XEROX

PALO ALTO RESEARCH CENTER
Computer Science Laboratory

To: Beginning MESA programmers
From: Niklaus Wirth
Date: February 22, 1977

SoME SAmple Programs

I had started "experimenting" with the programming language MESA about three months ago. Having had already experience in programming with high-level languages in general and with PASCAL in particular, I launched into the MESA world by rewriting some typical student sample programs, remembering their basic algorithms from earlier teaching endeavours. In early January, I reported on my "joys and perils", focussing on those points that a MESA novice is likely to encounter, for good or for worse. The positive response to this talk, and a request for a written document that might be useful for other newcomers to MESA have encouraged me to write this document. It is neither a manual nor a handbook nor a carefully prepared comprehensive set of sample programs. It is merely the collection of my programs which, I hope, are formulated clearly and suitably. I refrain from explaining the particular algorithms in detail, but shall rather point out details that might be noteworthy from the point of view of programming with MESA.

1. A prime number generator.

The algorithm is the one developed by Dijkstra, avoiding the use of multiplication and division (see also Wirth: *Systematic Programming*, Chap. 15). The program generates and tabulates the first N prime numbers, where N is a given constant. It illustrates the use of the data types INTEGER and BOOLEAN, the data structure ARRAY, and the most common conditional and iterative statement forms. It also shows a pure PROCEDURE with a single parameter. There is no input; output goes to the standard destination: the display.

For output, the standard I/O package is used. For this purpose, the program's directory contains the entry IODefs. The two procedures used are *WriteNumber* and *WriteChar*, the latter for terminating a line after 10 numbers with a CR character. *WriteNumber* requires the specification of a format. This is done by declaring a record of type *NumberFormat*; its last field denotes the number of characters to be allocated to the number. *WriteNumber*, *WriteChar*, *NumberFormat*, and CR are imported from IODefs.

Note: the constant N1 = sqrt(N) cannot be specified in terms of N, because MESA doesn't contain the notion of square root. The constant 10 in *print* specifies the number of primes to be put on each line. The product of this and the number field width must not exceed the line length of about 80. Actually, the MESA output display window is even narrower (which I found a nuisance). However, the output is stored as a file named *mesa.typescript*, and is best hardcopied by the operating system command *Gears mesa.typescript* .

If you are used to Algol, PL/1, or Pascal programming, you may have a tendency to write a semicolon following the procedure heading, whereas MESA expects an equal sign. Try to figure out what would happen with a semicolon. It will be much more disastrous than a mere syntax error! Why? The reason for MESA's use of an equal sign is due to its uniformity of syntax for all declarations.

<identifier list> : <category> <initialization>

The category is denoted by the keywords TYPE, PROCEDURE, PROGRAM, etc., or a type expression. The initialization part is either empty or consists of an expression (yielding the initial value) preceded by an equal sign (=) or an arrow (←). The equal sign signifies that the new object is to be *constant* throughout its lifetime, whereas the arrow denotes the assignment to an initial value to the *variable* that may be changed through further assignments. Conventional programming languages consider procedures always as constant, i.e. a given procedure identifier always denotes the same procedure code. In MESA, however, this constancy must be explicitly indicated by an equal sign. If initialization is missing (or if an arrow is used), the new object is a so-called *procedure variable.*

```
DIRECTORY
  IODefs: FROM "iodefs";
DEFINITIONS FROM IODefs;

Primes: PROGRAM =
BEGIN
  N: INTEGER = 1600;  N1: INTEGER = 40;  -- N1 = sqrt(N)
  x, square, i,k, cnt, lim: INTEGER;
  prim: BOOLEAN;
  v, p: ARRAY [1 .. N1] OF INTEGER;
  f: NumberFormat = [10, FALSE, FALSE, 8];

  print: PROCEDURE [x: INTEGER] =
    BEGIN WriteNumber[x, f]; cnt ← cnt + 1;
      IF cnt = 10 THEN
        BEGIN WriteChar[CR]; cnt ← 0
        END
    END ;

  p[1] ← 2; cnt ← 0; print[2];
  x ← 1; lim ← 1; square ← 4;
  FOR i INCREASING IN [2 .. N] DO
    prim ← FALSE;
    UNTIL prim DO
      x ← x + 2;
      IF square <= x THEN
      BEGIN v[lim] ← square;
        lim ← lim+1; square ← p[lim]*p[lim]
      END ;
      k ← 2; prim ← TRUE;
      WHILE prim AND (k < lim) DO
        IF v[k] < x THEN v[k] ← v[k] + p[k];
```

```
        prim ← x # v[k];  k ← k+1
      ENDLOOP
    ENDLOOP ;
    IF i <= N1 THEN p[i] ← x;
    print[x]
  ENDLOOP ;
  WriteChar[CR]
END .
```

Computation of the first 1600 prime numbers took 2 minutes on Alto2.

