Formatted input and output

```
--  This chapter defines routines to be used in other programs for
--  converting between internal and textual representations of data,
--  principally
--
--      write.string, write.decimal.int, write.hexadecimal.int, etc
--      read.signed, read.line
--      write.formatted
--      make.decimal.string.int
--

--
--  write.character, write.string, write.bool
--

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

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

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)
:
```

```
--
--  write.hexadecimal
--

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]
:

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

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

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))
:

PROC write.hexadecimal.int32(CHAN OF BYTE output, VAL INT32 bits)
  VAL INT64 mask IS (INT64 (MOSTNEG INT32)) << 1 :
  write.hexadecimal.int64(output, (INT64 bits) /\ (BITNOT mask))
:
```

```
--
--  write.decimal
--

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

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

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

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

PROC write.decimal.int16(CHAN OF BYTE output, VAL INT16 number)
  write.decimal.int64(output, (INT64 number))
:

PROC write.decimal.int32(CHAN OF BYTE output, VAL INT32 number)
  write.decimal.int64(output, (INT64 number))
:
```

```
--
--  read.signed
--

VAL BOOL otherwise IS TRUE :

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
  BYTE char, sign :
  SEQ

    input ? char
    WHILE char = '*s'                -- skip leading spaces
      input ? char

    IF
      (char = '+') OR (char = '-')  -- read a possible sign
        SEQ
          sign := char
          input ? char
      (char <> '+') AND (char <> '-')
        sign := '+'

    WHILE char = '*s'                -- skip any spaces after the sign
      input ? char

    n := 0
    ok := ('0' <= char) AND (char <= '9')  -- check for digits

    WHILE ('0' <= char) AND (char <= '9')  -- and read them
      SEQ
        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
        input ? char
:
```

```
--
--  read.line
--

PROC read.line(CHAN OF BYTE keyboard, echo, []BYTE s, INT n)
  VAL BYTE backspace IS '*#08' :  -- control-H
  VAL BYTE bell      IS '*#07' :  -- control-G
  VAL BYTE cancel    IS '*#15' :  -- control-U
  VAL BYTE delete    IS '*#7F' :
  VAL BYTE enter     IS '*c' :
  VAL INT max.contents IS (SIZE s) - 1 : -- leave room for enter
  SEQ
    n := 0    -- string intially empty
    BYTE char :                         -- most recent char typed
    WHILE (n = 0) OR (char <> enter)    -- not used before set
      SEQ
        keyboard ? char
        CASE char
          backspace
            IF
              n > 0  -- string not empty
                SEQ
                  echo ! backspace
                  echo ! '*s'
                  echo ! backspace
                  n := n - 1
              n = 0  -- nothing to delete
                echo ! bell
          cancel
            WHILE n >= 0            -- cancel
              SEQ                          --   backspaces over the
                n := n - 1                 --   whole of the line
                echo ! backspace
                echo ! '*s'
                echo ! backspace
          enter
            SEQ                            -- carriage return
              s[n] := enter      --    is added to the string
              n := n + 1         --    and terminates the loop
          ELSE
            IF
              (n < max.contents) AND
                  ('*s' <= char) AND (char < delete)
                SEQ  -- if there is room and char is printable
                  echo ! char        --  it is echoed
                  s[n] := char       --  and added to the string
                  n := n + 1
              otherwise     -- anything else is an error
                echo ! bell
:
```

```
--
--  routines used throughout write.formatted
--

--
--  min, max, lower.case, upper.case
--

INT FUNCTION min(VAL INT a, b)
  INT min :
  VALOF
    IF
      a <= b
        min := a
      b <= a
        min := b
    RESULT min
:

INT FUNCTION max(VAL INT a, b)
  INT max :
  VALOF
    IF
      a >= b
        max := a
      b >= a
        max := b
    RESULT max
:

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
:

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
:
```

```
--
--  the DATA.ITEM protocol
--

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
:

--
--  digits in numeric text, precision constants
--

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

VAL INT real32.precision    IS  9 :
VAL INT real64.precision    IS 17 :
VAL INT nibbles.in.a.byte   IS  2 :
VAL INT nibbles.in.an.int16 IS  4 :
VAL INT nibbles.in.an.int32 IS  8 :
VAL INT nibbles.in.an.int64 IS 16 :

--
--  indexing constants for flag and field arrays
--

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 :

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

