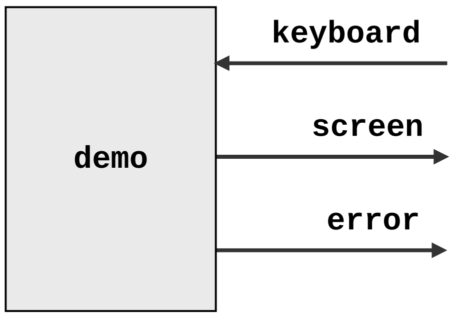
Concurrency and occam-π

occam Exercises (system control)

Exercise 3:

[The starter file for this is `q3.occ` in your `course/exercises` folder.]

This is an exercise on modifying the way we interact with a component without modifying the component itself - i.e. treating the component as a `black box'. In this case, the black box is the main process in the `demo.occ` example (see <*wherever*>`/course/examples`):

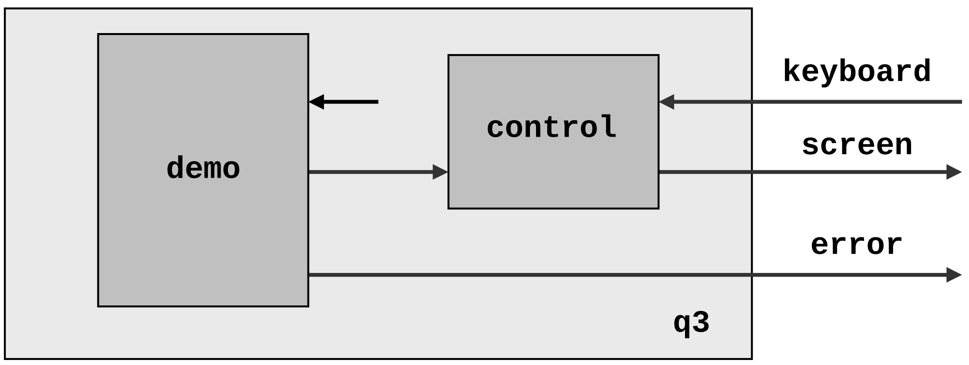


`demo` only uses its `screen` channel – so, kroc users may remove the other two. *[For the moment, Transterpreter users must have all three for their main processes.]*

This exercise is to produce a new component that produces the same stream of bytes (ASCII) on its `screen` channel, but which also responds to keystrokes delivered to its `keyboard` channel as follows. If the keystroke is:

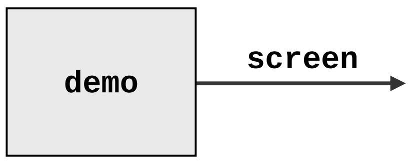
- ‘f’ : freeze all output, resuming output (from wherever it left off) only after receiving the next `keyboard` input. (any character);
- anything else : accept, but ignore.

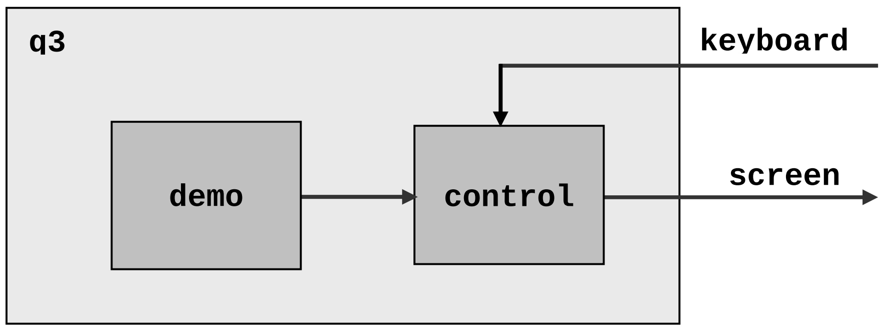
This new component must be built using an unaltered `demo` component. To get the required control, we must put something (another process) between the interface pins (external channels) of `demo` and the actual `keyboard` and `screen` channels:



Design and implement a suitable `control` process and build the above `q3` system. The `demo` process is included in the starter file for this exercise.

