10

Formatted input and output

One of the things you will probably miss in occam if you are used to writing programs in a conventional high-level language is the support for input and output of text. There are usually either language constructs or predefined routines which take the data from your program – that is, bit patterns representing strings, integers, floating-point numbers, and so on – and translate them into appropriate sequences of characters sent to terminals, printers or files. In the same way, there are usually built-in ways of analysing input from keyboard, file or card-reader and interpreting, for example, sequences of digits as the integer value to be stored in some variable. It is almost always possible to write input and output routines of your own but those provided for you will usually do very well.

Since occam is designed to be usable for applications where the program may be required to run in a 'stand-alone' environment, possibly on many processors, there can be no assumption of a standard operating system; the program must perform its own input and output translations. The *Reference Manual* describes a number of standard libraries of procedures and functions that an implementation will normally make available, including one providing low-level support for such conversions. This consists of a number of procedures which essentially take either a value and a string buffer in which to return a formatted representation, or a string and a variable into which to place the result of parsing the string as the representation of an object of some particular data type.

This chapter explores a possible implementation of these input and output translations, taking the alternative approach of converting directly between values and sequences of bytes communicated along a channel. Some procedures are introduced which are used in programs later in the book, including a higher-level routine which does multiple output translations according to a programmer supplied format.

Simple output

This section describes a number of procedures which convert the bit patterns of each of the primitive data types into canonical sequences of characters sent down a channel with the `BYTE` protocol. The particular representation chosen will result in a fixed number of characters being sent for every value of a given type. This is perhaps not the natural representation for everyday use, although it may simplify the tabulation of data; but it will be shown below how the fixed format of output can be massaged to fit one desired by the user.

The procedure for sending a single character (Process 10.1) is so simple that it is hardly worth writing.

Process 10.1

```
PROC write.character(CHAN OF BYTE output, VAL BYTE character)
  output ! character
:
```

Of course, a `BYTE` can also be interpreted as a numeric value, which you may wish to display as a decimal numeral, or indeed in hexadecimal notation. In this case, however, it is redundant to invest any mental effort or program code on the details of generating such a representation, since one may as well produce the same characters for a number in the range 0 to 255 – say, 137 – whether it is encoded internally as `137(BYTE)`, `137(INT)` or `137(INT64)`. The definitions of Process 10.2 and Process 10.3 become very simple once the routines for dealing with `INT64` values have been written.

Process 10.2

```
PROC write.decimal.byte(CHAN OF BYTE output, VAL BYTE number)
  write.decimal.int64(output, (INT64 number))
:
```

Process 10.3

```
PROC write.hexadecimal.byte(CHAN OF BYTE output, VAL BYTE bits)
  write.hexadecimal.int64(output, (INT64 bits))
:
```

A similar argument disposes of the other integer types, and the definition of the decimal output routines for these is as simple as in the `BYTE` case, provided the bit patterns are to be interpreted as signed, twos-complement numbers. In representing the unsigned value corresponding to the bit pattern, as is probably desirable for the hexadecimal case, a complication arises. If `#FFFF(INT16)` is to produce only four ‘F’s, no matter how many digits are displayed, then it must be treated differently from `(INT64 #FFFF(INT16))`, which is, of course, `#FFFFFFFFFFFFFFFF(INT64)`. Indeed, it should be treated like `#FFFF(INT64)` instead. In order to effect this transformation the range of bits which are significant in the longer data type must be restricted; the sign-extended value must be trimmed by a `BITAND` with the mask which has as many low-order bits set as fit into the shorter type.

How should this mask be denoted? In each integer type the bit pattern with all bits set represents -1; but the operands of `BITAND` must be of the same type, and it is precisely -1 in the shorter type that we have seen to change bit pattern when it is coerced to the longer. The bit pattern could be hand-coded into each routine, or could be generated by shifting `-1(INT64)` right by the difference in lengths of the two types; but these are *ad hoc* solutions. Neither, for instance, would allow a machine-independent expression for the correct mask in the `INT` case. It is preferable to let sign extension, which causes the problem, also solve it: `MOSTNEG INT16` has only the sign bit set, so `INT64 (MOSTNEG INT16)` has all the bits that are not wanted set, and also the next most significant. Thus shifting this left gives the complement of the mask needed for Process 10.4.

Process 10.4

```
PROC write.hexadecimal.int16(CHAN OF BYTE output, VAL INT16 bits)
  VAL INT64 mask IS (INT64 (MOSTNEG INT16)) << 1 :
  write.hexadecimal.int64(output, (INT64 bits) /\ (BITNOT mask))
:
```

The other integer types can be dealt with after the same fashion.

This leaves us with the task of converting `INT64` values into decimal and hexadecimal representations. The hexadecimal case is very straightforward: Process 10.5 transmits the representation of each four bits in turn.

Process 10.5

```
PROC write.hexadecimal.int64(CHAN OF BYTE output, VAL INT64 bits)
  VAL []BYTE hex.digit IS "0123456789abcdef" :
  SEQ i = 1 FOR 64 / 4
    VAL nibble IS INT ((bits >> (64 - (4 * i))) /\ #F(INT64)) :
    output ! hex.digit[nibble]
:
```

A standard technique for outputting a decimal number relies on the observation that, for a positive n and a power of ten $tens$, the value of $(n/tens)\backslash 10$ is the coefficient of $tens$ in the decimal expansion of n. This can be converted to the corresponding digit either by subscripting a string or by the arithmetic trick of Process 10.6, which relies on the digit characters having consecutive codes.

Process 10.6

```
BYTE FUNCTION decimal(VAL INT d) IS BYTE (d + (INT '0')) :
```

This gives rise to an algorithm which performs the digit calculation for each power of ten not exceeding n, and the results are produced in sequence in decreasing order of their weight.

```
INT64 tens :
SEQ
  tens := 1(INT64)
  WHILE (n / tens) >= 10(INT64)
    tens := 10(INT64) * tens
  WHILE tens <> 0(INT64)
    SEQ
      output ! decimal(INT ((n / tens) \ 10(INT64)))
      tens := tens / 10(INT64)
```

This carefully calculates the minimum power of ten required, and so yields the shortest possible sequence of bytes along *output*. For a fixed length sequence of bytes, to simplify further manipulations, it suffices to replace the first occurrence of n by `MOSTPOS INT64` – or, as the effect of the first `WHILE` is then constant, just to initialize *tens* to the appropriate power of ten, which calculation shows to be 10^{18}.

That process works for all positive n and, as a special case, for zero. To extend it to output negative numbers as well, it is tempting to try changing the sign first; but this is wrong, because to change the sign of `MOSTNEG INT64` with normal arithmetic negation is an error, and `MINUS (MOSTNEG INT64)` is just `MOSTNEG INT64` again. The standard, if perhaps confusing, solution is to treat positive numbers as special cases which are best output by making them negative, or equivalently to change the sign of *tens* so that the result of dividing by *tens* is consistently negative. This gives us Process 10.7. Notice that it is a matter of the definition of the division and remainder operators in occam that changing the sign either of *tens* or of *number* just changes the sign of the expression $(number/tens)\backslash 10$. The division of *tens* by ten always gives an exact answer, excepting the final occasion, when *tens* is one, and the result of the division is zero.

A procedure which sends the characters of a string down a channel (Process 10.8) has already been seen, in Chapter 5. This can be used to send appropriate representations of the Boolean values with Process 10.9.

There remains the task of producing representations of floating-point values. This is a problem of much greater complexity than for the discrete data types, and it is quite possible to write at great length on algorithms for producing the right sort of accuracy. The difficulties arise from the fact that to represent an N-bit binary mantissa *exactly* takes N decimal digits (as two is not a factor of five), but mantissas differing only in their least significant bit differ in their decimal representation in about the $(N/\log_2 10)^{\text{th}}$ digit, so over two-thirds of the exact representation's apparent accuracy is illusory.

Process 10.7

```
PROC write.decimal.int64(CHAN OF BYTE output, VAL INT64 number)

  -- Write a signed decimal representation of number
  -- to the output channel

  VAL INT64 ten.to.the.eighteen IS 1000000000000000000(INT64) :
  INT64 tens :
  SEQ
    IF
      number < 0(INT64)
        SEQ
          output ! '-'
          tens :=  ten.to.the.eighteen
      number >= 0(INT64)
        SEQ
          output ! '+'
          tens := -ten.to.the.eighteen
    WHILE tens <> 0(INT64)
      SEQ
        output ! decimal(INT (-((number / tens) \ 10(INT64))))
        tens := tens / 10(INT64)
:
```

Process 10.8

```
PROC write.string(CHAN OF BYTE output, VAL []BYTE string)
  SEQ i = 0 FOR SIZE string
    output ! string[i]
:
```

Process 10.9

```
VAL []BYTE true.string  IS "true" :
VAL []BYTE false.string IS "false" :

PROC write.bool(CHAN OF BYTE output, VAL BOOL condition)
  IF
    condition
      write.string(output, true.string)
    NOT condition
      write.string(output, false.string)
:
```

The floating-point number is, after all, only approximating the 'correct' real value to within about half a least significant bit.

Analysis identifies the accuracy to aim for, but this cannot always be attained using the same format of floating-point arithmetic to scale the mantissa associated with a power-of-two exponent to obtain one corresponding to a power-of-ten exponent. This problem can be solved; see for example [15]. Although we could code the arithmetic in occam, the necessary algorithms are both long and intricate and not central to our presentation, so are omitted from this book. The following code assumes the existence of functions

```
INT, INT64 FUNCTION split.real32(VAL REAL32 r)
  ...
:
INT, INT64 FUNCTION split.real64(VAL REAL64 r)
  ...
:
```

which when applied to a floating-point number r of the appropriate type return a pair $\langle exponent, mantissa \rangle$ such that

$$r = m \times 10^{exponent} \text{ and } m = exponent = 0 \ \vee \ 0.1 \leq |m| < 1.0$$

$$mantissa \simeq m \times 10^{N} \text{ where } N = \begin{cases} 9, & \text{if } r \text{ is a } \texttt{REAL32} \\ 17, & \text{if } r \text{ is a } \texttt{REAL64} \end{cases}$$

for some real number m. It is then a simple matter to output the *exponent* and *mantissa* as integers, with suitable punctuation.

Simple input

Constructing a data object from its textual representation is slightly more difficult because, in general, not all sequences of characters will be legal representations. For example, a process to read a numeral might expect some spaces, perhaps a sign and some more spaces, and then a sequence of digits, followed by something else. If there are no digits, or if the number represented is too large to be encoded as a bit pattern of the appropriate type, then an error has occurred.

The particular action to be taken to recover from an error depends on the circumstances of the conversion: for example, whether the digits are being read from a terminal keyboard, or a magnetic tape, whether the process is running in a desk-top microcomputer or in aircraft autopilot equipment. For a general purpose routine, it is reasonable to return a Boolean indication of whether the conversion was successful. (Other indications might be possible, for example, a communication on a channel laid aside for indicating errors.)

Ignoring, for the present, the matter of the sign, and the possibility of error, a sequence of digits can be converted into a bit pattern representing the same number by

```
BYTE char :
SEQ
  n := 0
  input ? char
  WHILE ('0' <= char) AND (char <= '9')
    SEQ
      n := (10 * n) + ((INT char) - (INT '0'))
      input ? char
```

The arithmetic is essentially dual to that in the output routine. If *char* is the character code of a digit, then it lies between the codes of the characters zero and nine, and ($char -$ `'0'`) is its value as a digit, having interpreted the `BYTE` values as integers in order to perform arithmetic on them.

As in the case of output, it is necessary to be careful with the most negative integer: it will not do to read negative numerals by reading the digits as if they were those of a positive numeral and then changing the sign of the result. Furthermore, to avoid overflow the new value of n must be compared with either the most positive or the most negative bit pattern, again being careful to keep all the arithmetic in the expressible range

```
VAL INT digit IS (INT char) - (INT '0') :
IF
  (sign = '+') AND (n <= (((MOSTPOS INT) - digit) / 10))
    n := (10 * n) + digit
  (sign = '-') AND ((((MOSTNEG INT) + digit) / 10) <= n)
    n := (10 * n) - digit
  otherwise
    ok := FALSE    -- an error has occurred
```

A possible solution to the problem of errors would be to omit the third branch of the conditional entirely, so that the routine would stop in case of an overflow. This more general solution postpones the decision, giving the caller of the process the option of ignoring the error, or of acting on it in any way he chooses, including the option of stopping.

The appendix contains a routine complementary to the decimal output routines which has the following specification:

```
PROC read.signed(CHAN OF BYTE input, INT n, BOOL ok)
  -- Read an (optionally signed) decimal numeral from
  -- the input returning the corresponding value in n,
  -- and TRUE in ok precisely if the conversion succeeded
  ...
:
```

In many programming languages a routine like *read.signed* could only be used for conversion of a numeral being read from a peripheral device. In PASCAL, for example, such a routine would be reading from a file, but an entirely different routine would be needed to convert a numeral stored in an array of characters. In occam, there is nothing to stop you doing this by putting input and output routines together in parallel. The process

```
CHAN OF BYTE internal :
PAR
  write.string(internal, "-137*C")
  read.signed(internal, n, ok)
```

sets n to -137. This might not look very useful for constant strings, but the same can be done with variable arrays of characters. This means, for example, that it is easy to separate the business of line construction, editing and echoing when reading from a terminal, from whatever data conversion you might want to perform on the input.

For completeness, the appendix also contains a coding of a line construction process suitable for input from a VDU

```
PROC read.line(CHAN OF BYTE keyboard, echo, []BYTE s, INT n)
  -- Construct a string in [s FROM 0 FOR n] from the
  -- printable characters read from keyboard and sent
  -- back on the echo channel.
  ...
:
```

As it reads characters from the *keyboard* stream, this process packs them into the byte array s and copies them to the *echo* stream, allowing the usual sort of line editing. For example, if the *char* typed on *keyboard* is backspace the last character in the line is cancelled, and its echo is removed from the screen by writing a blank in its place:

```
CASE char
  ...
  backspace
    IF
      n > 0
        SEQ
          echo ! backspace
          echo ! '*s'
          echo ! backspace
          n := n - 1
      n = 0            -- nothing to delete
        echo ! bell
  ...
```

Multiple output

Most high-level languages provide some way of combining the output of various pieces of data into a single command. In PASCAL this leads to an exceptional violation of the language's rules that all procedures have a fixed number of arguments of fixed types: rather than requiring the sequence of commands

```
BEGIN
  write_string(output, 'the square root of ');
  write_integer(output, n);
  write_string(output, ' is ');
  write_real(output, sqrt(n))
END
```

the first relaxation is that all the procedures are called *write*, and then you are allowed to substitute a single procedure call

```
write(output, 'the square root of ', n, ' is ', sqrt(n))
```

There are rightly no such exceptions in occam.

The C language is another which makes output formatting easy by providing a routine *printf* which takes as one of its arguments a control string that describes how to interpret the bit patterns of the other parameters. A call like

```
printf("The square root of %5d is %E", n, sqrt((float) n));
```

says that n is to be interpreted as an integer, and output in a field of width five characters, and $sqrt((float)\,n)$ is a floating-point number to be output in a default exponential format. If n is an integer which has the value 1936 then the resulting output will be

```
The square root of  1936 is 4.400000E+01
```

(If n happened to be a floating-point variable the first number printed would be that obtained in occam by `VAL INT n RETYPES n`!) This section develops the full expressive power of *printf* within the safe, restrictive occam philosophy.

In occam, a procedure certainly cannot have a variable number of arguments; so how can the required flexibility be obtained? It would be possible, but inelegant, to declare a procedure which took as many arguments as we can imagine ever possibly wanting to use, and supplying zero or some other value as the actual parameter in the places not needed by a particular call. Another possibility would be to build the arguments into a table, and allow the body of the procedure to work out how many there were using `SIZE`. But neither of these allows objects of more than one type to be passed: the formal parameter specifier fixes the type of each individual argument in the first case, and of all the arguments *en masse* in the second. To pass items of different type would need some kind of variant record or union type, which is not provided in occam. But as was remarked in Chapter 3 there is an occam 2 analogue, the discriminated protocol; moreover, a variable and unbounded number of items may be communicated along a channel. The solution, therefore, is to declare a protocol DATA.ITEM and write a procedure *write.formatted* with three parameters:

```
PROC write.formatted(CHAN OF BYTE output,
                     VAL []BYTE control,
                     CHAN OF DATA.ITEM data )
  ...
:
```

What are the variants that DATA.ITEM must allow? One tag is required for each primitive type, if we want to be able to set different default field widths for the different length data types; *printf* also allows strings, which would naturally correspond to a counted-array protocol. One other item that may prove useful, especially if we use the same protocol in an input filter, is a tag to mark abnormal termination of the formatting operation.

```
PROTOCOL DATA.ITEM
  CASE
    data.bool;   BOOL
    data.byte;   BYTE
    data.int;    INT
    data.int16;  INT16
    data.int32;  INT32
    data.int64;  INT64
    data.real32; REAL32
    data.real64; REAL64
    data.string; INT::[]BYTE
    data.abort
:
```

Using this procedure and protocol, the earlier example can be coded as

```
CHAN OF DATA.ITEM data :
PAR
  write.formatted(output,
                  "The square root of %5d is %E", data)
  SEQ
    data ! data.int; n
    data ! data.real32; sqrt(REAL32 ROUND n)
```

The control string, to follow the model of *printf* reasonably closely, will be an array of characters most of which will simply be sent down the *output* channel. At every place in the sequence where a '%' is encountered, a number of characters taken together will describe the formatting to apply to an item input along *data*, and the result of this conversion will be inserted into the output stream. If the hard work is abstracted into a procedure, *process.item*, the required routine is then Process 10.10.

