTER time.out
        out ! –1
```



RUN-TIME DECISION:

listen out for the `in.2` channel?

set the timeout?

Choice and Non-Determinism

Non-determinism ...

The **ALT** and **PRI ALT** …

Control and real-time …

Resets and kills …

Memory cells ...

Pre-conditioned guards …

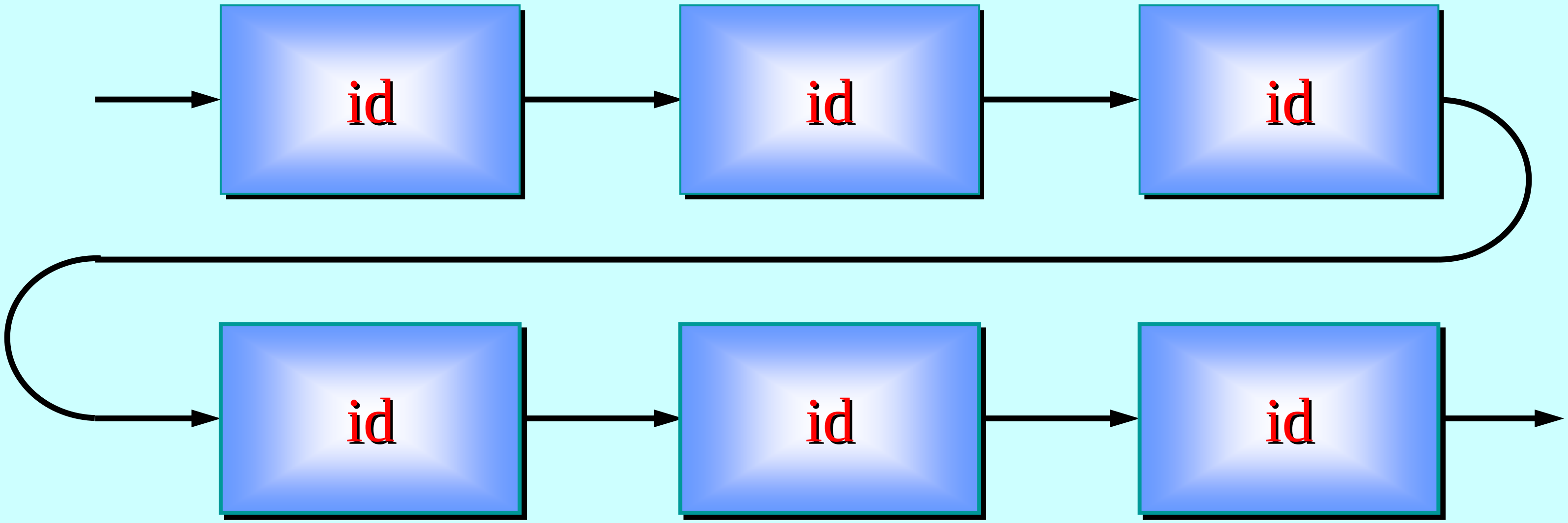
Serial FIFO *(‘ring’)* buffer …

The replicated **ALT** …

Nested **ALT**s …

Another (*FIFO*) Buffer Process

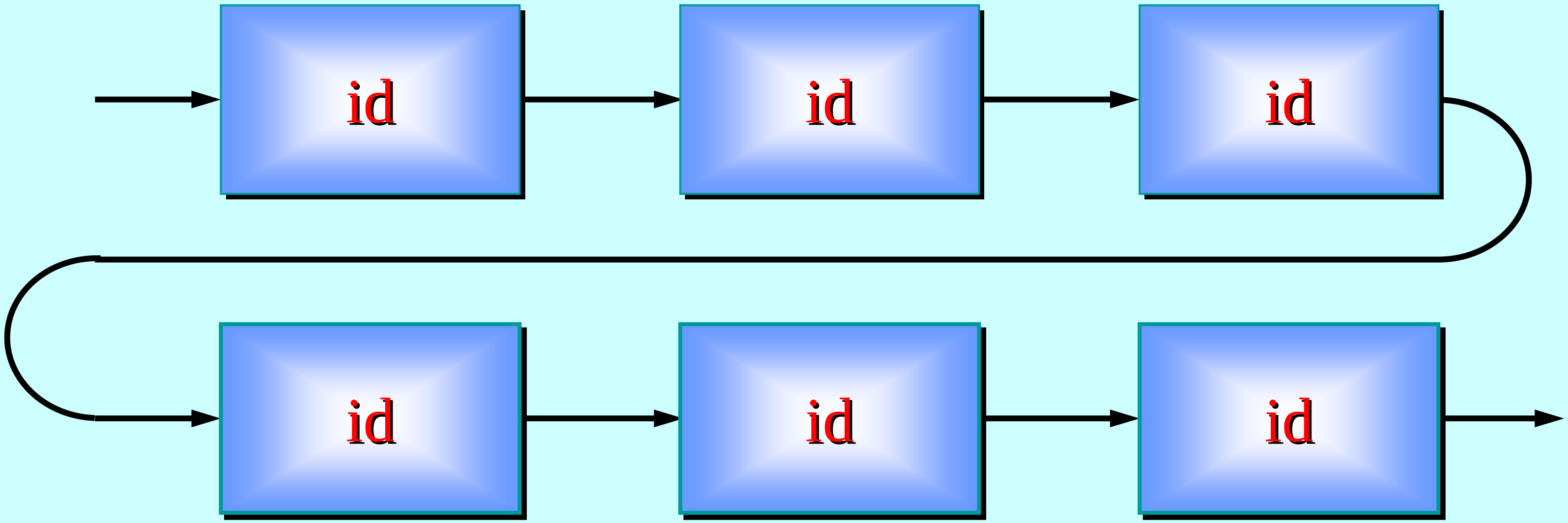
Recall that …



is a blocking ***FIFO*** buffer of capacity 6

Another *(FIFO)* Buffer Process

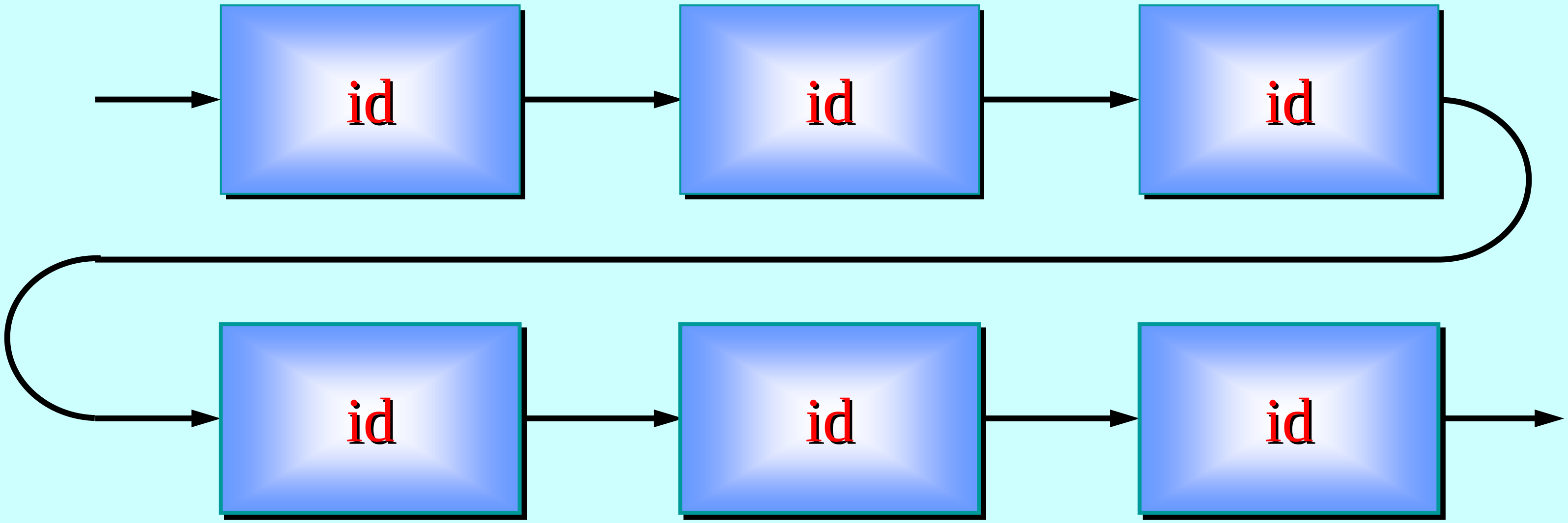
This is a great and simple design … *for hardware* …



… where buffered data can flow *in parallel* along the pipeline …

Another (*FIFO*) Buffer Process

This is a great and simple design … *for hardware* …



… but not so good *for software* … where each item of buffered data must be copied (from process to process) **N** times (where **N** is the size of the buffer).

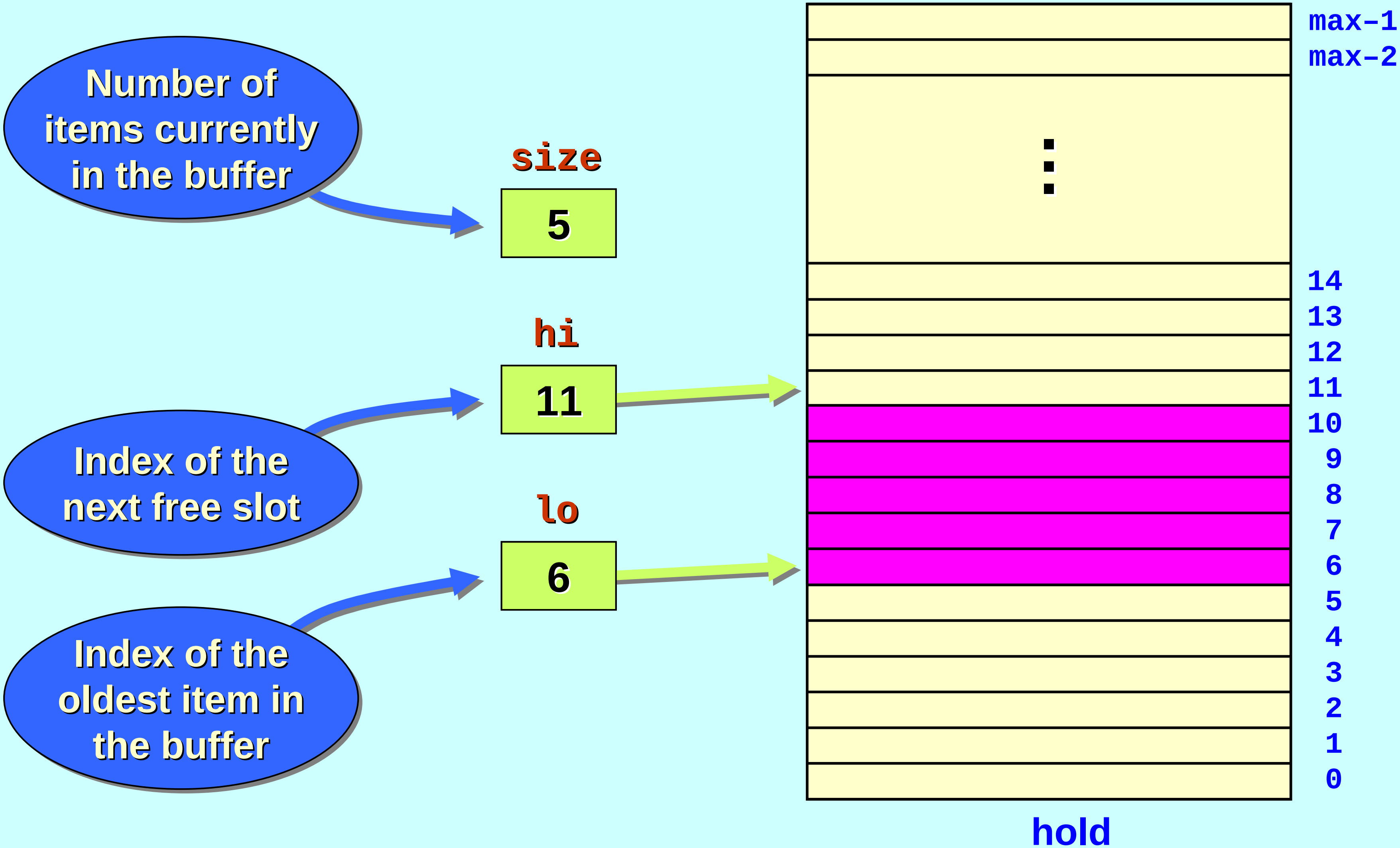
Another *(FIFO)* Buffer Process