[Note: those using `kroc` *(rather than the Transterpreter just yet) may modify* `demo` *and* `q3` *to drop the unused channels. The problem remains the same … but the system diagrams simplify:*





]

Modify the `control` process a little further so as to see the results from some run-time errors. Not that your systems will ever suffer such things – this is just so you have the experience, ☺. Here is the new spec – if the system receives:

- 'f' : *freeze* all output, resuming output (from wherever it left off) only after receiving the next `keyboard` input. (any character);
- 'd' : force the system into *deadlock* – the *occam-pi* runtime will detect this and exit your program;
- 'z' : attempt to *divide-by-zero* – the *occam-pi* runtime will detect the error and exit your program;
- 'v' : attempt to *violate an array index bound* – as above;
- 's' : attempt to execute a `STOP`;
- anything else : *accept*, but ignore.

You will need to think how to force a deadlock. Leaving a process waiting on an *external* channel (e.g. `control` waiting for `keyboard` input) is not deadlock … the external event will be accepted. For deadlock, all processes must be blocked waiting on *internal* channels. There is a trivial solution.

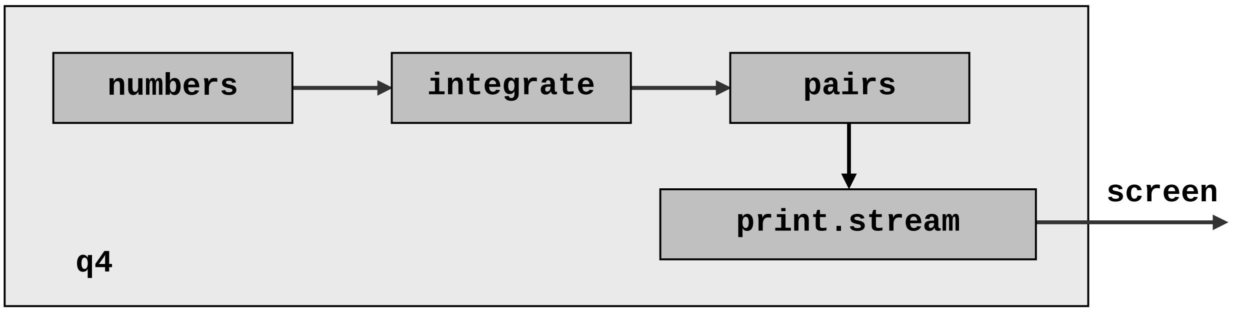
Formally, the *correct* semantics of the `STOP` primitive is that the process freezes … but that processes running in parallel with it carry on unaffected. By default, this is *not* the way `kroc` (nor the Transterpreter) compiles it! For pragmatic reasons, we compile it as a (deliberate) *run-time error* and any process executing a `STOP` crashes the whole system. Similarly, the *correct* semantics of a process executing any run-time error is that it freezes (like `STOP`). The default compilation is that any process executing any run-time error crashes the whole system – just like `STOP`. This mode of compilation is helpful when developing code – *arguably*. For the *correct* semantics of `STOP` and run-time errors, use the compiler "`-S`" flag (e.g. "`kroc -S q3.occ`", that's an upper case "-S"). With the *correct* semantics, run-time errors are isolated in the processes that make them – *arguably* safer for critical applications.

[By the way, to get a bit more feedback on run-time errors, compile with the debug flag set (e.g. "`kroc -d q3.occ`", that's a lower case "-d", or maybe "`kroc -S -d q3.occ`").]

Exercise 4:

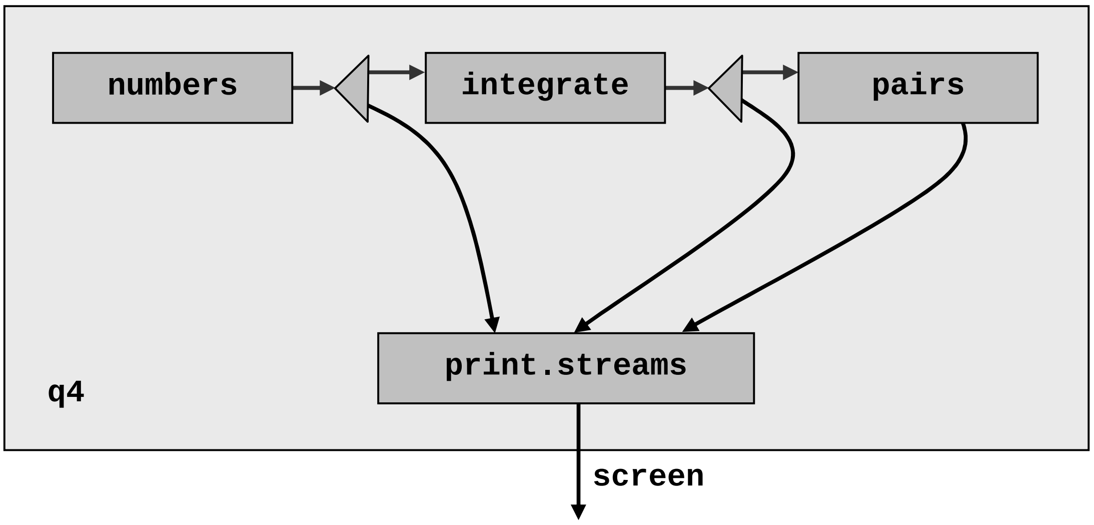
[The starter file for this is `q4.occ` in your `course/exercises` folder.]

Rebuild the `squares` pipeline (used in the demo program and given in slide 95 of "basics") and pipeline it into `print.stream` to make a system that outputs perfect squares — one per line:



Don't rewrite the sub-processes above. The first three are in the `course.module` and the other is already in your starter file – just instance them. *[Note: those using the Transterpreter will have to add external `keyboard` and `error` channels.]*

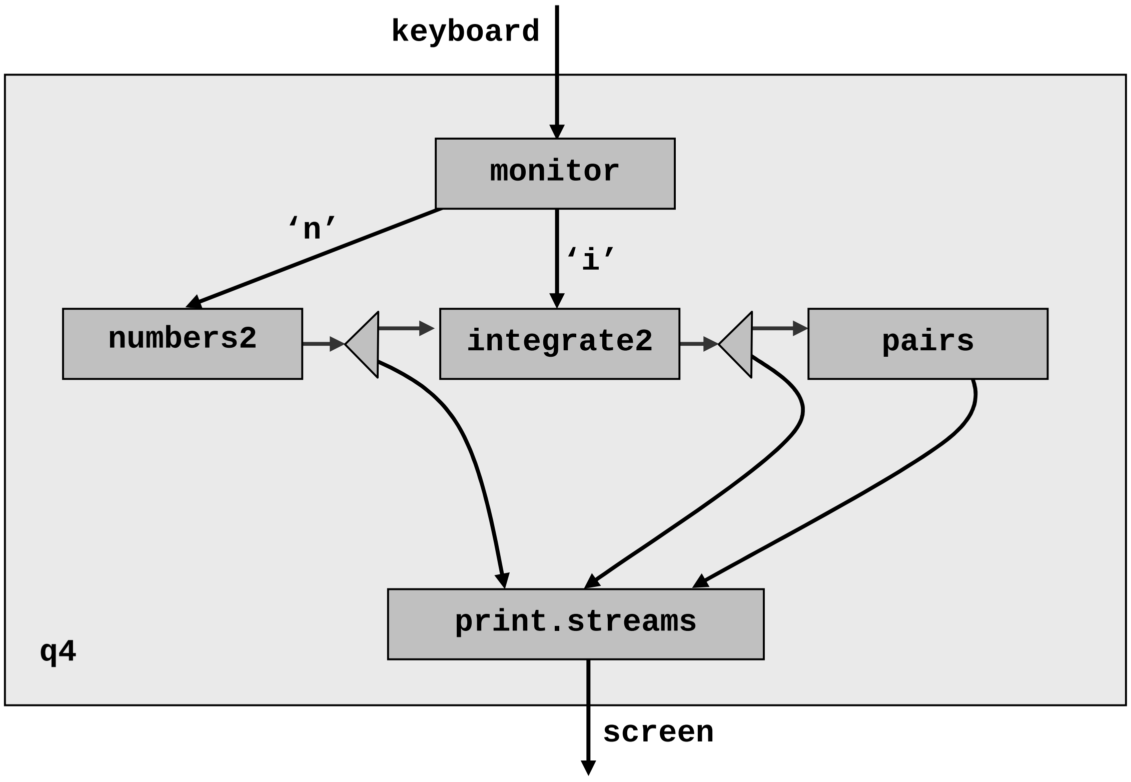
Modify this system to tap into the internal channels, using `delta` processes (in `course.module`) to duplicate lanes to a `print.streams` multiplexor (in the starter file). Then, we can see the streams at each stage in the pipeline:



Now, add the `keyboard` channel with a `monitor` process so we can control this machine! Get the system to respond to `keyboard` input as follows. If the system receives:

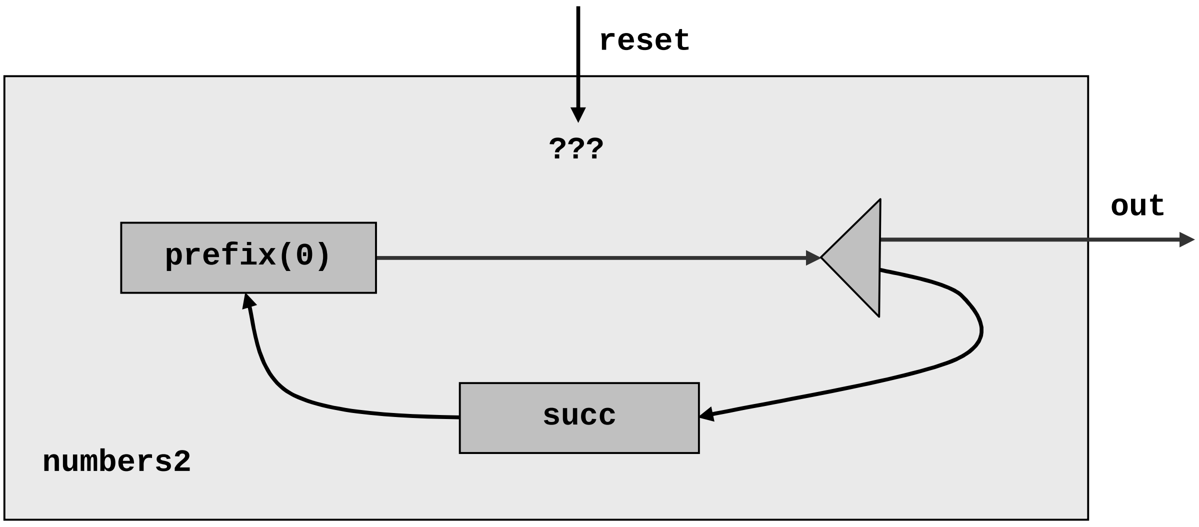
- 'n' : *reset* the `numbers` process to start counting from zero again;
- 'i' : *reset* the running sum in the `integrate` process back to zero again;
- anything else : *accept*, but ignore.