Process 10.10

```
PROC write.formatted(CHAN OF BYTE output,
                     VAL []BYTE control,
                     CHAN OF DATA.ITEM data )

  PROC process.item(CHAN OF DATA.ITEM data,
                    VAL []BYTE control, INT i,
                    CHAN OF BYTE output       )
    -- interpret characters from [control FROM i FOR ...]
    -- read and format item accordingly from data and send
    -- along output; set i beyond last character of format
    ...
  :
  INT i :
  SEQ
    i := 0
    VAL INT size IS SIZE control :
    WHILE i < size
      IF
        control[i] <> '%'
          SEQ
            output ! control[i]
            i := i + 1
        control[i] = '%'
          SEQ
            i := i + 1
            process.item(data, control, i, output)
:
```

The individual format descriptions are analysed into four parts, most of which can be omitted to obtain default values. They are: flag characters, determining whether the field should be left-justified, whether positive numbers should appear preceded by a '+', or by an extra space; a decimal numeral representing the minimum field width; a dot followed by a decimal numeral denoting some notion of *precision*, which varies according to the type of format; and a non-optional character defining the type of format.

As an extra sophistication either numeral in the format description may be replaced by an asterisk, which causes another item (which must be an `INT`) to be input first, to supply the number required. The coding of *process.item* will require the format parameters to be passed to subsidiary routines. In order to avoid having to supply a large number of arguments to each routine, the information will be packaged as a pair of arrays, with named constants to index them:

```
VAL INT left.justify IS 0 :
VAL INT    sign.plus IS 1 :
VAL INT   sign.blank IS 2 :
VAL INT number.of.flags IS 3 :
[number.of.flags]BOOL flag :

VAL INT  get.flag IS -1 :
VAL INT     width IS  0 :
VAL INT precision IS  1 :
VAL INT field.numbers IS 2 :
[field.numbers]INT field :
```

The body of the procedure is then a `WHILE` loop which scans through the control string, keeping track of which of the first three parts of the format it has already found present or omitted, and processing characters which make sense as a component of one of the remaining ones. When it reaches some other character – or runs off the end of the string – the loop terminates and the procedure tries to interpret the latest character as a format type (Process 10.11).

The actual acceptance of data from the source stream and transmission of the textual representation along *output* is delegated to a routine

```
PROC formatted.output(VAL BYTE format,
                      VAL []BOOL flag, []INT field,
                      CHAN OF DATA.ITEM data,
                      CHAN OF BYTE output          )
  ...
:
```

Process 10.11 always invokes this routine, even when errors have been detected. It would not have been unreasonable to omit the final branches of each of the `IF` processes in *process.item*, and so cause the whole to stop at any of those errors in the control string. Falling out of the `WHILE` loop, however, with *format.char* containing the offending character, which cannot be valid as a format designator, gives the option of generating a message explaining the error. It would be possible to handle the other tagged variants on the input for a '`*`' similarly, setting *this.format* false or coercing the value to an `INT` where possible; but the user process could still cause a deadlock by refusing to communicate. In this example, for simplicity, erroneous communication will cause the natural deadlock.

Before proceeding to the details of the *formatted.output* procedure, there remain one or two other gaps in Process 10.11 which are straightforward

Process 10.11

```
PROC process.item(CHAN OF DATA.ITEM data,
                  VAL []BYTE control, INT i,
                  CHAN OF BYTE output       )
  ... declare arrays, constants and PROC formatted.output
  INT which.field :
  BOOL this.format :
  BYTE format.char :
  VAL INT size IS SIZE control :
  SEQ
    flag := [FALSE, FALSE, FALSE]
    field := [-1, -1]
    which.field, this.format := get.flag, TRUE
    format.char := '*#00' -- not a valid format type
    WHILE this.format AND (i < size)
      SEQ
        format.char := control[i]
        CASE format.char
          '-', '+', '*s'
            IF
              which.field = get.flag
                ... set appropriate flag
              which.field > get.flag
                this.format := FALSE  -- error case
          '.'
            IF
              which.field < precision
                which.field, field[precision] := precision, 0
              which.field >= precision
                this.format := FALSE  -- error case
          '**'
            IF
              (which.field < width) AND (field[width] < 0)
                SEQ
                  which.field := width
                  data ? CASE data.int; field[width]
                  this.format := field[width] >= 0  -- error?
              (which.field = precision) AND (field[precision] < 0)
                SEQ
                  data ? CASE data.int; field[precision]
                  this.format := field[precision] >= 0  -- error?
              otherwise
                this.format := FALSE  -- error case
          '0', '1', '2', '3', '4', '5', '6', '7', '8', '9'
            ... accumulate number for width or precision
          ELSE
            this.format := FALSE  -- parsed format successfully
        i := i + 1
    formatted.output(format.char, flag, field, data, output)
:
```