So let's do something better suited for *software* … that does not do all that copying. Let's just have *one* process.

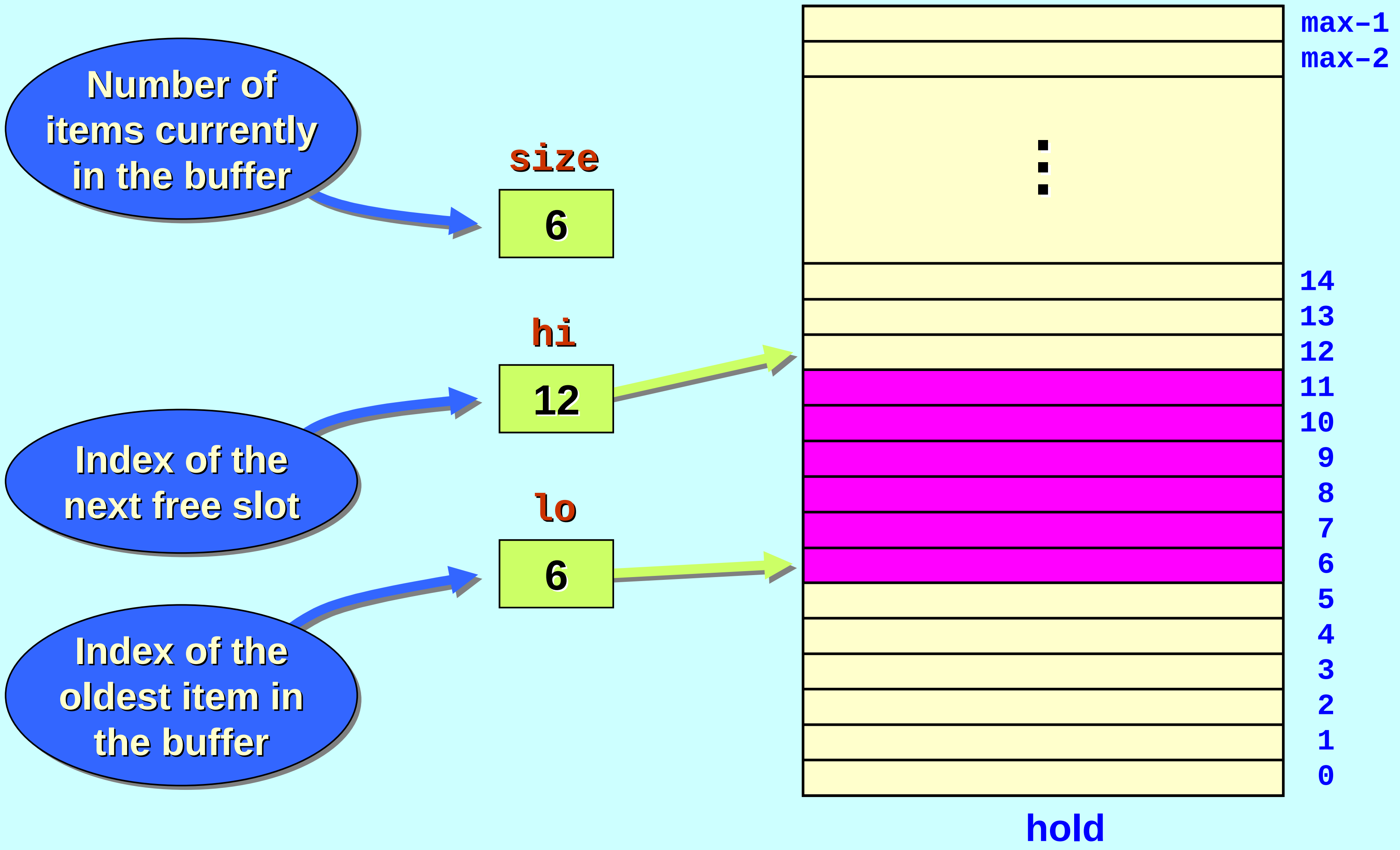
`buffer` has a capacity of **`max`** *(say)*. A process may send data into the buffer until it is *full*. If it then tries to send more, it will be blocked until the buffer gets emptier.

A process may extract data *(by first making a* **`request`***)* until the **`buffer`** is empty. If it then requests more, it will be blocked until the **`buffer`** gets some data.

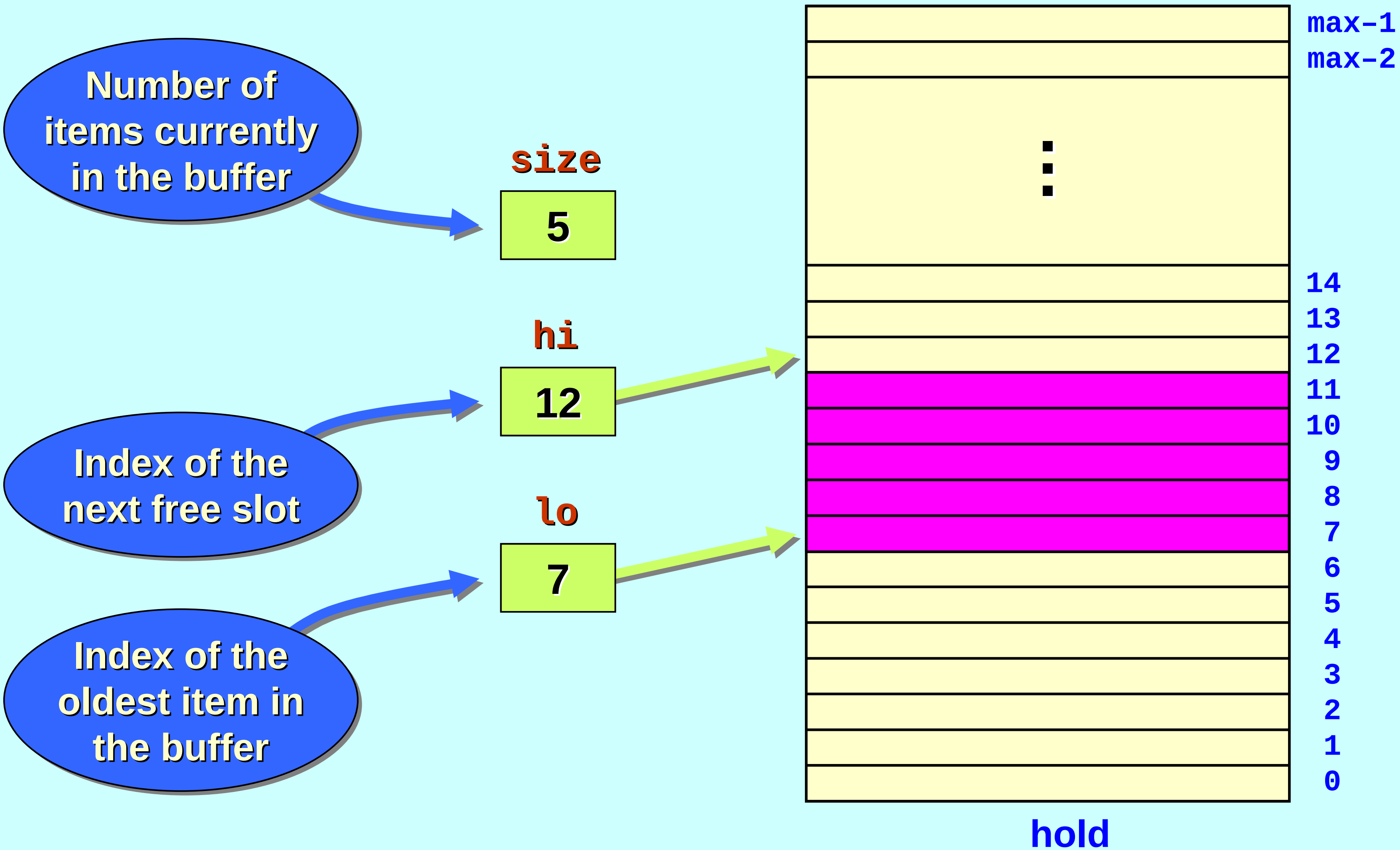
Within **buffer**, we declare an array (to **hold** up to **max** items) and *three control variables*:



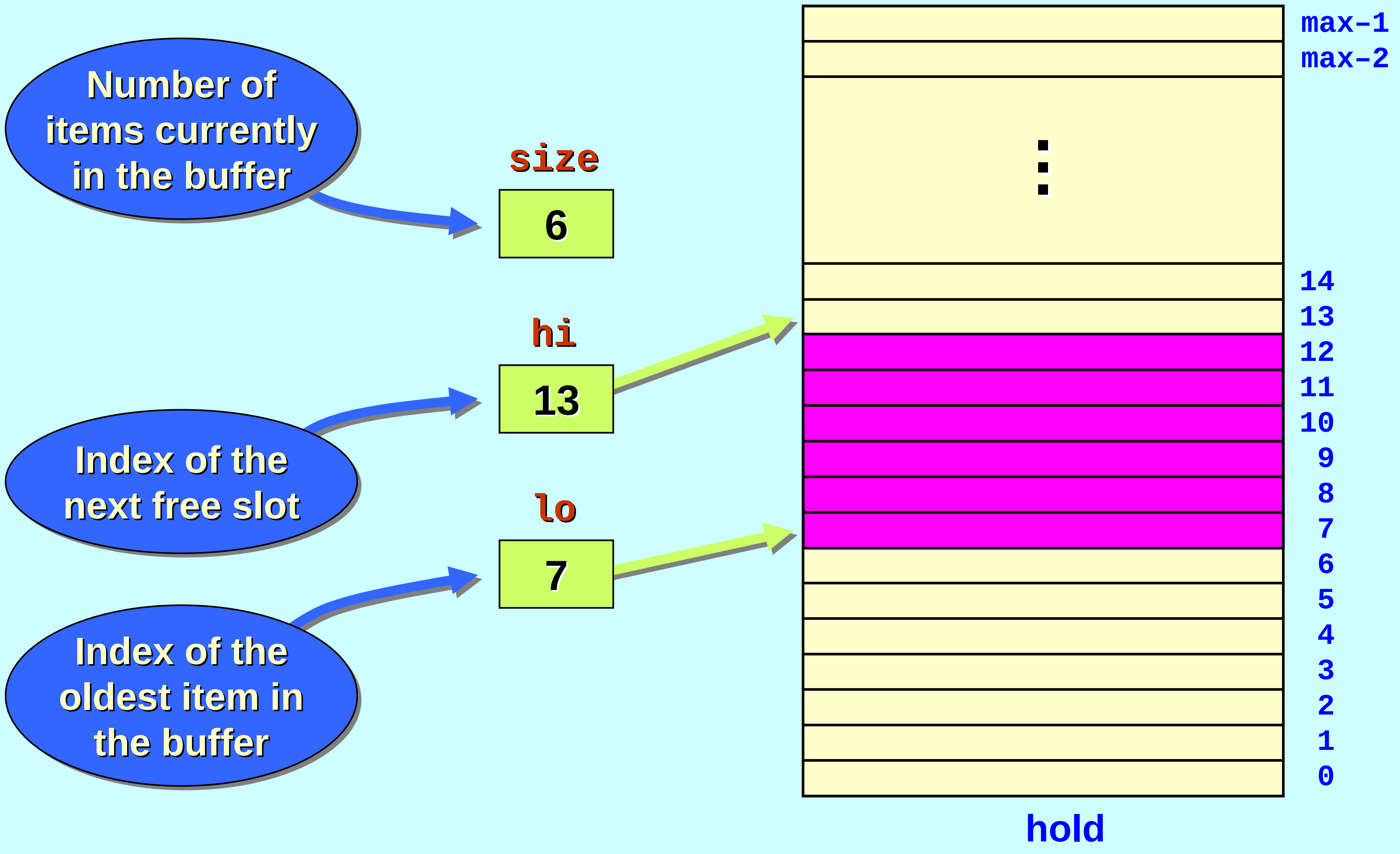
If **buffer** receives another item:



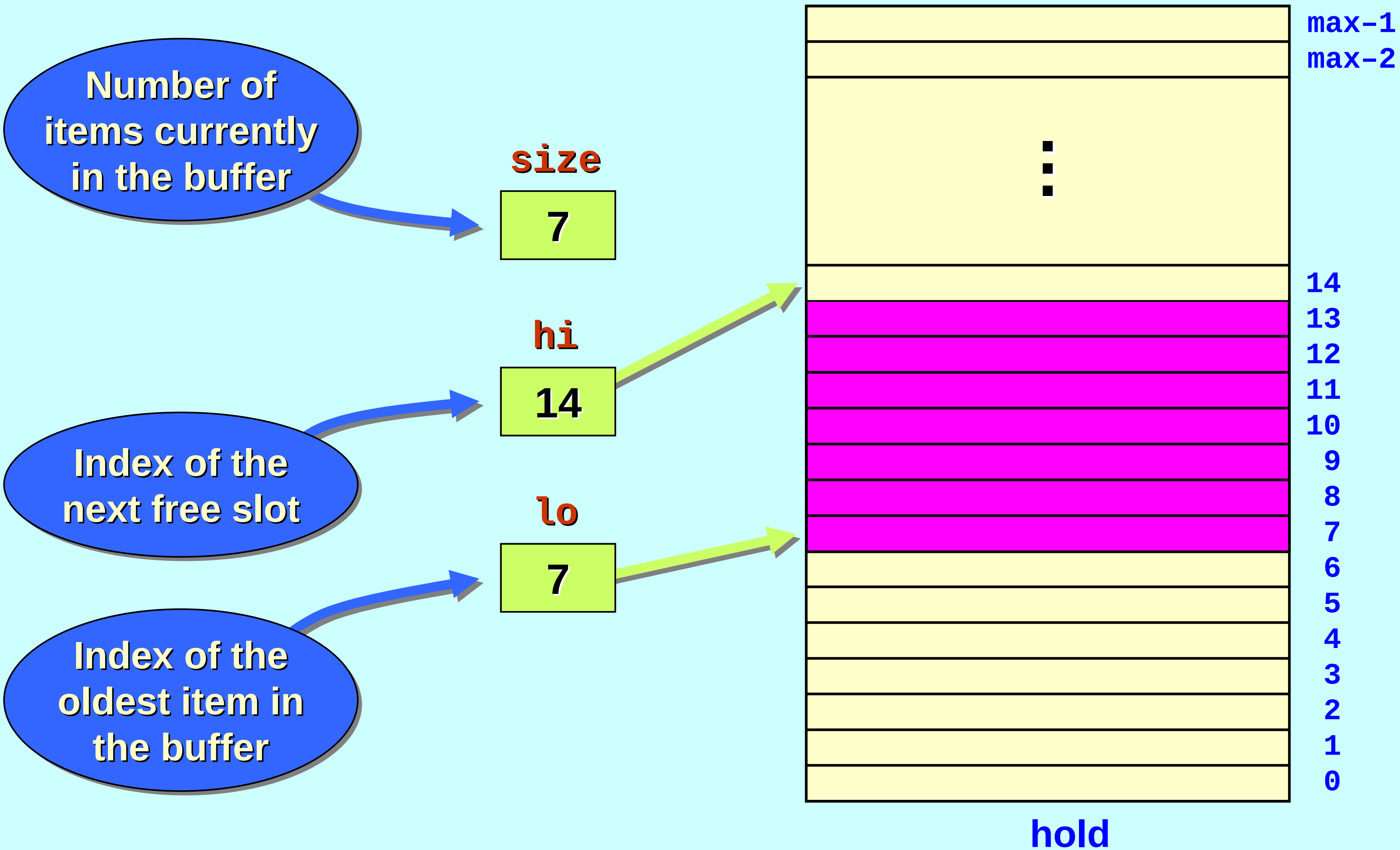
And, then, is requested for and delivers an item:



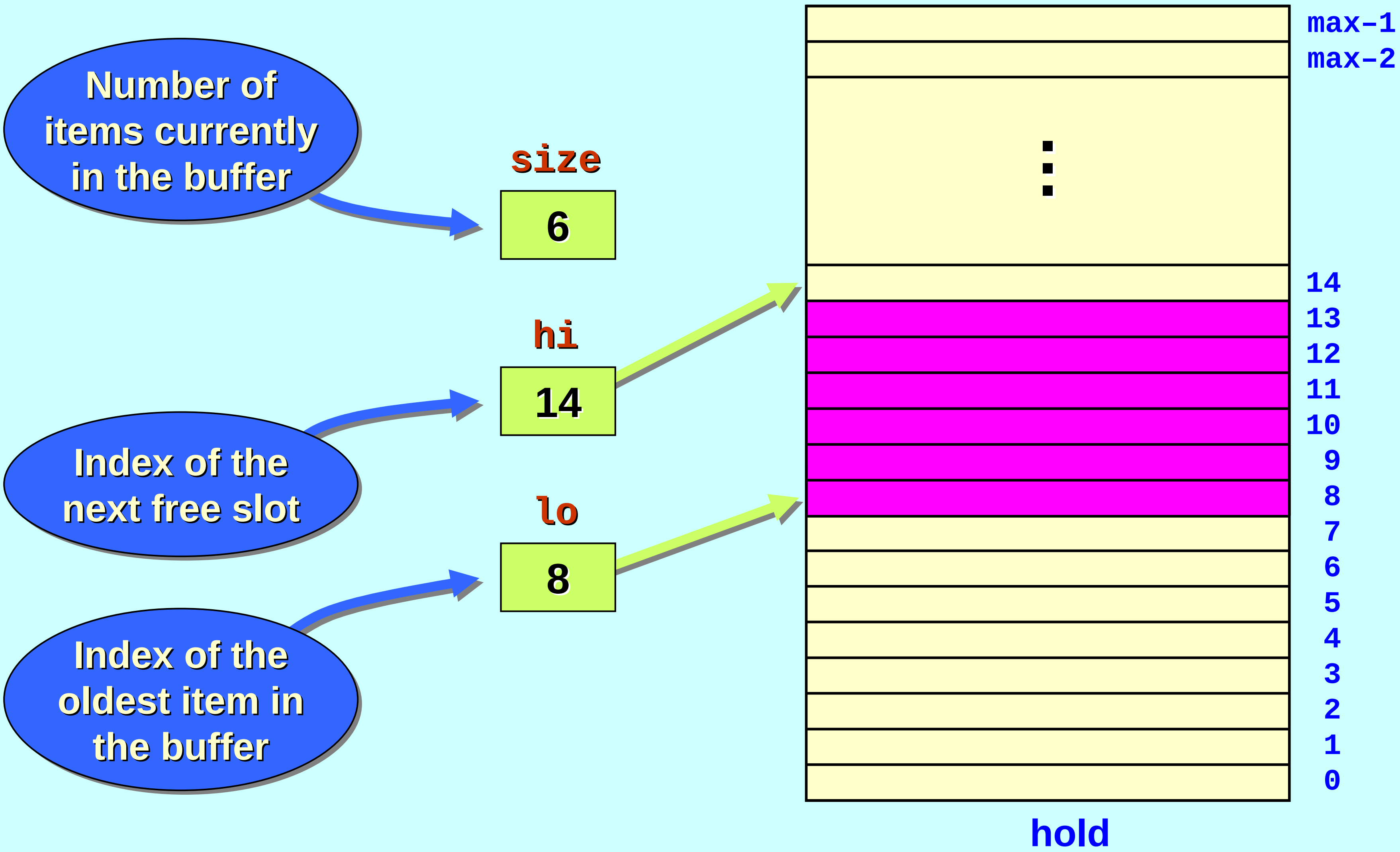
And, then, receives another item :



And another item :



And, then, is requested for and delivers an item:

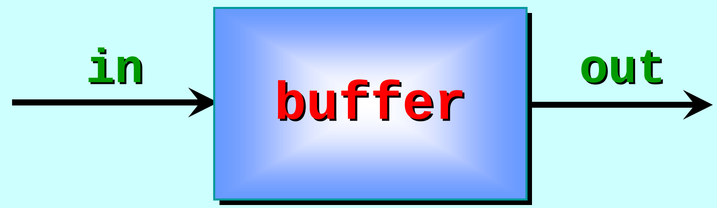



```
PROC buffer (CHAN INT in?, CHAN BOOL request?, CHAN INT out!)
  [max]INT hold:
  INT lo, hi, size :    -- size = hi – lo (modulo wrap-around)
  SEQ
    lo, hi, size := 0, 0, 0
    WHILE TRUE
      ALT
        (size < max) & in ? hold[hi]
          SEQ
            hi := (hi + 1)\max
            size := size + 1
        BOOL any:
        (size > 0) & request ? any
          SEQ
            out ! hold[lo]
            lo := (lo + 1)\max
            size := size – 1
:
```

index wrap-around


```
PROC buffer (CHAN INT in?, CHAN BOOL request?, CHAN INT out!)
  [max]INT hold:
  INT lo, hi, size :    -- size = hi – lo (modulo wrap-around)
  SEQ
    lo, hi, size := 0, 0, 0
    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
:
```

Note: the process taking items from this buffer has to make a request … *because output guards are not supported* … despite their semantic power.

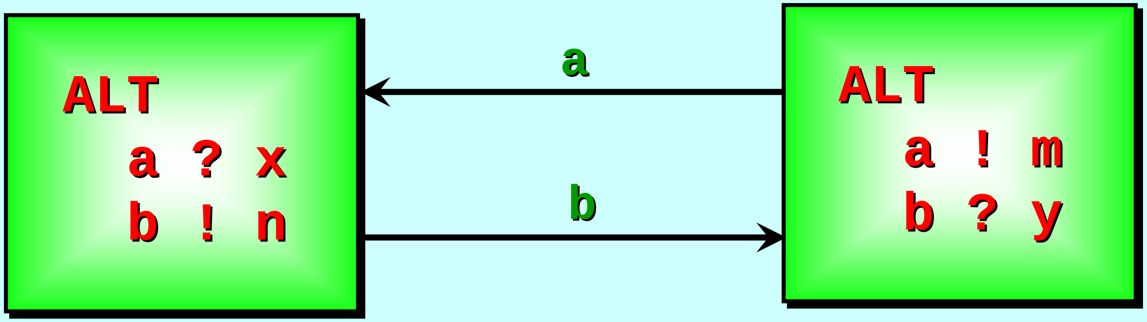