```
--
--  discard.input, set.unless.specified, conditionally.repeat,
--  get.first.significant, split.real32, split.real64
--

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

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

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
:

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
:

INT, INT64 FUNCTION split.real32(VAL REAL32 r)
  ... code omitted
:

INT, INT64 FUNCTION split.real64(VAL REAL64 r)
  ... code omitted
:
```

```
--
--  raw.output
--

PROC raw.output(CHAN OF DATA.ITEM data, VAL BYTE format,
                CHAN OF INT info, CHAN OF BYTE uncooked )
  CASE format
    '%'  --  percent sign: no data item; just write.character
      write.character(uncooked, '%')

    'b'  --  Boolean: 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)

    'c'  --  character: accept data.byte for write.character
      BYTE character :
      SEQ
        data ? CASE data.byte; character
        write.character(uncooked, character)

    'd'  --  decimal (signed) integer
      data ? CASE
        BYTE number :
        data.byte; number
          write.decimal.byte(uncooked, number)
        INT number :
        data.int; number
          write.decimal.int(uncooked, number)
        INT16 number :
        data.int16; number
          write.decimal.int16(uncooked, number)
        INT32 number :
        data.int32; number
          write.decimal.int32(uncooked, number)
        INT64 number :
        data.int64; number
          write.decimal.int64(uncooked, number)
```

```
'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
  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
    PAR
      write.decimal.int64(uncooked, mantissa)
      info ! exponent
```

```
    'x'  --  hexadecimal (unsigned) integer
      data ? CASE
        BYTE number :
        data.byte; number
          SEQ
            info ! nibbles.in.a.byte
            write.hexadecimal.byte(uncooked, number)
        INT16 number :
        data.int16; number
          SEQ
            info ! nibbles.in.an.int16
            write.hexadecimal.int16(uncooked, number)
        INT32 number :
        data.int32; number
          SEQ
            info ! nibbles.in.an.int32
            write.hexadecimal.int32(uncooked, number)
        INT64 number :
        data.int64; number
          SEQ
            info ! nibbles.in.an.int64
            write.hexadecimal.int64(uncooked, number)
        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)
:
```

```
--
--  copy.translating
--

PROC copy.translating(VAL BOOL change, CHAN OF BYTE source,
                      VAL INT count, VAL BYTE first,
                      CHAN OF BYTE 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
:

--
--  set.padding
--

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)
:
```

```
--
--  cook.output
--

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
    --
    --  set up expected numbers of characters etc
    --
    CASE format.type
      '%', 'c'
        raw.size, field[precision] := 1, 1

      'b'
        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)

      'd'
        raw.size := decimal.raw

      'e', 'f'
        INT max.chars IS field[precision] :
        INT default.precision :
        SEQ
          info ? default.precision
          set.unless.specified(max.chars, default.precision)
          raw.size, max.chars :=
               decimal.raw, min(max.chars, default.precision)

      'x'
        INT min.chars IS field[precision] :
        INT default.precision :
        SEQ
          info ? default.precision
          set.unless.specified(min.chars, default.precision)
          raw.size := hexadecimal.raw
```

```
--
--  process signs and leading zeros
--
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] :
    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 = 'd'
          field[precision] := max(text.size, field[precision])
        format <> 'd'
          SKIP

  'x'
    SEQ
      text.size, signed := raw.size, FALSE
      get.first.significant(uncooked, text.size, first)
      field[precision] := max(text.size, field[precision])
```

```
--
--  output the massaged representation
--
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'
    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)