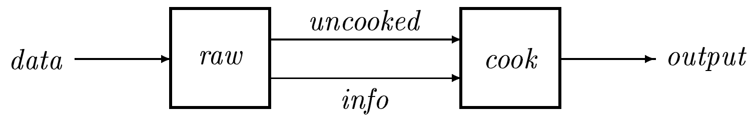
to fill in. These are setting the flags, which is a simple case discrimination on the flag character

```
-- set appropriate flag
CASE format.char
  '-'
     flag[left.justify] := TRUE
  '+'
     flag[sign.plus] := TRUE
  '*s'
     flag[sign.blank] := TRUE
```

and accumulating the numbers, taking care to ensure arithmetic overflow does not occur

```
-- accumulate number for width or precision
VAL INT digit IS (INT format.char) - (INT '0') :
CASE which.field
  get.flag
    which.field, field[width] := width, digit
  width, precision
    INT parameter IS field[which.field] :
    IF
      parameter <= (((MOSTPOS INT) - digit) / 10)
        parameter := (parameter * 10) + digit
      parameter >  (((MOSTPOS INT) - digit) / 10)
        this.format := FALSE  -- error case
```

The bulk of the remaining work is in the code to implement recognized formats. This can be separated into at least two tasks: accepting an appropriate value from *data* and converting it into a sequence of bytes; and massaging that sequence into the one specified by the values of *flag* and *field*. In some cases the default values to supply for the *field* settings and so on depend on the type of value input: for instance the hexadecimal output for an `INT16` is 'naturally' made up of four digits, while that of an `INT64` needs all sixteen; so a channel is required to transfer miscellaneous information. This gives the scheme



for the non-error case of Process 10.12. The capital letter format characters represent the same formatting instructions as the corresponding lower-case

Process 10.12

```
PROC formatted.output(VAL BYTE format,
                      VAL []BOOL flag, []INT field,
                      CHAN OF DATA.ITEM data,
                      CHAN OF BYTE output          )

  PROC raw.output(CHAN OF DATA.ITEM data,
                  VAL BYTE format,
                  CHAN OF INT info,
                  CHAN OF BYTE uncooked )
    ...
  :

  PROC cook.output(CHAN OF BYTE uncooked,
                   CHAN OF INT info,
                   VAL BYTE format,
                   VAL []BOOL flag, []INT field,
                   CHAN OF BYTE cooked          )
    ...
  :

  VAL BYTE format.type IS lower.case(format) :
  CASE format.type
    '%', 'b', 'c', 'd', 'e', 'f', 'x'
      CHAN OF BYTE uncooked :
      CHAN OF INT info :
      PAR
        raw.output(data, format.type, info, uncooked)
        cook.output(uncooked, info, format, flag, field, output)
    ELSE  -- unrecognized format character
      SEQ
        write.string(output, "<format error>")  -- e.g.
        STOP
:
```

ones, but with any alphabetic characters in the output forced into capitals. (This affects the larger hexadecimal digits and the exponent sign, as well as the strings for the Boolean values and character output.) The same raw output will be generated for both, and the translation will form part of the cooking process. This explains the use of the function *lower.case*, which is given in Process 10.13; the function *upper.case* (Process 10.14), which we will need later, is similar except that it modifies lower-case letters.

The production of the basic character sequence by the *raw.output* routine (Process 10.15) is a matter of a case discrimination on the format character, in each case performing a discriminating input on *data*, converting appropriate variants to the canonical sequence of characters along *uncooked*, preceeded by size information along *info* where needed. The one exception

Process 10.13

```
BYTE FUNCTION lower.case(VAL BYTE char)
  BYTE result :
  VALOF
    IF
      ('A' <= char) AND (char <= 'Z')
        result := BYTE ((INT char) >< ((INT 'A') >< (INT 'a')))
      (char < 'A') OR ('Z' < char)
        result := char
    RESULT result
:
```

Process 10.14

```
BYTE FUNCTION upper.case(VAL BYTE char)
  BYTE result :
  VALOF
    IF
      ('a' <= char) AND (char <= 'z')
        result := BYTE ((INT char) >< ((INT 'A') >< (INT 'a')))
      (char < 'a') OR ('z' < char)
        result := char
    RESULT result
:
```

to this is where a repeated '%' stands for itself. Any type-dependent size information that is needed, for instance the number of significant figures produced by the *split.real*— functions or the number of hexadecimal characters required for a value of some type

```
VAL INT real32.precision IS 9 :
VAL INT real64.precision IS 17 :
VAL INT nibbles.in.a.byte IS 2 :
```

is sent along *info* to act as default values for the *field* parameters. The canonical sequence of characters corresponding to the input value is then sent along *uncooked*.

The *raw.output* process does not generate any explicit termination signal for the cooking process. Instead, the latter must know how many characters to expect. By the construction of the simple routines, the canonical raw text in most cases has a length determined purely by the format character:

```
VAL INT decimal.raw IS 1+19 : -- sign and digits from decimal
VAL INT hexadecimal.raw IS 16 : -- and hexadecimal raw output
```

Only in the Boolean case is the length of the character sequence dependent on the actual value to be represented, as *true.string* and *false.string* will in

Process 10.15

```
PROC raw.output(CHAN OF DATA.ITEM data, VAL BYTE format,
                CHAN OF INT info, CHAN OF BYTE uncooked )
  CASE format
    '%'  -- percent sign
      write.character(uncooked, '%')
    'b'  -- Boolean
      ... accept data.bool for write.bool after 'info'
    'c'  -- character
      BYTE character :
      SEQ
        data ? CASE data.byte; character
        write.character(uncooked, character)
    'd'  -- decimal (signed) integer
      ... as for 'x' using write.decimal.<type> and no 'info'
    'e' -- exponential form of real
      INT exponent :
      INT64 mantissa :
      SEQ
        data ? CASE
          REAL32 float :
          data.real32; float
            SEQ
              exponent, mantissa := split.real32(float)
              info ! real32.precision
          REAL64 float :
          data.real64; float
            SEQ
              exponent, mantissa := split.real64(float)
              info ! real64.precision
        write.decimal.int64(uncooked, mantissa)
        write.decimal.int(uncooked, exponent)
    'f'  -- positional form of real
      ... as 'e' but write mantissa and 'info ! exponent' in PAR
    'x' -- hexadecimal (unsigned) integer
      data ? CASE
        BYTE number :
        data.byte; number
          SEQ
            info ! nibbles.in.a.byte
            write.hexadecimal.byte(uncooked, number)
        ... similarly for INT16, INT32 and INT64
        INT number :
        data.int; number
          SEQ
            VAL []BYTE count.bytes RETYPES number :
            info ! (SIZE count.bytes) * nibbles.in.a.byte
            write.hexadecimal.int(uncooked, number)
:
```

general have different lengths. But with only two values, and the *info* channel available, this causes no difficulty:

```
-- accept data.bool for write.bool after 'info'
BOOL condition :
SEQ
  data ? CASE data.bool; condition
  IF
    condition
      info ! SIZE true.string
    NOT condition
      info ! SIZE false.string
  write.bool(uncooked, condition)
```