2. Compute Fractions.

This program computes the fractions 1/2, 1/3, ... , 1/N, where N is a given constant. In addition to the fractions, it lists the length of the period, if there is one. The output, with N = 20, looks as follows:

```
 2  .5
 3  .3...   1
 4  .25
 5  .2
 6  .16...   1
 7  .142857...   6
 8  .125
 9  .1...   1
10  .1
11  .09...   2
12  .083...   1
13  .076923...   6
14  .0714285...   6
15  .06...   1
16  .0625
17  .0588235294117647...  16
18  .05...   1
19  .052631578947368421...  18
20  .05
```

The variables have the following meaning:

i	counter
d	divisor
q	quotient
r	remainder

The algorithm is that of so-called long division. The remainders are "marked" in the array R, i.e. if the remainder r occurs in division step i, then R[r] obtains the value i. The first reoccurrence of r signals termination of the fraction's period.

Output is again directed to the display, i.e. the file *mesa.typescript*. Since the fraction is computed digit after digit, the output is generated character after character using *WriteChar* imported from IODefs. The procedure has a character as its parameter. This character, however, is represented as an integer. The argument is the character's ASCII code (e.g. 40B for a blank, 60B for 0, 101B for A, etc.). This explains the additive term 60B. A

forthcoming version of the MESA compiler will treat the type CHARACTER as a new basic type instead of as a subrange of INTEGER. This will allow for full type checking by the compiler.

```
DIRECTORY IODefs: FROM "iodefs";
DEFINITIONS FROM IODefs;

Fractions: PROGRAM =
BEGIN
  N: INTEGER = 64;
  i,d,q,r: INTEGER;
  R: ARRAY [0 .. N) OF INTEGER;
  f: NumberFormat = [10, FALSE, FALSE, 4];

  R[0] ← 1;
  FOR d IN (1 .. N] DO
    WriteNumber[d,f]; WriteString["  ."];
    FOR r IN [1 .. d) DO
      R[r] ← 0
    ENDLOOP ;
    i ← 1; r ← 1;
    WHILE R[r] = 0 DO
      R[r] ← i ← i+1;
      r ← 10*r; q ← r/d; r ← r - q*d; WriteChar[q+60B]
    ENDLOOP ;
    IF r > 0 THEN
      BEGIN WriteString["..."]; WriteNumber[i-R[r]+1, f]
      END ;
    WriteChar[CR]
  ENDLOOP
END .
```

The Alto takes about 20 seconds to compute and display the fractions 1/2, ... , 1/64.

3. Tabulate powers of 2.

In the next program we show how to output onto a file that is newly created and given a fixed name. We do so with the basic language facilities, i.e. without reliance on the IODefs package. Nevertheless, we need some additional support facilities, namely *StreamDefs* and *SegmentDefs*. (These two are used by *IODefs* too.) From *StreamDefs* we import the data type *StreamHandle* and the function procedure *CreateByteStream*. The former is a pointer referring to a *StreamObject*, a descriptor of a stream, i.e. of a sequence of elements, in this case of characters (or bytes?). The stream is a notion (or abstraction) that builds on the Alto OS notion of a file or segment. Therefore we need a facility to create a new file that appears in MESA as a stream. From *SegmentDefs* we import the function procedure *NewFile*; its parameters are the new file's name (a string) and so-called file attributes. These attributes actually form a set, yet are represented as integers (powers of 2!) that can be added up to obtain combinations.

Actual output statements are formed with the basic MESA stream operator *put*. Note that *put* requires a prefix which specifies the stream on which *put* operates. This is because a stream is defined as an object described by a descriptor, which in turn is declared as a record. Some of its components represent variables (describing the length, address, etc. of

the actual stream), others denote procedures. Consequently, they have to be prefixed with the record identifier when called like any other field identifier. The procedure *put* also requires two arguments, the stream again and the value to be appended. The values to be output are integers, although we think of them as characters; note that we create a *ByteStream*. The term *Byte* essentially means that all appended values must be in the range 0 ... 255.

It is appropriate to mention at this point that MESA proper does not have any facilities for input and output. They are all built up as separate modules (packages to be imported). The programmer has the freedom to choose any one available according to his needs, or even to construct his own.

The algorithm used to illustrate stream output generates a table of positive and negative powers of 2. Each line is composed as follows:

$$2^k \quad k \quad 2^{-k}$$

with k ranging from 1 to N, where N is a given constant (<100). Obviously, we do not wish to compute each power from scratch, but rather obtain it by doubling and halving the previous results. