2 Programming structures

In the next few pages, the various pipes and joints of *occam* are demonstrated in some small plumbing exercises. Although these examples may seem unrealistic or overly elaborate for their size, they are intended to show some practical programming techniques.

Simple sequential processes

It is almost traditional that the first program anyone writes in a new programming language is one that writes "Hello", or some equally imaginative greeting, to the screen of their terminal. In an *occam* environment, the terminal screen is likely to be accessible as a channel: values output to the channel being displayed on the screen as characters. A first, unexciting attempt at the "Hello" program is

```
SEQ
  terminal.screen ! ‘H’
  terminal.screen ! ‘e’
  terminal.screen ! ‘l’
  terminal.screen ! ‘l’
  terminal.screen ! ‘o’
```

or more compactly

```
terminal.screen !  ! ‘H’; ‘e’; ‘l’; ‘l’; ‘o’
```

Looking for ways to generalize the program, it would be natural to write a loop that outputs each of the characters of an *occam* string. Recall that a string is a `BYTE` array, with the number of characters being `string[BYTE 0]` so that a program to write string should behave like

```
SEQ
  output ! string[BYTE 1]
  output ! string[BYTE 2]
         :
  output ! string[BYTE string[BYTE 0]]
```

This is patently a candidate for a SEQ-FOR loop, which can be written

```
SEQ character.number = [1 FOR string[BYTE 0]]
  output ! string[BYTE character.number]
```

The process can be packaged as a named process, corresponding to a procedure in a language such as PASCAL, for writing the characters of a string to a channel

```
PROC write.string(CHAN output, VALUE string[]) =
  -- Send the characters of the string along output
  SEQ character.number = [1 FOR string[BYTE 0]]
    output ! string[BYTE character.number]         :
```

The line with the hyphens on it is a comment: these can appear at the end of any line in an *occam* program, even otherwise blank ones as here. Writing comments summarizing the behaviour of named processes is probably a good habit to cultivate.

In consequence of some unfortunate experiences in my past, I have come to expect my computer to greet me with the shibboleth “Bootifrolo”. This might be done by using *write.string*, as follows

```
PROC write.string(CHAN output, VALUE string[]) =
  -- Send the characters of the string along output
  SEQ character.number = [1 FOR string[BYTE 0]]
    output ! string[BYTE character.number]         :

write.string(terminal.screen, "Bootifrolo")
```

Simple parallel processes

The simplest thing that you can usefully want a process to be doing, at the same time as another process is doing something else, is to copy data from one channel to another. This is just a matter of repeatedly taking input from one channel, storing it in a local variable, and then sending the value of the variable along another channel.

```
VAR local :
SEQ
  source ? local
  sink ! local
```

Why you might possibly want this done should be apparent: the local variable acts as a buffer in the data stream passing along the two channels. This copying process can be packaged as a named process that can be used to buffer any unbounded data stream passing between two processes

```
PROC buffer(CHAN source, sink) =
  WHILE TRUE
    VAR local :
    SEQ
      source ? local
      sink ! local             :
```

Now whereas the producer and the consumer process are tightly synchronized in a program like

```
PROC producer(CHAN output.stream) =
  WHILE TRUE
    VAR datum :
    SEQ
      ... calculate a new datum
      output.stream ! datum          :

PROC consumer(CHAN input.stream) =
  WHILE TRUE
    VAR datum :
    SEQ
      input.stream ? datum
      ... calculate using the datum    :

CHAN data.stream :
PAR
  producer(data.stream)
  consumer(data.stream)
```

with neither able to get ahead of the other, by adding a buffer

```
CHAN data.from.producer, data.to.consumer :
PAR
  producer(data.from.producer)
  buffer(data.from.producer, data.to.consumer)
  consumer(data.to.consumer)
```

the two are slightly decoupled. The producer is now able to run up to one item of data ahead of the consumer. (‘Magic buffers’, so called, that would allow the consumer to run an item ahead of the producer are a mite more difficult to implement, even in *occam*!)

More buffering is easily provided by inserting more buffers in the data path. This is like a ‘fall-through’ first-in-first-out store, where each item of data is passed along a register until it reaches the last unoccupied location. Several items can be in independent ‘free fall’ at once if the buffer is fairly empty.

```
CHAN data.stream[number.of.buffers + 1] :
PAR
  producer(data.stream[0])
  PAR index = [0 FOR number.of.buffers]
    buffer(data.stream[index], data.stream[index + 1])
  consumer(data.stream[number.of.buffers])
```

There is, of course, nothing to stop you programming a buffer with an array of variables governed by one process, just as in any conventional programming language.