Notice that should some other textual representation of the Boolean values be required throughout, then it is sufficient to change the two definitions

```
VAL []BYTE true.string  IS "gwir" :
VAL []BYTE false.string IS "gau" :
```

and the rest of the code automatically adapts to compensate.

Arranging the text

The output-cooking process has several jobs to do, the precise selection depending on the format type and the values of the flags. The tasks potentially required for the integer formats are typical:

- the number of characters to expect must be determined, and unspecified values of the *field* array filled in; in the hexadecimal case the default must be input from the *info* channel even if the precision has been specified

- those leading zeros not demanded by the *field*[*precision*] must be suppressed, and the remaining digits (preceded by a sign, if required) must be fitted, right-justified, into a sequence of no less than *field*[*width*] number of bytes

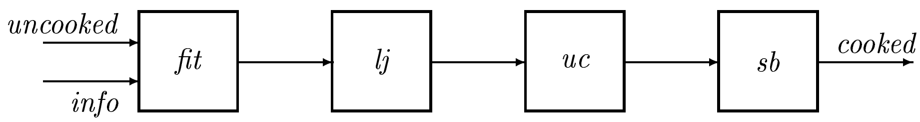
- if *flag*[*left.justify*] is set, the result must be left-justified, perhaps by counting leading spaces and retransmitting them after the significant text

- in an `'X'` format, letters in the hexadecimal representation must be translated to upper case

- *flag*[*sign.blank*] may require a plus sign to be replaced by a blank.

The main difference in the other formats is the meaning attached to the *precision* field: this specifies the minimum number of digits for the integer formats, but the maximum number of significant digits for floating-point numbers, and the maximum number of characters to print from the textual representation of Boolean values.

The first two of the listed tasks naturally coalesce; the one establishes the boundaries within which the second operates. Conceptually, therefore, there might be a chain of processes



each passing on the result of its modifications to the next. There are a number of disadvantages to this scheme, however.

If for instance, as will often be the case, none of the three later features is selected, then all the processes bar the first will be acting as simple buffers, with no benefit in exchange for the consequent overhead of frequent rescheduling. This could be overcome by setting up the pipeline including just those elements required, although the resulting procedure would be relatively complex and hard to comprehend. Moreover, since the *field*[*width*] specification is only the minimum number of bytes to generate for this item – neither truncation nor replacing with '*'s seems an adequate way of dealing with 'format overflow' when the channel can accept a stream of unlimited length – the output from the field-fitting stage would be of unpredetermined length; and so there would be the added complexity of organizing the distributed termination of the parallel components.

It will therefore be desirable to collapse the pipeline (distributed in 'space') into a sequential process (distributed in time). The following remark suggests that this is possible without having to read all the data from *uncooked* before producing any on *cooked*: in each numeric format the sign character and ultimate spacing are all determinable at the moment that the first non-zero digit, if any, is received on the *uncooked* channel; and the only state required to determine them is the sign and the position of that digit in the stream. For since the input stream is of known length, by construction, and the punctuation for the format is fixed, it is easy to tell how many significant characters there will be; and thence how many spaces and zeros are required to meet the specification of *field* and *flag*. Any initial text can then be sent, the remaining characters copied (with translation if requested), and finally any left-justifying spaces complete the output for that item. Each character of the non-numeric formats is significant, and so the number of significant characters is already known and the same argument applies.

Process 10.16

```
PROC cook.output(CHAN OF BYTE uncooked, CHAN OF INT info,
                 VAL BYTE format, VAL []BOOL flag, []INT field,
                 CHAN OF BYTE cooked                            )

  VAL BYTE format.type IS lower.case(format) :

  INT raw.size, text.size :
  BOOL signed :
  BYTE sign.char, first :
  SEQ
    CASE format.type
      '%', 'c'
        raw.size, field[precision] := 1, 1
      'b'
        ... set up Boolean
      'd'
        raw.size := decimal.raw
      'e', 'f'
        ... set up approximate real
      'x'
        ... set up hexadecimal


    CASE format.type
      '%', 'b', 'c'
        SEQ
          text.size, signed := raw.size, FALSE
          uncooked ? first
      'd', 'e', 'f'
        VAL BOOL plus  IS flag[sign.plus] :
        VAL BOOL blank IS flag[sign.blank] :
        ... signed numeral
      'x'
        ... unsigned numeral


    VAL BOOL left  IS flag[left.justify] :
    VAL BOOL right IS NOT left :
    VAL BOOL upper IS format <> format.type :
    INT zero.pad, space.pad, copy, truncate :
    CASE format.type
      '%', 'b', 'c', 'd', 'x'
        ... straightforward cases
      'e'
        ... send exponent before final spaces
      'f'
        ... may need leading or trailing zeros
:
```

Since the formatting tasks differ for different format characters, the body of *cook.output* could be written as a case discrimination on the format type. There are, however, some tasks which can be done the same way for a number of formats, but the sharing differs from task to task. This can be exploited by placing three case discriminations in sequence (Process 10.16). The first determines the number of characters to be expected; the second determines the number of significant digits, reading the first, and also decides about any sign; the last performs the actual output.