```
PROC buffer (CHAN INT in?, CHAN INT out!)
  [max]INT hold:
  INT lo, hi, size :    -- size = hi - lo (modulo wrap-around)
  SEQ
    lo, hi, size := 0, 0, 0
    WHILE TRUE
      ALT
        (size < max) & in ? hold[hi]
          SEQ
            hi := (hi + 1)\max
            size := size + 1
        (size > 0) & out ! hold[lo]
          SEQ
            lo := (lo + 1)\max
            size := size - 1
:
```

Note: the processes taking items from this buffer have to make a request because *output guards are not supported* ... despite their semantic power.

☹ This is not allowed ☹

Output guards require an independent mediator to resolve choices – because more than one process *must make the same choice*. For example:

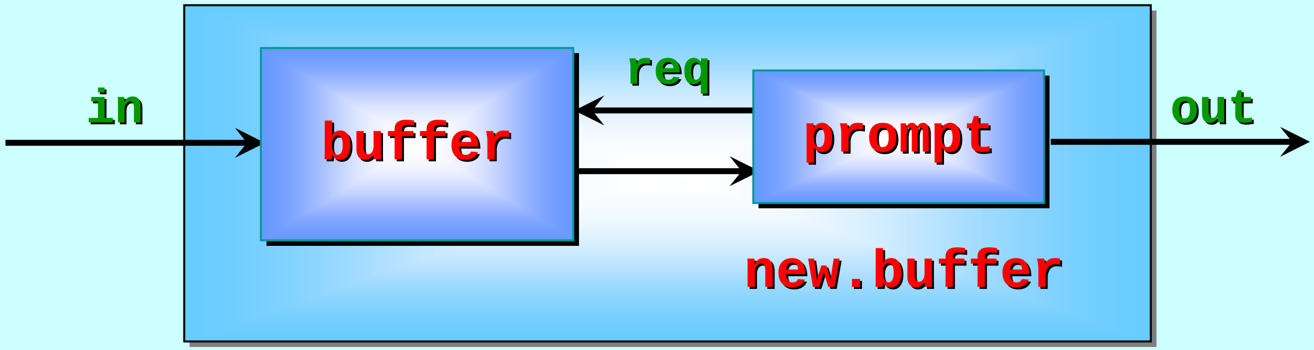


Which communication should be done? Either is allowed. Both processes must reach the same decision.

We know how to solve this … *but it costs!*

By only allowing input guards, only one process is ever involved in any choice (i.e. if one process is **ALT**ing, no process communicating with it can be **ALT**ing).

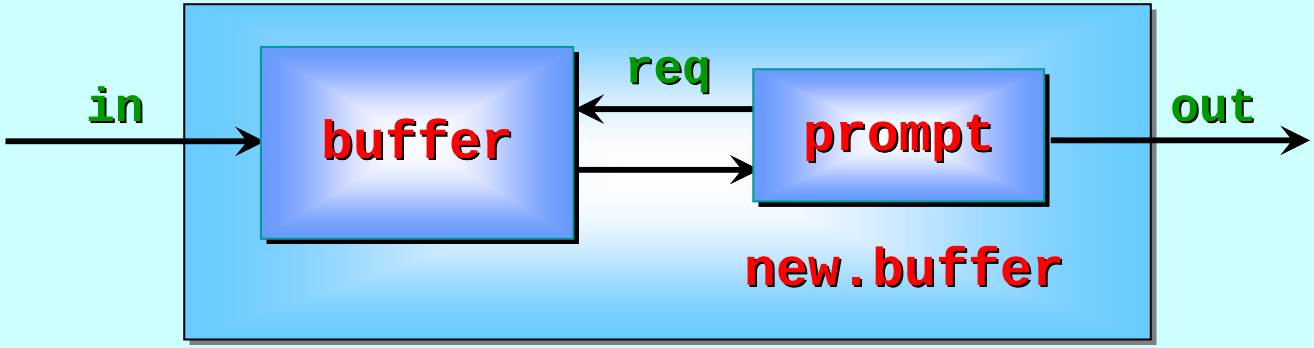
To relieve the receiving process from the bother of making the requests, we can install an *auto-prompter* alongside the **buffer**:




```
PROC prompt (CHAN BOOL request!, CHAN INT in?, out!)
  WHILE TRUE
    INT x:
    SEQ
      request ! TRUE
      in ? x
      out ! x
:
```

seen before

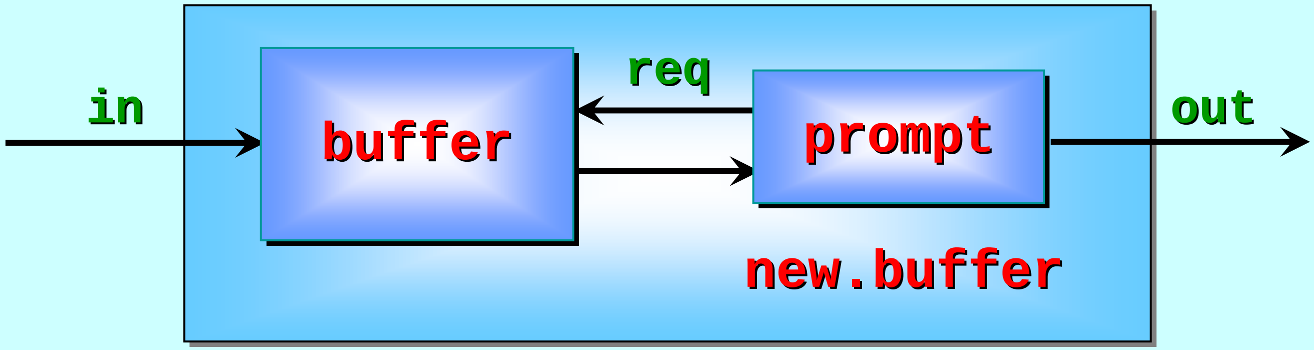
To relieve the receiving process from the bother of making the requests, we can install an *auto-prompter* alongside the **buffer**:



Just as when used like this with the **mem.cell** process, **prompt** holds old *(stale)* data. Meanwhile, the **buffer** holds anything new that arrives. *This is a good thing this time!*

Whatever takes data from **new.buffer** wants the *oldest* item put into it – it is, after all, a *FIFO*. ☺ ☺ ☺

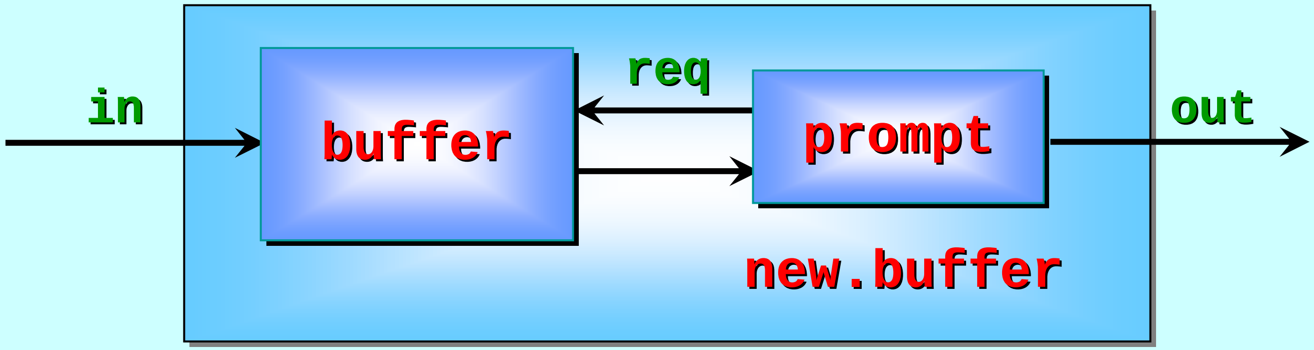
To relieve the receiving process from the bother of making the requests, we can install an *auto-prompter* alongside the **buffer**:



The **prompt** process will be blocked making its first **request** until something is put into the **buffer**.

It then extracts that item and offers it **out**. When (if) that is taken, **prompt** again requests from **buffer**, which *may* or *may not* have accumulated more items.

To relieve the receiving process from the bother of making the requests, we can install an *auto-prompter* alongside the **buffer**:

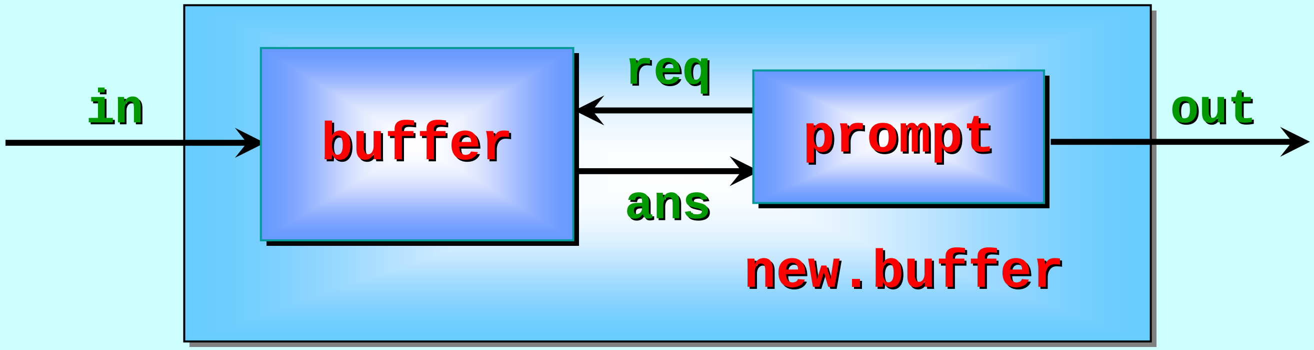


An empty **buffer** always blocks a **request** from **prompt**, leaving **new.buffer** not trying to **out** anything.

An non-empty **buffer** always gives **prompt** its *oldest* item, which **prompt** then offers on **out**.

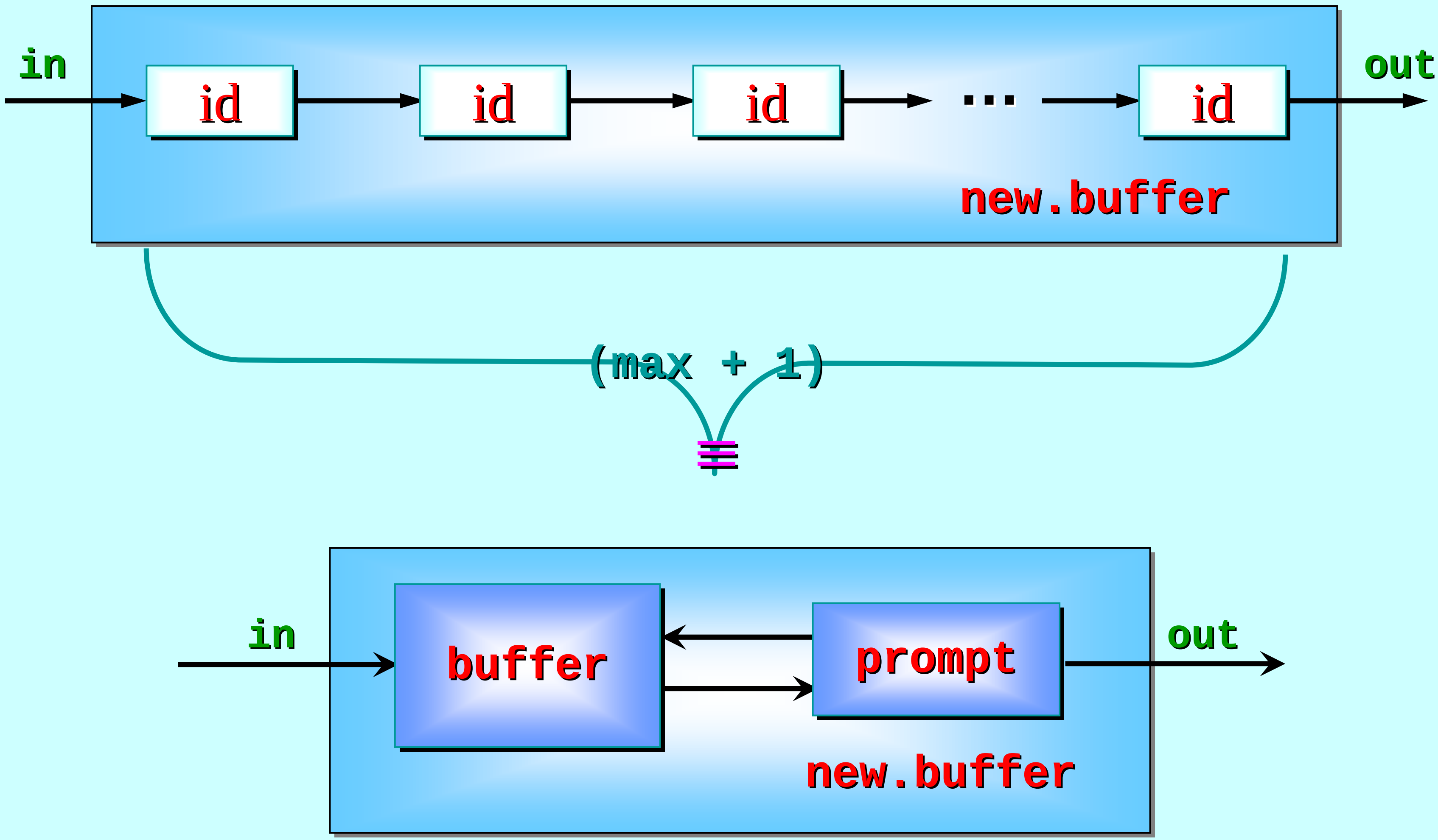
So, **new.buffer** is just a *FIFO* with capacity **(max + 1)**. And it has single input/output lines – no request is needed.

To relieve the receiving process from the bother of making the requests, we can install an *auto-prompter* alongside the **buffer**:

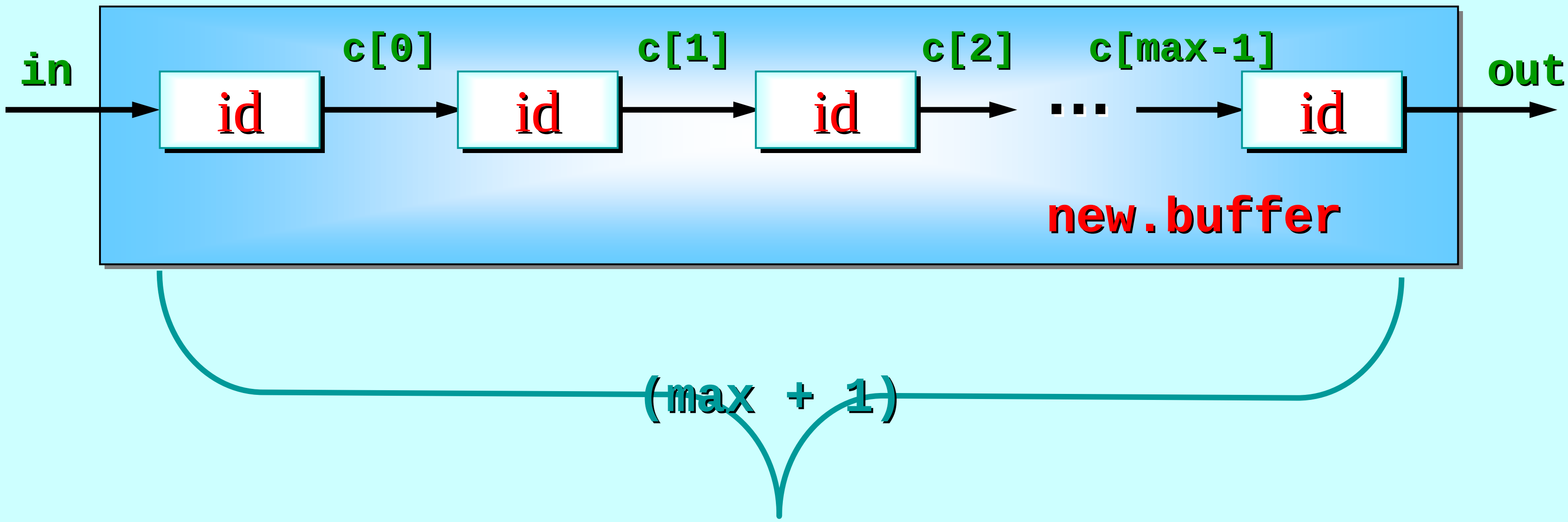



```
PROC new.buffer (CHAN INT in?, out!)
  CHAN BOOL req:
  CHAN INT ans:
  PAR
    buffer (in?, req?, ans!)
    prompt (ans?, req!, out!)
:
```

The capacity of **new.buffer** is **(max + 1)**

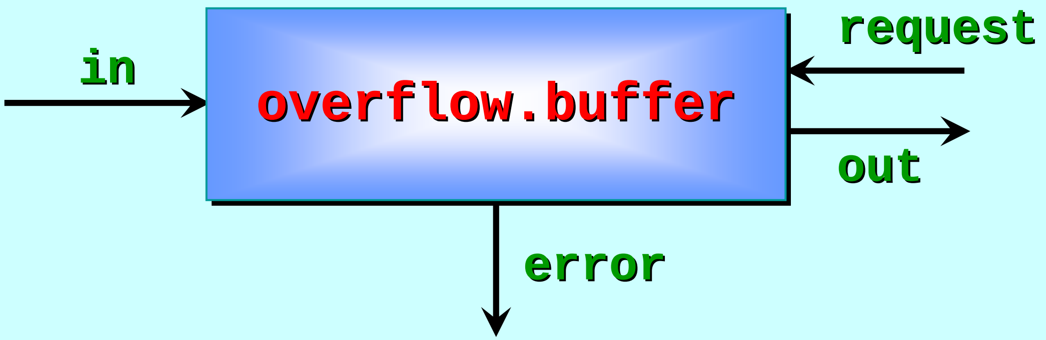


The top version is a more regular and simpler design. The bottom is more efficient for software – less copying of data.




```
PROC new.buffer (CHAN INT in?, out!)
  [max]CHAN INT c:
  PAR
    id (in?, c[0]!)
    PAR i = 0 FOR max – 1
      id (c[i]?, c[i+1]!)
    id (c[max – 1]?, out!)
:
```

Exercise:



This is the same as **buffer**, except that it does not block the source when it is full. Instead, it outputs a signal on the (**BOOL**) **error** line and discards the incoming item.

This type of buffer is used in a real-time system if it is important not to delay the source process if the receiver is slow *and* it is not crucial if we miss some items, so long as we know about it!

Exercise:

This is the similar to **overflow.buffer**; it also does not block the source when it is full. However, the incoming item (when full) is not discarded but *overwrites* the oldest item in the buffer. No error is reported for this (though another version could easily do that).

This type of buffer is used in a real-time system if it is important not to delay the source process if the receiver is slow *and* we don't mind losing old items when full. Whatever it holds, it always holds the *latest* values received from the source.

Choice and Non-Determinism

Non-determinism ...

The **ALT** and **PRI ALT** …

Control and real-time …

Resets and kills …

Memory cells ...

Pre-conditioned guards …

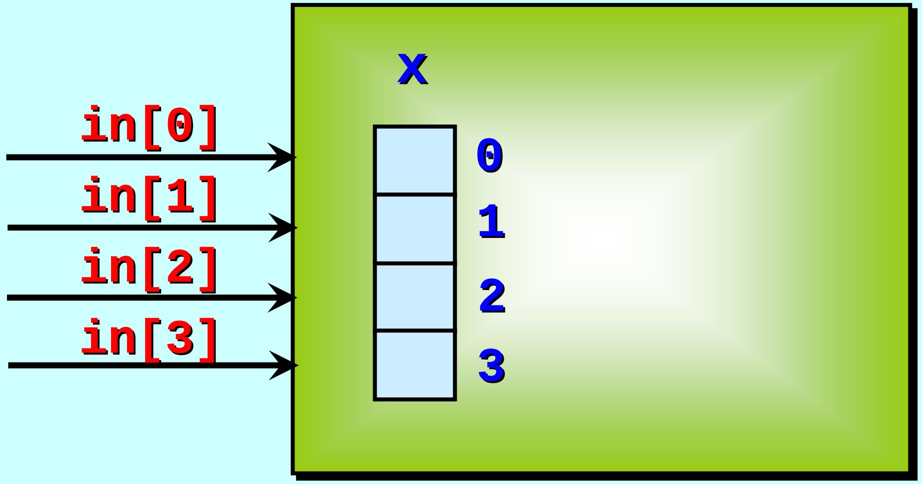
Serial FIFO *(‘ring’)* buffer …

The replicated **ALT** …

Nested **ALT**s …

The Replicated ALT

Consider a process with an array of input channels:

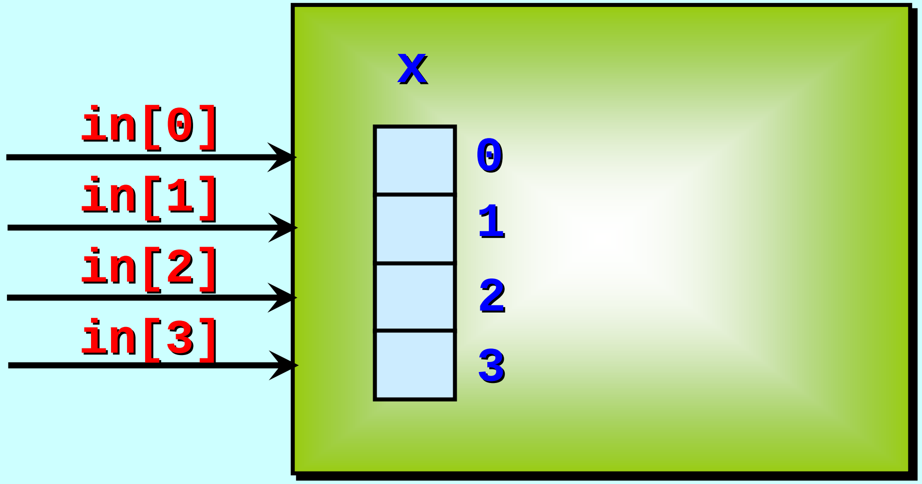


And an internal data array of the same type and size as the input channel array.

The process needs to accept any message from any input channel, putting it into the corresponding element of its data array.

The Replicated ALT

Consider a process with an array of input channels:

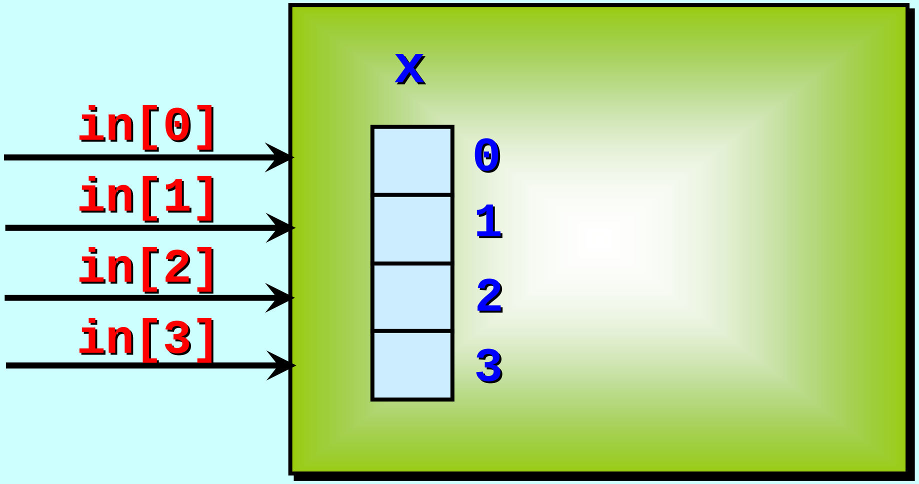


Before, we introduced the *replicated* **PAR** for this. We knew that a message on *one* channel was accompanied by a message on *all* channels.

This time, we don't know the frequency *(if any)* with which messages will arrive from any channel.

The Replicated ALT

We must await these inputs with an **ALT**:

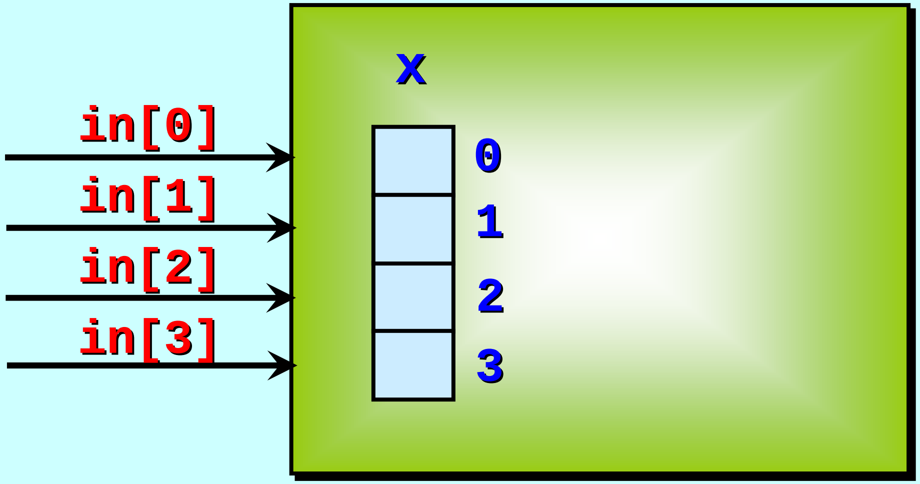



```
ALT
  in[0] ? x[0]
    ...  deal with it
  in[1] ? x[1]
    ...  deal with it
  in[2] ? x[2]
    ...  deal with it
  in[3] ? x[3]
    ...  deal with it
```

But what if there were 40 channels in the array? Or 400 … or 4000 … ?!!

The Replicated ALT

We must await these inputs with an **ALT**:



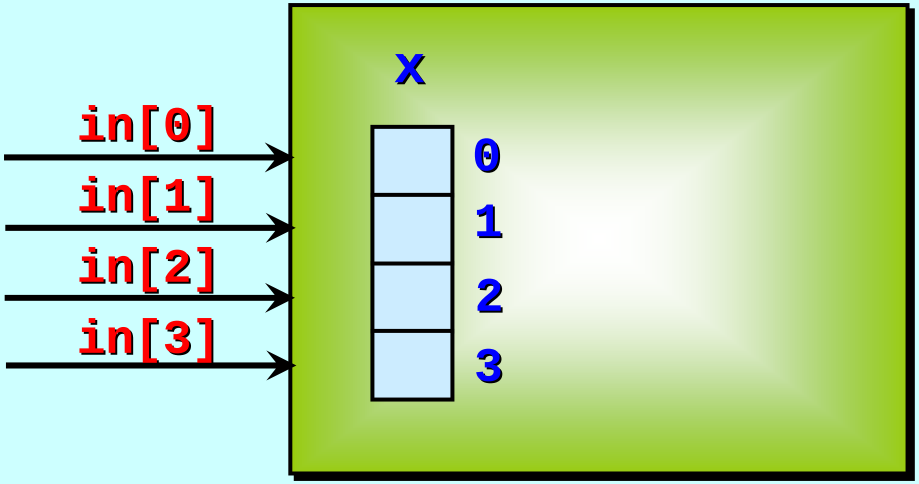
INT declaration · **first value** · **number of replications**

```
ALT i = 0 FOR 4
  in[i] ? x[i]
    ...  deal with it
```

This guarded process gets replicated

The Replicated ALT

We must await these inputs with an **ALT**:

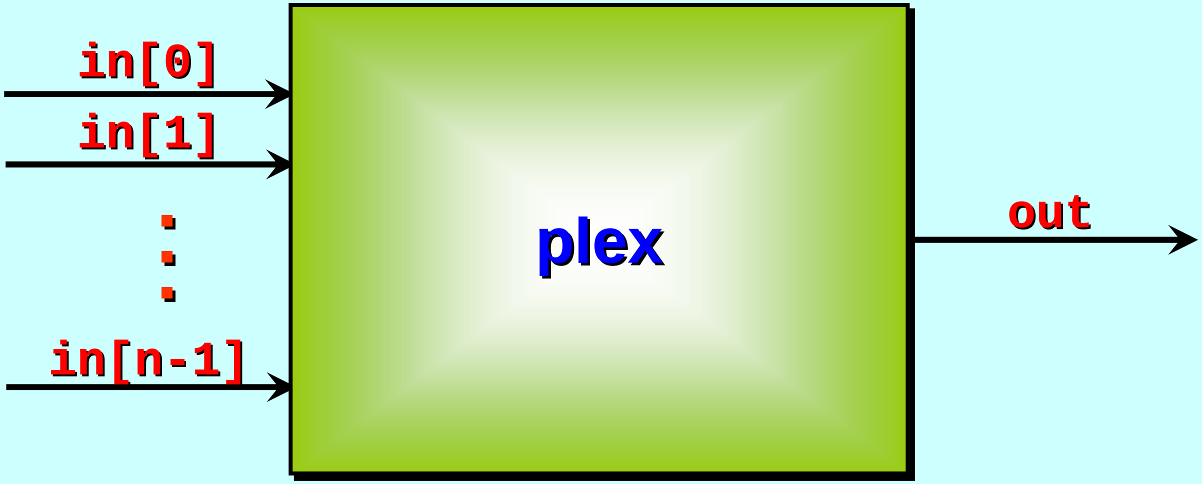



```
ALT
  in[0] ? x[0]
    ...  deal with it
  in[1] ? x[1]
    ...  deal with it
  in[2] ? x[2]
    ...  deal with it
  in[3] ? x[3]
    ...  deal with it
```

≡

```
ALT i = 0 FOR 4
  in[i] ? x[i]
    ...  deal with it
```

A Simple Multiplexor

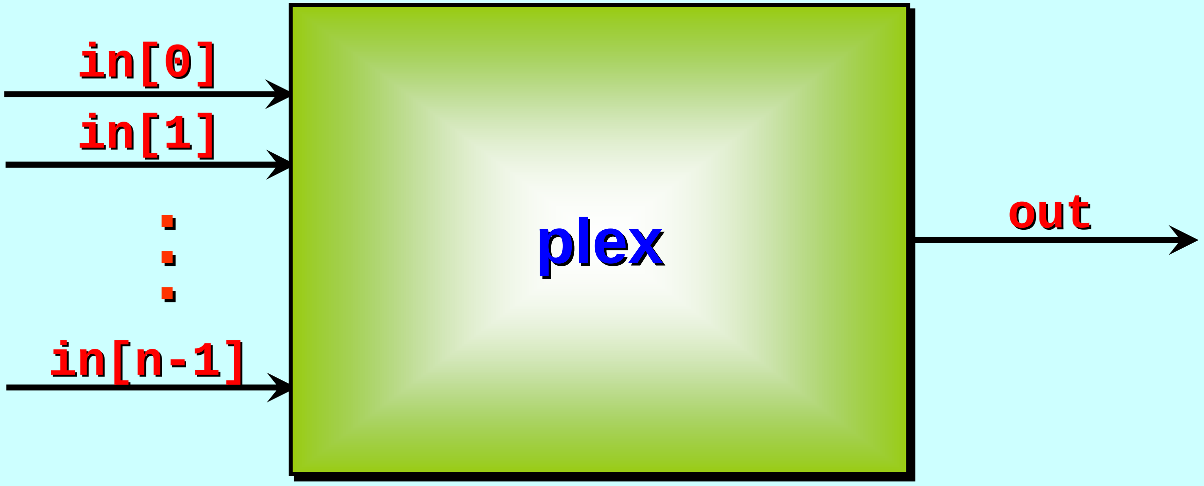


This process just forwards any message it receives …

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

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

A Simple Multiplexor




```
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

A Matching De-Multiplexor

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

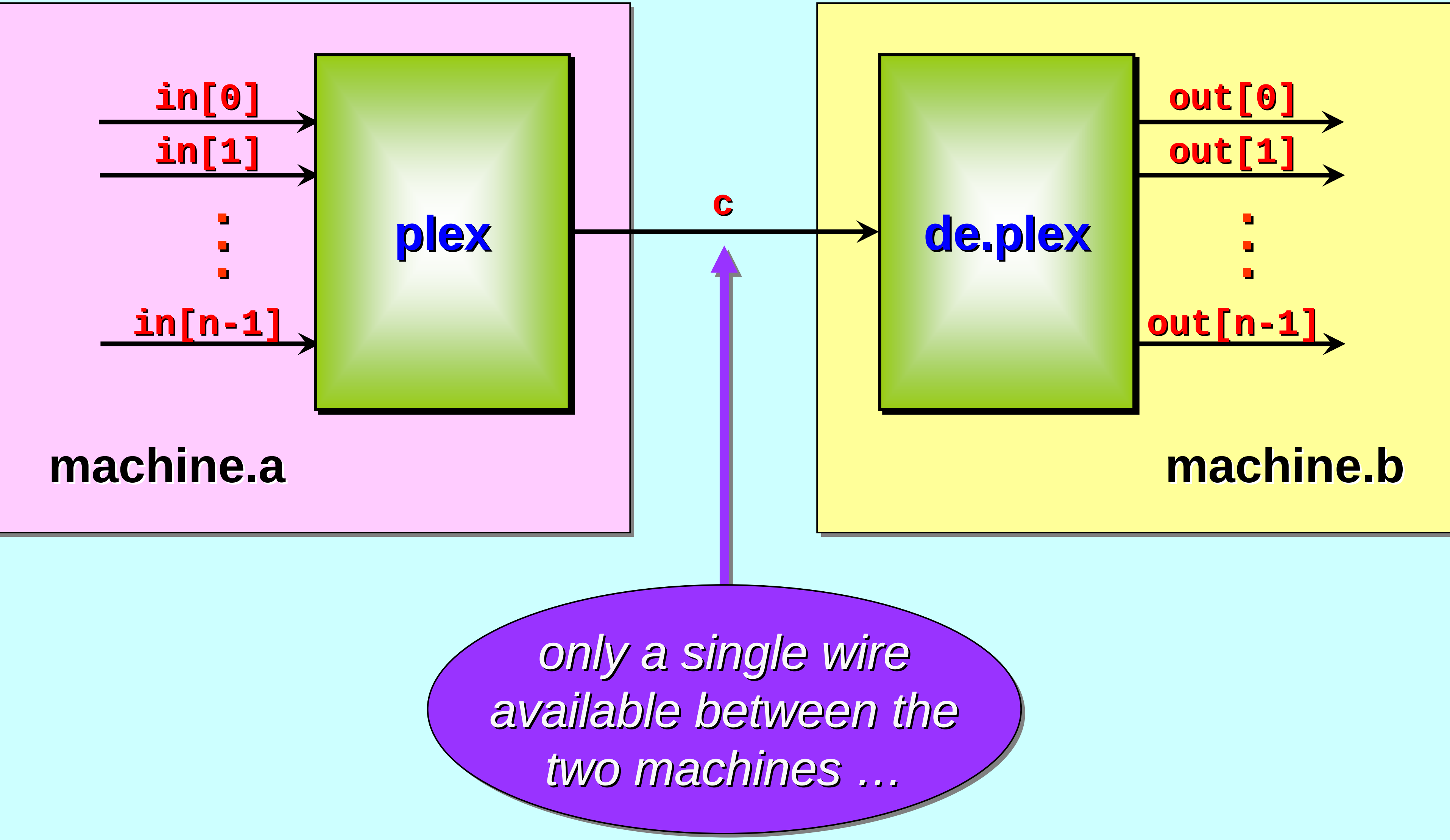
Each message must be a `<index, data>` pair, generated by a `plex` process (with the same number of inputs as this has outputs).