Synchronization by control signals

You might try putting the buffer process into the stream that goes to the terminal screen from the “Bootifrolo” program

```
CHAN internal.stream :
PAR
  write.string(internal.stream, "Bootifrolo")
  buffer(internal.stream, terminal.screen)
```

but this is not quite right. The *buffer* initially performs well, and copies all of the characters to the screen. Eventually, however, all of the string has been sent, and the *write.string* process terminates. This leaves the *buffer* in a somewhat undignified state, trying to perform an input on *internal.stream* when there will never again be a corresponding output. The program is deadlocked.

Some way is needed of telling the buffer that it should not expect any more input along its *source* channel, and that it should accordingly terminate. The process

```
PROC copy.characters(CHAN source, end.of.source, sink)=
  -- Copy characters from source to sink
  -- until there is a signal on end.of.source
  VAR more.characters.expected :
  SEQ
    more.characters.expected := TRUE
    WHILE more.characters.expected
      VAR ch :
      ALT
        source ? ch
          sink ! ch
        end.of.source ? ANY
          more.characters.expected := FALSE    :
```

behaves just like the *buffer*, copying from *source* to *sink*, except that it may also take input from the channel *end.of.source*. The keyword `ANY` just means that the actual value received is immaterial, so it need not be stored in a variable. When an input signal is received on *end.of.source*, the variable *more.characters.expected* is set `FALSE`, so the `WHILE` loop terminates.

The right way of buffering the output of *write.string* is to send its output to a *copy.characters* process, and to send a termination signal after the whole string has been sent. Since the value received as a termination signal is ignored, it does not matter what is sent: outputting `ANY` has the effect of sending something unspecified.

```
CHAN internal.stream, end.of.internal.stream :
PAR
  SEQ
    write.string(internal.stream, "Bootifrolo")
    end.of.internal.stream ! ANY
  copy.characters(internal.stream,
                  end.of.internal.stream,
                  terminal.screen)
```

In the case of the *copy.characters* process, there is no need to use a signal on an extra channel, because there is already a channel going in the right direction. The data and the termination signal can be multiplexed onto one channel. You could select some value, say

```
DEF end.of.stream = -1
```

which is not a possible character value, and send that after the last real character of the message

```
DEF end.of.stream = -1 :
CHAN internal.stream :
PAR

  PROC write.string(CHAN output, VALUE string[]) =
    -- Send the characters of the string along output
    SEQ character.number = [1 FOR string[BYTE 0]]
      output ! string[BYTE character.number]         :
  SEQ
    write.string(internal.stream, "Bootifrolo")
    internal.stream ! end.of.stream

  PROC copy.characters(CHAN source, sink) =
    -- Copy characters from source to sink
    -- until a signal arrives on end.of.stream
    VAR more.characters.expected :
    SEQ
      more.characters.expected := TRUE
      WHILE more.characters.expected
        VAR ch :
        SEQ
          source ? ch
          IF
            ch <> end.of.stream
              sink ! ch
            ch = end.of.stream
              more.characters.expected := FALSE    :

  copy.characters(internal.stream, terminal.screen)
```

In this particular case, there is little to choose between the two styles: the latter program may be marginally more efficient.

In many cases there will be no data-stream going in the right direction, and a channel exclusively for synchronization will be essential. A circular buffer implemented using an array of variables is an example of this kind. Assume that the array is declared by

```
VAR datum[size] :
```

and that the variables

```
VAR reader, writer :
```

have values in the range zero to $size - 1$, so that the oldest value to leave the buffer will be found at $datum[reader]$, and the next to enter the buffer will be written to $datum[writer]$. It will be convenient to keep track of the number of unoccupied locations in the buffer by a further variable

```
VAR count :
```

whose value ranges from zero, for a full buffer, to *size* for an empty one.

There are two activities in which the buffer must be able to participate: provided that it is not full, that is that $count > 0$, it must be possible to add another value to the buffer

```
SEQ
  source ? datum[writer]
  writer := (writer + 1) \ size
  count  := count - 1
```

and provided that $count < size$ it must be possible for the oldest value to be read from the buffer

```
SEQ
  sink ! datum[reader]
  reader := (reader + 1) \ size
  count  := count + 1
```

The buffer must allow the producing and consuming processes to control its activity, selecting between writing and reading, provided only that there is room to write, or something to read, respectively.