The three elided cases in the first discrimination all share a need to accept a default value on the *info* channel and set the precision if it has been left unspecified. As to require no characters at all would be bizarre, a precision specified as zero will be taken as an explicit request for the default. Process 10.17 performs the conditional assignment.

Process 10.17

```
PROC set.unless.specified(INT parameter, VAL INT default)
  IF
    parameter >  0  -- already specified
      SKIP
    parameter <= 0  -- left to default
      parameter := default
:
```

Using this routine, the initialization task for the Boolean case is

```
-- set up Boolean
INT max.chars IS field[precision] :
INT default.precision :
SEQ
  info ? default.precision
  set.unless.specified(max.chars, default.precision)
  raw.size, max.chars :=
       default.precision, min(max.chars, default.precision)
```

The other two cases are similar, setting *raw.size* to either *decimal.raw* or *hexadecimal.raw*. In the cases where the precision is an upper limit, the default is the maximum value it may take; for hexadecimal numerals it may cause padding with leading zeros, so the restriction of *field*[*precision*] to at most *default.precision* is omitted.

At the start of the second case discrimination *raw.size* characters representing the item being processed remain to be read from *uncooked*. By the end of it, *text.size* contains the number of significant characters, including a *first* which has been read already; that a *sign.char* is to be displayed is separately encoded by *signed* being true. In the non-numeric formats there is no sign, and all characters are significant, so this is simply established by reading *first* and setting the variables as shown in Process 10.16.

Process 10.18

```
PROC get.first.significant(CHAN OF BYTE input,
                           INT number.of.digits, BYTE digit )
  SEQ
    input ? digit
    WHILE (number.of.digits > 1) AND (digit = '0')
      SEQ
        input ? digit
        number.of.digits := number.of.digits - 1
:
```

For numerals insignificant leading zeros must be discarded. If the *number.of.digits* parameter to *get.first.significant* (Process 10.18) is initially the number of digits to come, each time round the loop it is the number left to read plus one for the cached *digit*; at the end these are the values required for *text.size* and *first*. The loop ensures the return of a count of one and the character `'0'`, respectively, in the case of a stream representing zero.

The decimal numerals are preceded by their sign which must be input from *uncooked*. This sign will eventually be printed if it is minus, or if it is explicitly requested by *plus* or *blank* flags. The integer format must also ensure that no significant figures are discarded by the requested precision.

```
-- signed numeral
SEQ
  uncooked ? sign.char
  signed := (sign.char = '-') OR plus OR blank
  IF
    signed AND blank AND (sign.char = '+')
      sign.char := '*s'  -- replace '+' by space
    otherwise
      SKIP
  text.size := raw.size - 1  -- sign dealt with already
  get.first.significant(uncooked, text.size, first)
  IF
    format.type = 'd'
      field[precision] := max(text.size, field[precision])
    format.type <> 'd'
      SKIP
```

The code for the unsigned hexadecimal case simply replaces the sign manipulation by an assignment of `FALSE` to *signed*, and again adjusts the precision.

The third phase of *cook.output* uses the accumulated information to produce output on *cooked*. The precision now represents how many ordinary characters (excluding extras such as sign and decimal point) are to be displayed. Comparing this with the number of significant characters available determines the requisite number of padding zeros to insert or characters to discard according as the precision is greater or less than that number, and the number of characters to copy from *uncooked* to *cooked*; comparing with the field width gives the number of spaces needed to pad out the result. The necessary calculations are performed by *set.padding* (Process 10.19).

Process 10.19

```
PROC set.padding(VAL []INT field, VAL BOOL signed,
                 VAL INT significant.chars,
                 INT pad.zeros, pad.spaces, wanted, unwanted )
  VAL INT total.width IS field[width] :
  VAL INT displayed IS field[precision] :
  SEQ
    VAL INT discrepancy IS displayed - significant.chars :
    IF
      discrepancy >= 0  -- not enough uncooked text
        pad.zeros, wanted, unwanted :=
                discrepancy, significant.chars, 0
      discrepancy <= 0  -- too much
        pad.zeros, wanted, unwanted :=
                       0, displayed, -discrepancy
    pad.spaces := total.width - displayed
    IF
      signed
        pad.spaces := max(0, pad.spaces - 1)
      NOT signed
        pad.spaces := max(0, pad.spaces)
:
```

Notice that space is reserved for a sign character when one is required. The final `IF` in Process 10.19 could have been collapsed into a single line: since `INT TRUE` is one, whereas `INT FALSE` yields zero, the conditional pair of assignments are entirely equivalent to

```
pad.spaces := max(0, pad.spaces - (INT signed))
```

but the amount of explanation required by this coding would have more than made up the four lines saved by using it!

It now remains only to transmit the various batches of characters in the correct order. For all except the approximate real formats, these are a selection from the same sequence: right-justifying spaces; sign; leading zeros; characters from *uncooked*; and left-justifying spaces. Of course, not all of these will be present in displaying a single item; the spaces can only appear at one end, that determined by the *left.justify* flag. A Boolean never has a sign or leading zeros, but even a decimal numeral by default requires only the significant characters, if it is non-negative.

Rather than producing the optional material by a sequence of conditionals of the form

```
IF
  need.this.batch
    ... send this batch along cooked
  NOT need.this.batch
    SKIP
```

it is far neater to observe that each of the optional batches consists of a known number of a fixed character, and so abstract Process 10.20.

Process 10.20

```
PROC conditionally.repeat(VAL BOOL condition,
                          VAL INT count, VAL BYTE char,
                          CHAN OF BYTE output          )
  IF
    condition
      SEQ i = 1 FOR count
        output ! char
    NOT condition
      SKIP
:
```

The significant characters of the raw output must be copied, forcing letters to be capitals if the format character was itself one. This can be encapsulated in Process 10.21.