A Matching De-Multiplexor

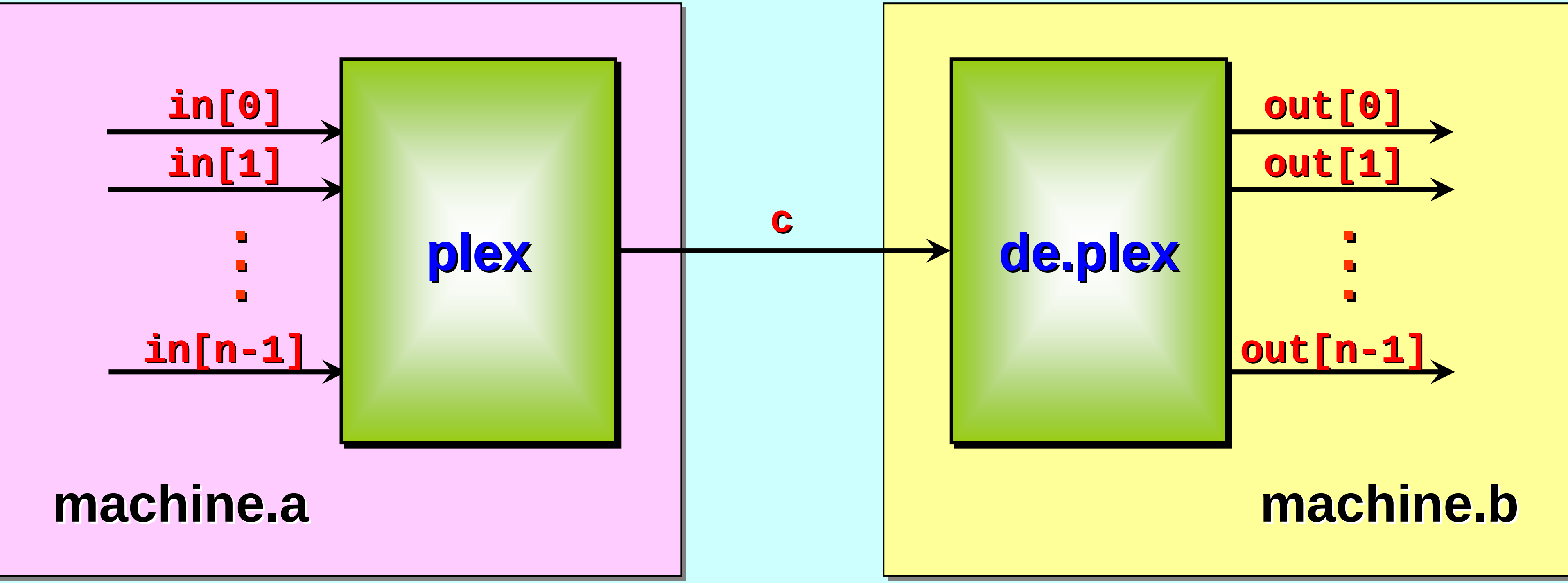

```
PROC de.plex (CHAN INT in?, []CHAN INT out!)
  WHILE TRUE
    INT i, x:
    SEQ
      in ? i
      in ? x
      out[i] ! x
:
```

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

Multiplexor Application (Example)

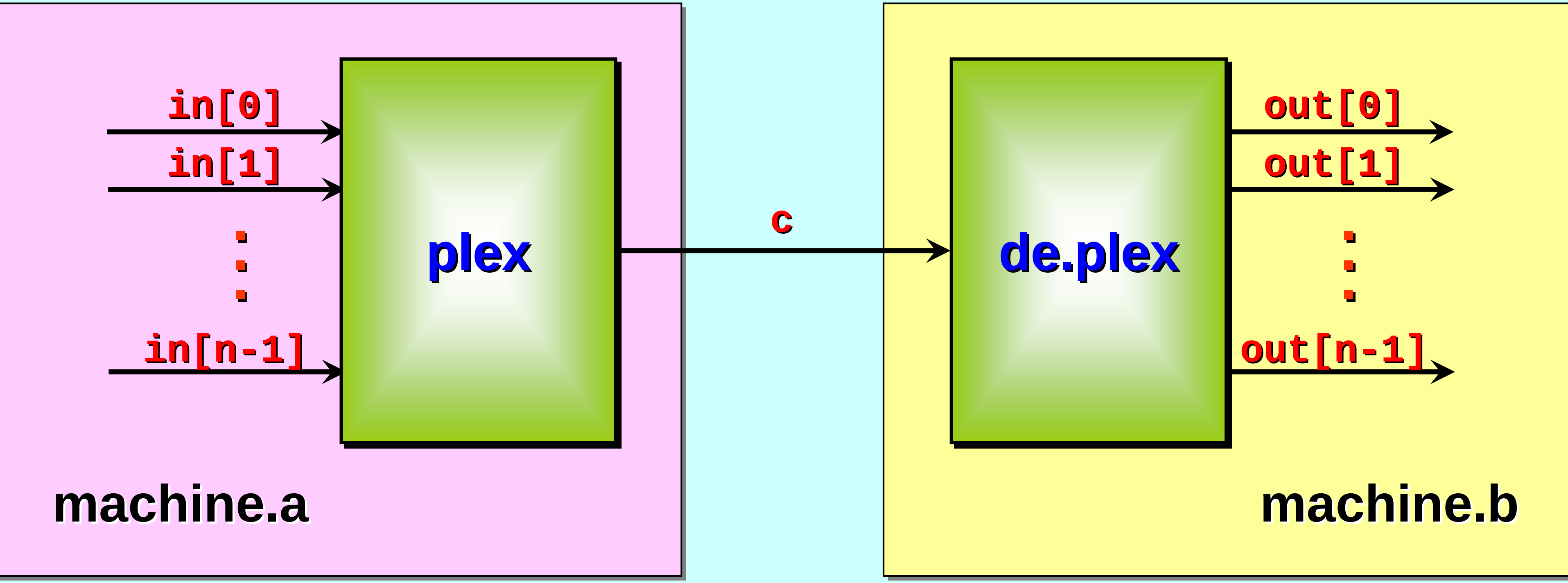


Multiplexor Application (Example)



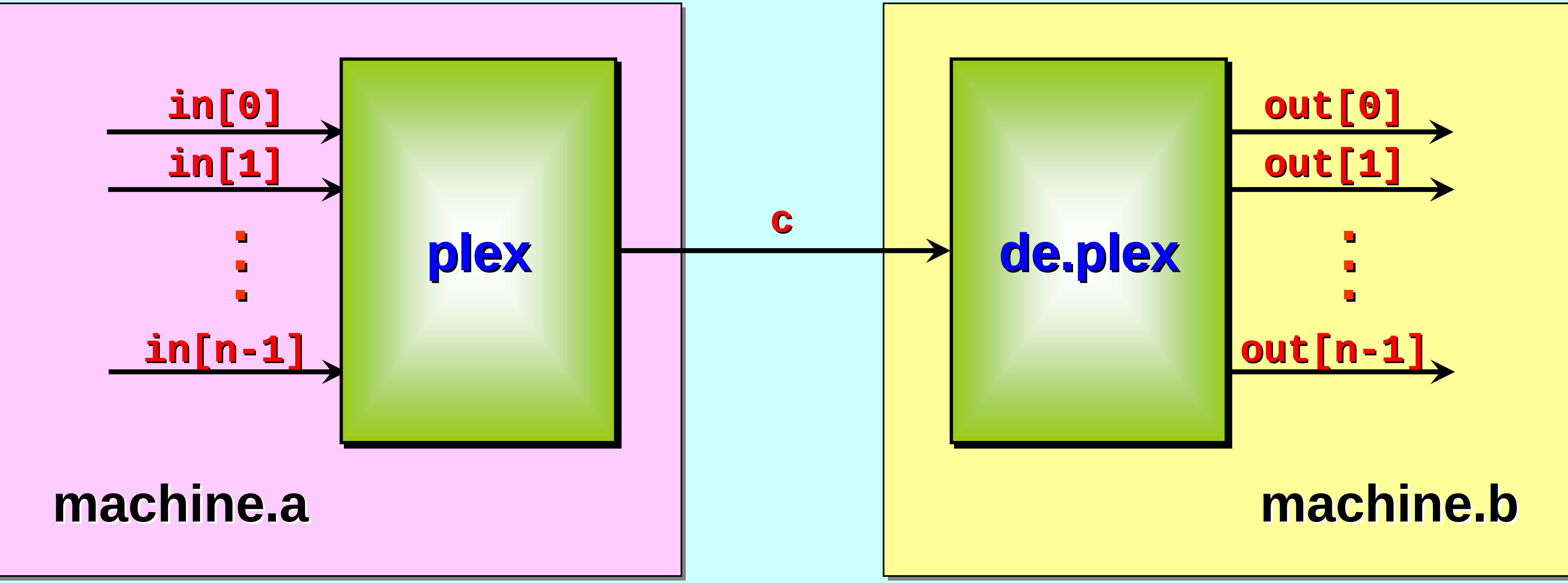
If each *message* arriving at **`plex`** (and departing **`de.plex`**) is of type **`THING`**, then each *message* on the *multiplexed* channel consists of a channel array index (type **`INT`**) followed by a **`THING`**.

Multiplexor Application (Example)



Message structures should be *documented* somewhere!

Multiplexor Application (Example)

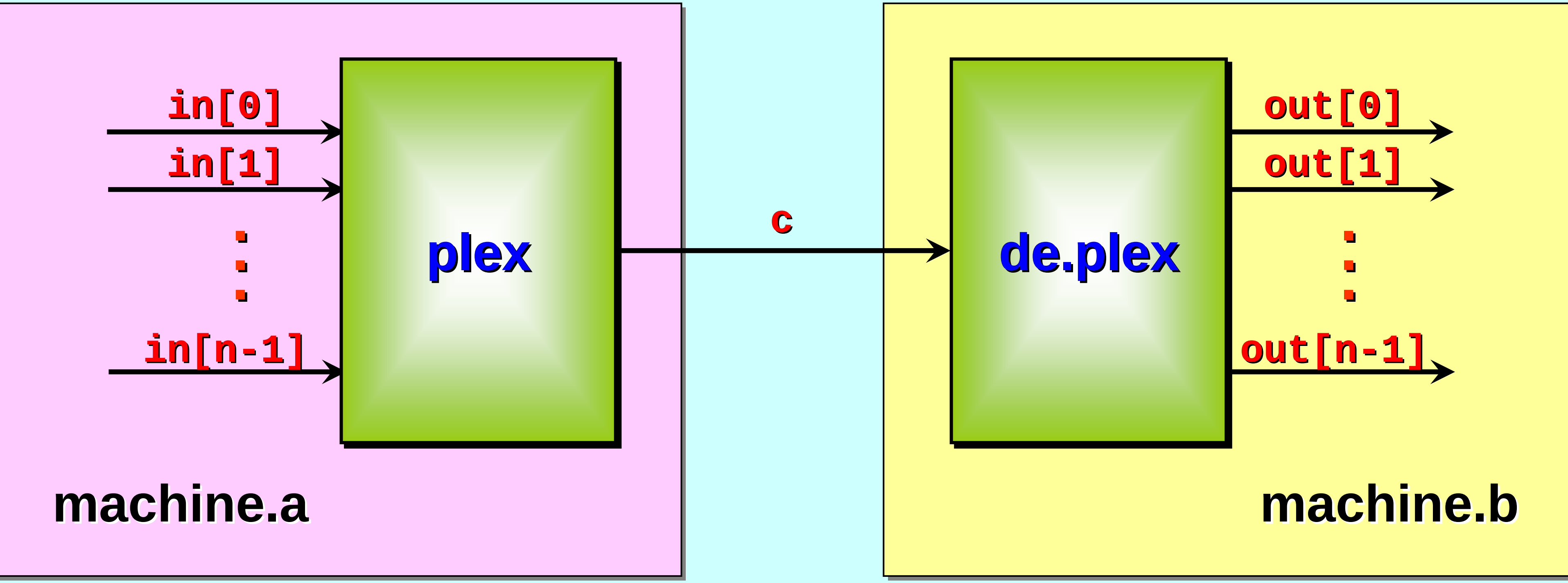


In our example, we were fortunate that the *messages* to be multiplexed were type **INT** – the same as channel indices!