Arbitration like this requires an `ALT` process, with the reading and writing actions being guarded by the conditions under which they may happen—$count > 0$ and $count < size$—and the readiness of the consumer and the producer. Since the consumer is ready when the buffer can output to *sink* and the producer when the buffer can input from *source* it is tempting to write

```
ALT
  count > 0   &   source ? datum[writer]
    SEQ
      writer := (writer + 1) \ size
      count  := count - 1
  count < size   &   sink ! datum[reader]
    SEQ
      reader := (reader + 1) \ size
      count  := count + 1
```

but output processes cannot be used to guard alternatives. The solution is to have a control signal from the consuming process indicating that it is ready to accept an input from *sink*. There is no need for the corresponding request before a write to the buffer, because the input along *source* serves perfectly well in place of a control signal.

```
PROC circular.buffer(CHAN source, request, sink) =

  -- Copy from source to sink, buffering up to size
  -- items. A signal is required on request before
  -- each item is read from sink.

  VAR reader, writer, count, datum[size] :
  SEQ
    reader := 0
    writer := 0
    count  := size
    WHILE TRUE
      ALT
        count > 0   &   source ? datum[writer]
          SEQ
            writer := (writer + 1) \ size
            count  := count - 1
        count < size   &   request ? ANY
          SEQ
            sink ! datum[reader]
            reader := (reader + 1) \ size
            count  := count + 1                    :
```

It is the responsibility of the consumer, whenever it reads from the buffer, to precede the input from *source* by an output on *request*, performing two communications in sequence

```
SEQ
  request ! ANY
  source ? ...
```

The ultimate consumer can be relieved of this burden at the expense of an extra process, executing concurrently with the circular buffer

```
PROC multiple.buffer(CHAN source, sink) =
  -- Copy from source to sink,
  -- buffering up to size+1 items.
  CHAN request, data :
  PAR
    circular.buffer(source, request, data)

    WHILE TRUE
      VAR datum :
      SEQ
        request ! ANY
        data ? datum
        sink ! datum                    :
```

The resulting *multiple.buffer* process has a behaviour which is indistinguishable from that of a chain of $size + 1$ single-item *buffer* processes acting in parallel.

Processes that evaluate expressions

Suppose now that you have a need to calculate the parity of the characters that are being sent to the terminal. (The parity of a character is an indication of whether there is an even or an odd number of one-bits in the binary representation of its code.)

In a language like PASCAL, you might write a function to calculate the parity of a character which was given to it as a parameter, but in *occam* (there being no 'function's) the natural construction is a named process

```
PROC calculate.parity(VALUE ch, VAR parity) =
  ... Return the parity of ch in parity
```

which returns the result by way of a `VAR` parameter. A representation will be necessary for parity values: I will choose the truth values

```
DEF even = TRUE, odd = NOT even :
```

although it turns out that the process will be independent of the particular bit pattern chosen to represent *even*.

Calculating the parity of *ch* involves considering each bit of *ch*: the simplest thing to do is to take them one at a time in sequence. (Expert bit twiddlers may care to code an algorithm logarithmic in the number of bits in the character.) The expression

```
ch /\ (1 << bit.number)
```

is either zero or not according as the *bit.number*$^{\text{th}}$ bit of *ch*, counting from zero at the least significant end. The loop

```
SEQ bit.number = [0 FOR number.of.bits.in.character]
  IF
    (ch /\ (1 << bit.number)) = 0
      SKIP
    (ch /\ (1 << bit.number)) <> 0
      parity := NOT parity
```

complements the value of *parity* as often as there are one bits in *ch*.

Since exclusive-or behaves like a conditional complement operation, the conditional process in the middle of this loop can be abbreviated to a simple assignment

```
parity := parity >< ((ch /\ (1 << bit.number)) <> 0)
```

with the same effect.

If *parity* is initialized to *even*, then its final value indicates the parity of *ch*, implementing the named process *calculate.parity*

```
DEF even = TRUE, odd = NOT even :

PROC calculate.parity(VALUE ch, VAR parity) =
  -- Return the parity of ch in parity
  DEF number.of.bits.in.character = 8 :
  SEQ
    parity := even
    SEQ bit.number=[0 FOR number.of.bits.in.character]
      parity := parity ><
                    ((ch /\ (1 << bit.number)) <> 0) :
```

Using parallelism as a tool for program modularity

If for some reason you wanted to modify the *write.string* process so that it wrote only the even parity characters from its argument, ignoring the rest, you could write

```
PROC write.even.parity.string(CHAN output,
                              VALUE string[]) =
  SEQ character.number = [1 FOR string[BYTE 0]]
    VAR parity :
    SEQ
      calculate.parity(string[BYTE character.number],
                                          parity)
      IF
        parity = even
          output ! string[BYTE character.number]
        parity = odd
          SKIP                                     :
```

This process is specialized: it performs its task well enough, but there are no recognizable separate components performing subtasks, which you might be able to use again in other programs. The code for selecting characters according to their parity is mixed in with the code for turning a string into a sequence of output processes.

A more modular program might use a process which turns a string into a sequence of outputs

```
PROC write.string(CHAN output, VALUE string[]) =
  ... Send the characters of string along output
```

and a separate process which splits a stream of characters into two streams according to their parities

```
PROC divide.on.parity(CHAN source, end.of.source,
                                even.sink, odd.sink) =
  ... Copy the even parity characters from source to
  ... even.sink, odd parity to odd.sink, until a signal
  ... arrives on end.of.source
```

This latter process can be used to filter out the odd or even parity character codes from any data stream.

```
PROC divide.on.parity(CHAN source, end.of.source,
                                even.sink, odd.sink) =
  -- Copy the even parity characters from source to
  -- even.sink, odd parity to odd.sink, until a signal
  -- arrives on end.of.source
  VAR more.characters.expected :
  SEQ
    more.characters.expected := TRUE
    WHILE more.characters.expected
      VAR ch :
      ALT
        source ? ch
          VAR parity :
          SEQ
            calculate.parity(ch, parity)
            IF
              parity = even
                even.sink ! ch
              parity = odd
                odd.sink ! ch
        end.of.source ? ANY
          more.characters.expected := FALSE           :
```

Since only the even parity stream is wanted in the present application, the unwanted stream must be discarded

```
PROC consume(CHAN source, end.of.source) =
  -- Discard input from source until a signal
  -- arrives on end.of.source
  VAR more.characters.expected :
  SEQ
    more.characters.expected := TRUE
    WHILE more.characters.expected
      ALT
        source ? ANY
          SKIP
        end.of.source ? ANY
          more.characters.expected := FALSE   :
```

Using the *divide.on.parity* component my “Bootifrolo” program might be written

```
PROC write.string(CHAN output, VALUE string[]) =
  ... Send the characters of string along output

PROC divide.on.parity(CHAN source, end.of.source,
                           even.sink, odd.sink) =
  ... Copy from source to even.sink or odd.sink

PROC consume(CHAN source, end.of.source) =
  ... Discard from source, until end.of.source

CHAN both.parities, end.of.both,
     odd.parity, end.of.odd.parity :
PAR
  SEQ
    write.string(both.parities, "Booting from Floppy")
    end.of.both ! ANY

  SEQ
    divide.on.parity(both.parities, end.of.both,
                     terminal.screen, odd.parity)
    end.of.odd.parity ! ANY

  consume(odd.parity, end.of.odd.parity)
```

In this particular case, the gain in modularity may not seem adequate to justify the expense, both in programming effort and execution time. The advantage is clearer in cases where the program must perform a number of tasks each of which divides its input data into chunks, and where the boundaries of these components do not coincide.

Using parallelism to resolve structure clash

A structure clash happens whenever a program must perform operations on data that must be divided into mutually overlapping components. In a text processing program, for example, it may prove necessary to do something to every line of a document, and something else to every sentence.

The natural way to code each of these tasks, individually, is to write programs whose structure reflects the structure of the document. To perform an action on every line:

```
WHILE ... there are more lines
  SEQ
    ... read a line
    ... process the line
```

and to perform an action on every sentence:

```
WHILE ... there are more sentences
  SEQ
    ... read a sentence
    ... process the sentence
```

Since sentences do not need to contain only complete lines, nor lines complete sentences, it is difficult to combine these two programs in a single sequential program. The somewhat unsatisfactory best that can be done in a sequential program is to treat the document as a sequence of words, these being the largest common sub-components of both lines and sentences.

```
WHILE ... there are more words
  SEQ
    ... read a word
    ... if it completes a line process the line
    ... if it completes a sentence process the sentence
```

In a parallel program, the structure of both component processes can be retained by performing the two divisions of the document concurrently

```
CHAN lines, sentences :
PAR
  ... copy the document to lines and sentences

  WHILE ... there are more lines
    SEQ
      ... read a line from lines
      ... process the line

  WHILE ... there are more sentences
    SEQ
      ... read a sentence from sentences
      ... process the sentence
```

The simplest case of a structure clash arises from attempting to pack data into fixed sized blocks that will not accommodate an exact whole number of items. It might be necessary, for example, to pack a stream of characters into half-kilobyte blocks for transmission or storage on a medium which accepts only such blocks. Consider first a case in which there is no structure clash: the medium is represented as a channel that accepts only slice outputs of half a kilobyte, and characters are represented by codes in the range from 0 to 255, so that a whole number of characters exactly fill a block.