  'e'
    VAL INT exp IS 3 :  -- digits of exponent
    SEQ
      -- adjust for punctuation:   point   E  sign
      field[width] := field[width] - (1 + (1 + (1 + exp)))
      set.padding(field, signed, text.size,
                             zero.pad, space.pad, copy, truncate )
      conditionally.repeat(right, space.pad, '*s', cooked)
      SEQ  -- mantissa
        conditionally.repeat(signed, 1, sign.char, cooked)
        cooked ! '.'  -- decimal point
        copy.translating(upper, uncooked, copy, first, cooked)
        conditionally.repeat(TRUE, zero.pad, '0', cooked)
        discard.input(uncooked, truncate)
      SEQ  -- exponent
        -- E and exponent sign
        copy.translating(upper, uncooked, 2, 'e', cooked)
        -- and then exponent digits
        text.size := raw.size - 1  -- sign already read
        get.first.significant(uncooked, text.size, first)
        field[precision] := exp
        INT ignored :
        set.padding(field, TRUE, text.size,
                               zero.pad, ignored, copy, truncate )
        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)
```

```
        'f'
          INT exponent, number.of.digits, extra.zeros :
          SEQ
            field[width] := field[width] - 1  -- allow for point
            info ? exponent
            INT sig.figs IS field[precision] :
            IF
              exponent >  0  -- number of digits before point
                number.of.digits := max(exponent, sig.figs)
              exponent <= 0  -- MINUS number of zeros after point
                number.of.digits := sig.figs - exponent
            INT ignored :
            set.padding(field, signed, text.size,
                                    zero.pad, ignored, copy, truncate )
            text.size, field[precision] :=
                                     field[precision], number.of.digits
            INT ignored1, ignored2 :
            set.padding(field, signed, text.size,
                           extra.zeros, space.pad, ignored1, ignored2 )
            conditionally.repeat(right, space.pad, '*s', cooked)
            IF
              exponent >= copy              --  all digits before point
                SEQ
                  zero.pad := zero.pad + extra.zeros
                  copy.translating(upper, uncooked,
                                          copy, first, cooked )
                  conditionally.repeat(TRUE, zero.pad, '0', cooked)
                  cooked ! '.'

              (0 < exponent) AND (exponent < copy)
                                            --  split across point
                SEQ
                  copy.translating(upper, uncooked,
                                     exponent, first, cooked )
                  copy.translating(upper, uncooked,
                                  (copy - exponent) + 1, '.', cooked )

              exponent <= 0                 --  all after point
                SEQ
                  cooked ! '.'
                  conditionally.repeat(TRUE, extra.zeros, '0', cooked)
                  copy.translating(upper, uncooked,
                                                copy, first, cooked )
                  conditionally.repeat(TRUE, zero.pad, '0', cooked)

            discard.input(uncooked, truncate)
            conditionally.repeat(left, space.pad, '*s', cooked)
    :
```

```
--
--  formatted.output
--

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

--
--  process.item
--

PROC process.item(CHAN OF DATA.ITEM data,
                  VAL []BYTE control, INT i,
                  CHAN OF BYTE output       )
  [number.of.flags]BOOL flag :
  [field.numbers]INT field :
  INT which.field :
  BOOL this.format :
  BYTE format.char :
  VAL INT size IS SIZE control :
  SEQ
    flag, field := [FALSE, FALSE, FALSE], [-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, i := control[i], i + 1
        CASE format.char

          ELSE
            this.format := FALSE  -- parsed format successfully
```

```
        '-', '+', '*s'
          IF
            which.field = get.flag
              CASE format.char
                '-'
                  flag[left.justify] := TRUE
                '+'
                  flag[sign.plus] := TRUE
                '*s'
                  flag[sign.blank] := TRUE
            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'
          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

  formatted.output(format.char, flag, field, data, output)
:
```

```
--
--  write.formatted
--

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

```
--
--  make.decimal.string.int
--

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
:
```