This lets us type the *multiplexed* channel: **CHAN INT c:**

Remembering that *messages* on **c** have form: **INT; INT**

Multiplexor Application (Example)

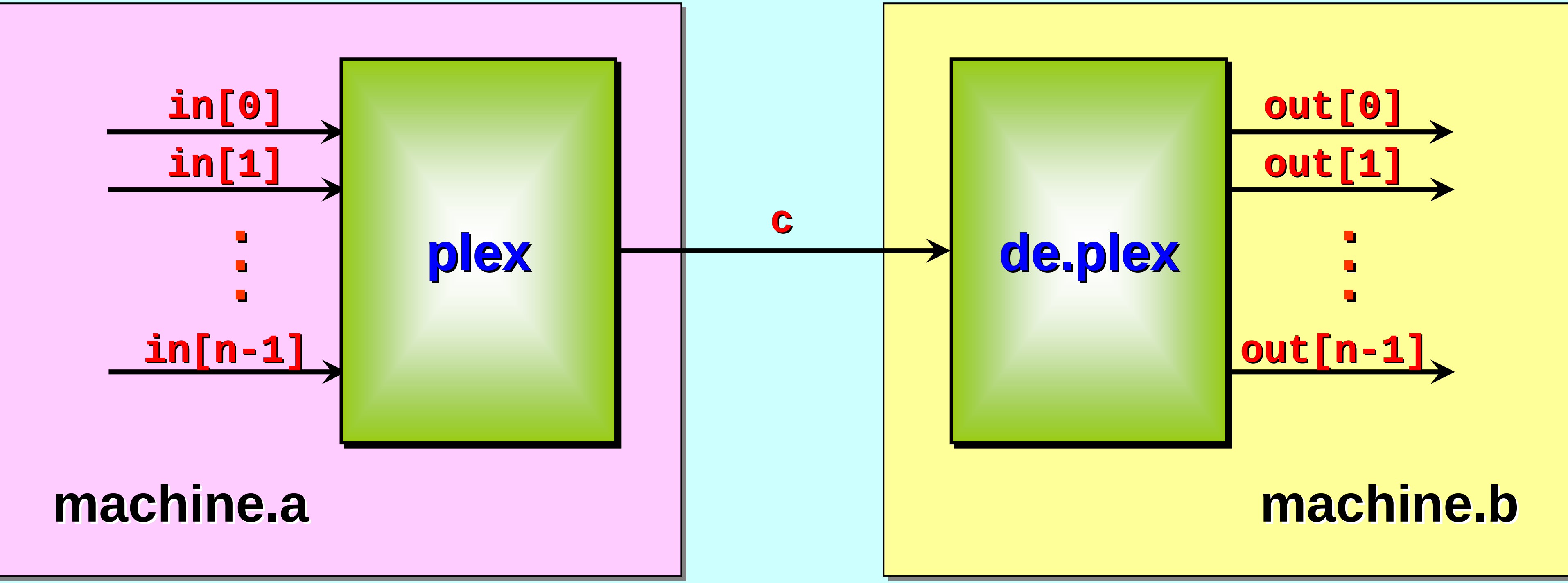


However, suppose that the *messages* to be multiplexed were type **REAL64** …

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

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

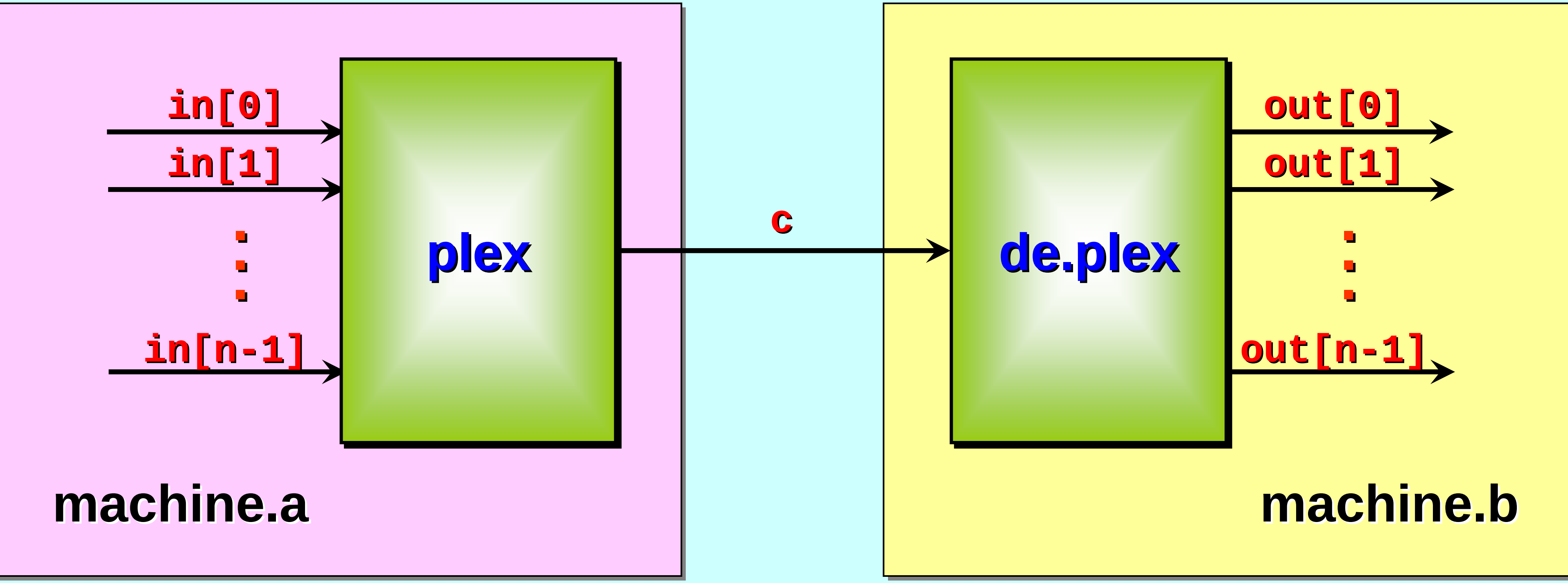
Multiplexor Application (Example)



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).

Multiplexor Application (Example)



We will return to this example in the chapter on message **PROTOCOL**s.

Choice and Non-Determinism

Non-determinism ...

The **ALT** and **PRI ALT** …

Control and real-time …

Resets and kills …

Memory cells ...

Pre-conditioned guards …

Serial FIFO *('ring')* buffer …

The replicated **ALT** …

Nested **ALT**s …

Nested ALTs and PRI ALTs

```
ALT
  <guard 0>
    <process 0>
  ALT
    <guard 1>
      <process 1>
    <guard 2>
      <process 2>
  <guard 3>
    <process 3>
```

≡

```
ALT
  <guard 0>
    <process 0>
  <guard 1>
    <process 1>
  <guard 2>
    <process 2>
  <guard 3>
    <process 3>
```

The inner **ALT** disappears and its *guarded processes* align with the *guarded processes* of the outer **ALT**.

Nested ALTs and PRI ALTs

```
PRI ALT
  <guard 0>
    <process 0>
  ALT
    <guard 1>
      <process 1>
    <guard 2>
      <process 2>
  <guard 3>
    <process 3>
```

≡

```
PRI ALT
  <guard 0>
    <process 0>
  <guard 1>
    <process 1>
  <guard 2>
    <process 2>
  <guard 3>
    <process 3>
```

An **ALT** nested inside a **PRI ALT** gets *prioritised* ...

Nested ALTs and PRI ALTs

```
PRI ALT
  <guard 0>
    <process 0>
  ALT
    <guard 1>
      <process 1>
    <guard 2>
      <process 2>
  <guard 3>
    <process 3>
```

≡

```
PRI ALT
  <guard 0>
    <process 0>
  <guard 1>
    <process 1>
  <guard 2>
    <process 2>
  <guard 3>
    <process 3>
```

... which is OK *(an* **ALT** *can always be replaced by a* **PRI ALT***)*

Nested ALTs and PRI ALTs

```
ALT
  <guard 0>
    <process 0>
  PRI ALT
    <guard 1>
      <process 1>
    <guard 2>
      <process 2>
  <guard 3>
    <process 3>
```

≢

```
ALT
  <guard 0>
    <process 0>
  <guard 1>
    <process 1>
  <guard 2>
    <process 2>
  <guard 3>
    <process 3>
```

A **PRI ALT** nested inside an **ALT** is *illegal* ...

Nested ALTs and PRI ALTs

```
ALT
  <guard 0>
    <process 0>
  PRI ALT
    <guard 1>
      <process 1>
    <guard 2>
      <process 2>
  <guard 3>
    <process 3>
```

≢

```
ALT
  <guard 0>
    <process 0>
  <guard 1>
    <process 1>
  <guard 2>
    <process 2>
  <guard 3>
    <process 3>
```

... *(a* **PRI ALT** *cannot always be replaced by an* **ALT***)*

Nested **ALT**s and **PRI ALT**s

```
ALT
  <guard 0>
    <process 0>
  ALT i = 0 FOR n
    <rep guard i>
      <rep process i>
  <guard 1>
    <process 1>
```

≡

```
ALT
  <guard 0>
    <process 0>
  ALT
    <rep guard 0>
      <rep process 0>
    .
    .
    .
    <rep guard (n-1)>
      <rep process (n-1)>
  <guard 1>
    <process 1>
```

Nested **ALT**s are mainly useful … *when the inner or outer is replicated.*

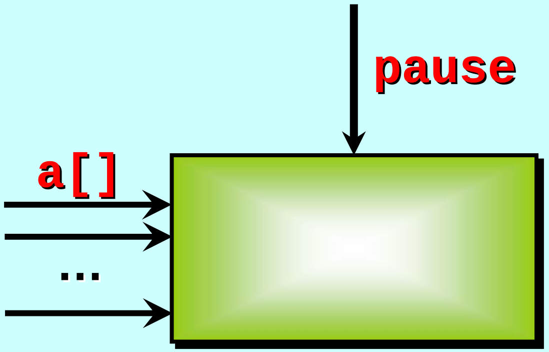
Nested ALTs and PRI ALTs

```
ALT
  <guard 0>
    <process 0>
  ALT i = 0 FOR n
    <rep guard i>
      <rep process i>
  <guard 1>
    <process 1>
```

≡

```
ALT
  <guard 0>
    <process 0>
  <rep guard 0>
    <rep process 0>
  .
  .
  .
  <rep guard (n-1)>
    <rep process (n-1)>
  <guard 1>
    <process 1>
```

They enable us to **ALT** between *arrays* of guards and *individuals*.



For example:

```
PRI ALT
  tim ? AFTER timeout
    ...  deal with it
  BOOL any:
  pause ? any
    pause ? any
  ALT i = 0 FOR SIZE a?
    INT x:
    a[i] ? x
      ...  deal with it
```

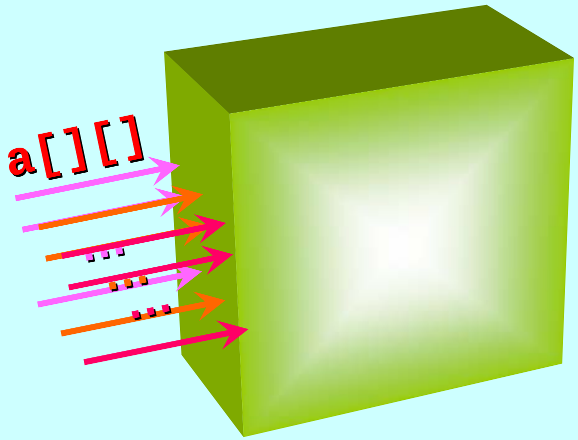
ALTing between *an array* of channel inputs, *a single* channel input and *a single* timeout.

For example:

ALTing between *two arrays* of guards.

```
ALT
  ALT i = 0 FOR SIZE a?
    INT x:
    a[i] ? x
      ...  deal with it
  ALT i = 0 FOR SIZE b?
    INT x:
    b[i] ? x
      ...  deal with it
```

For example:



ALTing between a *2D array* of guards.

```
ALT i = 0 FOR SIZE a?
  ALT j = 0 FOR SIZE a[i]?
    INT x:
    a[i][j] ? x
      ...  deal with it
```