The way to perform actions sequentially on the components of an array of bytes declared by

```
VAR buffer[BYTE bytes.in.a.block] :
```

is to use a sequential 'array' of processes created by the constructor

```
SEQ byte.number = [0 FOR bytes.in.a.block]
```

so this packing might be done by a process of the form

```
PROC pack.bytes.into.blocks(CHAN byte.source,
                           end.of.source, block.sink) =
  VAR more.bytes.expected :
  SEQ
    more.bytes.expected := TRUE
    WHILE more.bytes.expected
      VAR buffer[BYTE bytes.in.a.block] :
      SEQ
        SEQ byte.number = [0 FOR bytes.in.a.block]
          ALT
            more.bytes.expected &
                 byte.source ? buffer[BYTE byte.number]
              SKIP
            more.bytes.expected & end.of.source ? ANY
              more.bytes.expected := FALSE
            NOT more.bytes.expected & SKIP
              SKIP
        block.sink!buffer[BYTE 0 FOR bytes.in.a.block]:
```

The branch of the alternative that does all the work is the first, that guarded by an input from *byte.source* which inputs the next byte into the particular component of the buffer which is being considered. Since the guard does all the work, there is nothing left to be done in the guarded process, so this is SKIP. The condition before the SKIP guard ensures that it is ready when and only when there are no more bytes to be packed into the last block.

That process always sends a partly or completely empty block as its last output. The sending of a completely empty block could be prevented by looking ahead for the next byte:

```
DEF bytes.in.a.block = 512 :

PROC pack.bytes.into.blocks(CHAN byte.source,
                           end.of.source, block.sink) =
  -- Copy data from byte.source into complete blocks on
  -- block.sink, until there a signal arrives on
  -- end.of.source
  VAR next.byte :
  ALT
    byte.source ? next.byte    -- Read ahead first byte
      VAR more.bytes.to.pack :
      SEQ
        more.bytes.to.pack := TRUE
        WHILE more.bytes.to.pack
          DEF block.size  = bytes.in.a.block,
              buffer.size = block.size + 1  :
          VAR buffer[BYTE buffer.size] :
          SEQ
            buffer[BYTE 0] := next.byte
            SEQ byte.number = [1 FOR bytes.in.a.block]
              ALT
                more.bytes.to.pack &
                   byte.source?buffer[BYTE byte.number]
                  SKIP
                more.bytes.to.pack &
                   end.of.source ? ANY
                  more.bytes.to.pack := FALSE
                NOT more.bytes.to.pack & SKIP
                  SKIP
            block.sink ! buffer[BYTE 0 FOR block.size]
            next.byte := buffer[BYTE block.size]

    end.of.source ? ANY              -- No bytes at all
      SKIP                                          :
```

Even so, in case the entire message does not exactly fill a whole number of blocks, it has to be possible for a process that unpacks the characters from the blocks to deduce from those characters that it has reached the actual end of the character stream before the end of the last block.

Now consider the problem of trying to achieve a higher packing density, given that only character codes less than 128 are going to be sent, so that seven bits will suffice rather than eight. Seven bit values will not fit neatly into bytes, nor into half-kilobyte blocks. The problem can, however, be decomposed into two simpler separate problems in which there is no structure clash: turning seven bit character values into a sequence of bits, and packing a sequence of bits into blocks.

The packing of bits into blocks can be done in almost exactly the same way as that suggested for packing bytes into blocks. A byte can be considered to be an array of bits, indexed by using the bit-pattern manipulating operations. The assignment

```
buffer[BYTE byte.n] := buffer[BYTE byte.n] /\
                                 (NOT (1 << bit.n))
```

sets the *bit.n*$^{\text{th}}$ bit of the *byte.n*$^{\text{th}}$ byte of *buffer* to zero, whilst

```
buffer[BYTE byte.n] := buffer[BYTE byte.n] \/
                                     (1 << bit.n)
```

sets that same bit to one, so the conditional

```
PROC set.bit(VAR buffer[], VALUE byte.n, bit.n, bit) =
  IF
    bit = 0
      buffer[BYTE byte.n] := buffer[BYTE byte.n] /\
                                 (NOT (1 << bit.n))
    bit = 1
      buffer[BYTE byte.n] := buffer[BYTE byte.n] \/
                                     (1 << bit.n):
```

stores the given bit in the *bit.n*$^{\text{th}}$ bit of the *byte.n*$^{\text{th}}$ byte of the buffer. The buffer is being treated as a two-dimensional array, and the process that packs the buffer is a two-dimensional `SEQ-FOR` array of processes.

```
DEF bits.in.a.byte = 8, bytes.in.a.block = 512 :

PROC pack.bits.into.blocks(CHAN bit.source,
                           end.of.source, block.sink) =
  -- Copy data from bit.source in complete blocks onto
  -- block.sink, until there a signal arrives on
  -- end.of.source
  VAR next.bit :
  ALT
    bit.source ? next.bit   -- Read ahead the first bit
      VAR more.bits.to.pack :
      SEQ
        more.bits.to.pack := TRUE
        WHILE more.bits.to.pack
          DEF block.size = bytes.in.a.block :
          VAR buffer[BYTE block.size] :
          SEQ
            SEQ byte.number = [0 FOR bits.in.a.block]
              IF
                more.bits.to.pack
                  SEQ bit.number=[0 FOR bits.in.a.byte]
                    IF
                      more.bits.to.pack
                        SEQ
                          set.bit(buffer, byte.number,
                                  bit.number, next.bit)
                          ALT
                            bit.source ? next.bit
                              SKIP
                            end.of.source ? ANY
                              more.bits.to.pack:= FALSE
                      NOT more.bits.to.pack
                        SKIP
                NOT more.bits.to.pack
                  SKIP
            block.sink ! buffer[BYTE 0 FOR block.size]
    end.of.source ? ANY              -- No bits at all
      SKIP                                            :
```

Turning seven bit characters into a sequence of bits is also a simple task, since there is again no structure clash. The value of the expression

```
(character >> bit.number) /\ 1
```

is zero or one according to the value of the *bit.number*$^{\text{th}}$ bit of the value of character, so the character code can be treated as though it were an array of seven bits.

```
DEF bits.in.a.character = 7 :

SEQ bit.number = [0 FOR bits.in.a.character]
  bit.sink ! (character >> bit.number) /\ 1
```

The process to turn a stream of characters into a stream of the bits which make up their codes, least significant bit of the character first, has the same structure as the earlier unpacking processes:

```
DEF bits.in.a.character = 7 :

PROC unpack.bits.from.characters(CHAN char.source,
                                end.of.source, bit.sink) =
  -- Copy characters from char.source a bit at a time
  -- onto bit.sink, until a signal arrives on
  -- end.of.source
  VAR more.characters.expected :
  SEQ
    more.characters.expected := TRUE
    WHILE more.characters.expected
      VAR character :
      ALT
        char.source ? character
          SEQ bit.number = [0 FOR bits.in.a.character]
            bit.sink ! (character >> bit.number) /\ 1
        end.of.source ? ANY
          more.characters.expected := FALSE           :
```

Packing seven bit characters into half-kilobyte blocks is simply a matter of unpacking the characters into a stream of bits, and assembling the blocks from the stream of bits, the one activity in parallel with the other.

```
DEF bits.in.a.character =   7,
    bits.in.a.byte      =   8,
    bytes.in.a.block    = 512 :

PROC pack.characters.into.blocks(CHAN char.source,
                           end.of.source, block.sink) =

  PROC unpack.bits.from.characters(CHAN char.source,
                             end.of.source, bit.sink) =
    ... Send of character codes from char.source
    ... a bit at a time along bit.sink

  PROC pack.bits.into.blocks(CHAN bit.source,
                           end.of.source, block.sink) =
    ... Pack bits from bit.source into complete
    ... blocks and send them along block.sink

  CHAN bit.stream, end.of.bit.stream :
  PAR
    SEQ
      unpack.bits.from.characters(char.source,
                         end.of.source, bit.stream)
      end.of.bit.stream ! ANY

    pack.bits.into.blocks(bit.stream,
                         end.of.bit.stream, block.sink) :
```

Substantially the same program structure can clearly be used to turn the stream of blocks back into a stream of seven bit character codes, since that is just another, similar packing problem. The solution to each packing problem is of one of the three forms that I have shown here: grouping small objects to make larger ones; dividing large objects to make small ones; or a problem in which a structure clash requires that both the input data and the output data be divided into common sub-components.