Process 10.21

```
PROC copy.translating(VAL BOOL change, CHAN OF BYTE source,
                      VAL INT count, VAL BYTE first,
                      CHAN OF BYTE sink                      )
  -- send first, then count-1 bytes from source along sink
  BYTE char :
  IF
    change      -- letters must be made capital
      SEQ
        sink ! upper.case(first)
        SEQ i = 2 FOR count - 1
          SEQ
            source ? char
            sink ! upper.case(char)
    NOT change  -- send unchanged
      SEQ
        sink ! first
        SEQ i = 2 FOR count - 1
          SEQ
            source ? char
            sink ! char
:
```

Finally, if the output sequence is truncated by the specified precision there remain characters unread from *uncooked*. These must be thrown away by Process 10.22.

Process 10.22

```
PROC discard.input(CHAN OF BYTE input, VAL INT unwanted)
  SEQ i = 0 FOR unwanted
    BYTE discard :
    input ? discard
:
```

These procedures fit together as follows in the common case of the third phase of Process 10.16:

```
-- straightforward cases
SEQ
  set.padding(field, signed, text.size,
                zero.pad, space.pad, copy, truncate )
  conditionally.repeat(right, space.pad, '*s', cooked)
  conditionally.repeat(signed, 1, sign.char, cooked)
  conditionally.repeat(TRUE, zero.pad, '0', cooked)
  copy.translating(upper, uncooked, copy, first, cooked)
  discard.input(uncooked, truncate)
  conditionally.repeat(left, space.pad, '*s', cooked)
```

Again, the two remaining cases are quite similar, although more complex; the same component processes are used, but the exponent and decimal point come into the space calculations. The complete listings can be found in the appendix (page 250).

That more or less concludes the development of *write.formatted*. One or two loose ends remain to be tied up: the `DATA.ITEM` protocol includes a variant for strings, but this routine makes no provision for them. There is a reason for this; one of the design criteria of this suite has been that it should operate individually on the characters of the stream representing the objects to be displayed, with an absolute minimum of buffering. The only occam 2 protocol (as opposed to any higher-level protocol which the programmer may build out of sequences of communications) which allows the transfer of a message of undetermined length as one of the variants of a discriminated protocol is a counted-array protocol like that used in the *data.string* case above. But the target of a counted-array input must itself be an array, and so a buffer must be declared somewhere. Moreover, there is no means of specifying as part of the protocol the size of the largest segment that may be sent, so an arbitrarily large buffer is required.

It would not be particularly difficult to add a clause to *raw.output* to accept a string into a local buffer, of some size declared to be 'big enough' by arbitrary fiat. The size of the string could be passed over *info*, and then each byte in turn along *uncooked*, to be treated in the established way by *cook.output*, as the Boolean strings are. This would of course fail, irremediably, if a user process tried to send a string that was 'too big'.

An alternative, which would give full flexibility to the caller of such a 'library' routine, at the expense of increasing the complexity of the interface, would be to require a buffer to be passed as one of the arguments to *write.formatted*, and for this ultimately to be passed to *raw.output*. It would then be the user's responsibility to ensure that enough space was reserved for the largest string that might be sent. Indeed, one can always transform something of the form

```
PROC uses.buffer(...)
  [large.fixed]BYTE buf :

  ...

:

...

  uses.buffer(...)

  ...
```

into

```
PROC needs.buffer([]BYTE buf, ...)

  ...

:

...

  [just.right]BYTE user.buf :
  needs.buffer(user.buf, ...)

  ...
```

or *vice versa*, depending on the particular relative weights attached to these two concerns.

Output into buffers

There are other circumstances, apart from counted-array protocols, where buffering is required: one is where output from two sources is to be combined into packets to be sent down a channel with a sequential protocol. Here, because there must be a single output process for each packet, even though the number and size of the items is fixed they must be stored and then retransmitted.

This buffering of, for instance, the digits representing a number can be done by a process executed in parallel with an instance of *write.formatted*; but packing digits into a fixed-size buffer is a task which does not require the full flexibility of that routine. It can be done more simply by a process like *make.decimal.string.int* (Process 10.23). If there is no room for all the digits, or for any minus sign required by a negative number, the buffer is filled with *length* asterisk characters. As always, other error indications would have

Process 10.23

```
PROC make.decimal.string.int([]BYTE buffer, VAL INT number)
  CHAN OF BYTE internal :
  PAR
    write.decimal.int(internal, number)
    BYTE sign, digit :
    INT size :
    VAL INT length IS SIZE buffer :
    SEQ
      internal ? sign
      size := decimal.raw - 1  -- sign already read
      get.first.significant(internal, size, digit)
      IF
        (size < length) OR ((size = length) AND (sign = '+'))
          -- put blanks, sign, and digits in correct place
          VAL INT spaces IS length - size :
          SEQ
            SEQ i = 0 FOR spaces
              buffer[i] := '*s'
            CASE sign
              '-'
                buffer[spaces - 1] := sign  -- there is room
              '+'
                SKIP
            buffer[spaces] := digit
            SEQ i = spaces + 1 FOR size - 1 -- remaining digits
              internal ? buffer[i]
        (size > length) OR ((size = length) AND (sign = '-'))
          -- too large to fit, so fill with '*' characters
          SEQ
            SEQ i = 0 FOR length
              buffer[i] := '**'
            discard.input(internal, size - 1)  -- discard digits
:
```

been possible; but the 'field width' for this translation is given implicitly by the size of *buffer*, and to exceed it really is an error.

Having a buffer available in fact makes the translation of a number into a sequence of bytes simpler even than this last example, were we starting from scratch without a battery of components to re-use. You might want to try coding *make.decimal.string.int* directly, remembering that a buffer, unlike a channel, can be filled in reverse order.