Of course, the state values we are being asked to reset are *internal* to their respective processes and cannot be changed from the outside. Those processes, therefore, must be modified to accept *reset requests* (on extra channels) and perform the resets themselves. The *reset requests* will be generated by the `monitor` process:



For information, the *reset* channels above have been labelled with the characters that cause them to be fired. They will be integer carrying channels. Generalise the notion of these *resets* so that the processes may have their relevant states reset to *any* number sent down those channels. However, for this exercise, the `monitor` process should only send zeroes.

New processes, `numbers2` and `integrate2`, must be made that accept the new reset signals and respond appropriately. Maintain the same style of implementation as before. Don't modify any internal processes … but insert an extra one somewhere to do the job. For example:



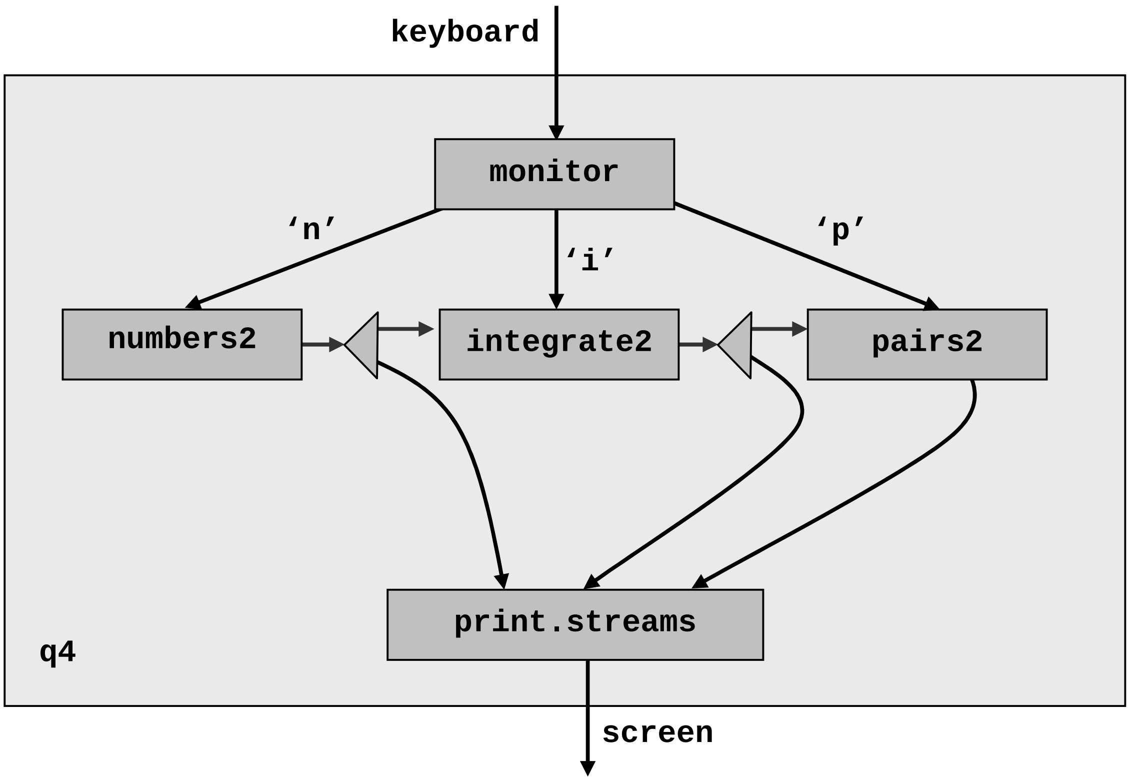
[Note: just instance `prefix`, `delta` and `succ` when you write the above – their declarations are in the course.module included at the start of your starter file. Look in your starter file for a component that might be suitable for the one missing above.]

Now, do something very similar for a *running-sum-resettable* `integrate2`.

Extend the behaviour of `monitor` so that the system has an additional control:

- ‘n’ : *reset* the `numbers` process to start counting from zero again;
- ‘i’ : *reset* the running sum in the `integrate` process back to zero again;
- ‘p’ : *flip* the behaviour of the `pairs` process so that its *adder* changes to a *subtractor* – a subsequent ‘p’ flips it back again. *[Note: in its flipped mode, the modified* `pairs2` *process becomes a differentiator so that the stream of numbers produced are the same (bar a bit of slosh) as those coming from* `numbers2`*.]*
- anything else : *accept*, but ignore.

We just need an additional channel to a modified `pairs`:



One way is to bring the new reset channel into a *modified* `plus` process inside `pairs2` – which flips between adding and subtracting as resets arrive. A neater way is to leave the existing sub-components alone, but introduce an extra component into the circuit to achieve the required effect. *[A process suitable for this new component has been described in the course.]*

Let’s carry on. Extend the behaviour of `monitor` so that the system has *freeze control*:

- ‘n’ : *reset* the `numbers` process to start counting from zero again;
- ‘i’ : *reset* the running sum in the `integrate` process back to zero again;
- ‘p’ : *flip* the behaviour of the `pairs` process so that its *adder* changes to a *subtractor* – a subsequent ‘p’ flips it back again. *[Note: in its flipped mode, the modified* `pairs` *process becomes a differentiator so that the stream of numbers produced are the same (bar a bit of slosh) as those coming from* `numbers`*.]*
- ‘f’ : *freeze* all output, resuming output (from wherever it left off) only after receiving the next `keyboard` input. (any character);
- anything else : *accept*, but ignore.

This is similar to Exercise 3 – but careful you don't introduce deadlock. You are on your own here – draw your own network diagram before writing any code!

Finally, let's introduce *speed control*:

- 'n' : *reset* the `numbers` process to start counting from zero again;
- 'i' : *reset* the running sum in the `integrate` process back to zero again;
- 'p' : *flip* the behaviour of the `pairs` process so that its *adder* changes to a *subtractor* – a subsequent 'p' flips it back again. *[Note: in its flipped mode, the modified* `pairs` *process becomes a differentiator so that the stream of numbers produced are the same (bar a bit of slosh) as those coming from* `numbers`*.]*
- 'f' : *freeze* all output, resuming output (from wherever it left off) only after receiving the next `keyboard` input. (any character);
- '+' : double the rate of output of lines of text (up to a maximum of 256 lines/second);
- '-' : halve the rate of output of lines of text (down to a minimum of 1 line/second);
- anything else : *accept*, but ignore.

Leave the definition of `print.streams` alone. Just instance it with a delay of minus one (-1); it plays no part now in controlling speed.

Specify, implement and place a new component (`speed.control`) to manage speed control. This new component sets an initial speed of 32 lines per second. *Note:* the system cannot, of course, generate output faster than the receiving device (e.g. a terminal screen) can take it. The new control process may ignore this problem – i.e. if the receiving device can only display 200 lines per second, speed settings beyond that will be automatically cut back to 200 (with no loss of data). This is fine!

The new component should also provide feedback to the user when a '+' or '-' tries to push the output speed over its limits: the attempt should be ignored and an *error message* generated. To keep things simple and not interfere with the columns of numbers being output, this error message should be a single `BELL` character (ASCII code 7) sent to the `error` output channel (which now needs to be introduced to the main process parameter list, if not there already). The `BELL` character is provided (as a `BYTE` constant) by the `course.module`. `BYTE`s output on the `error` channel do not need flushing – they are delivered to the user's terminal (window) without delay. Their effect varies depending on how the terminal has been set up. Some terminals will emit a short *beep*. Others will flash the screen by momentarily inverting the colours on the screen.

One last piece of low-level help: processing the character input from the `keyboard` channel is more neatly handled though the `CASE`/`ELSE` construct of occam-pi, rather than the `IF`. The `CASE`/`ELSE` construct is described in the occam-pi reference wiki:

https://www.cs.kent.ac.uk/research/groups/sys/wiki/OccamPiReference

and in slides 38-46 of the "shared-etc" slides.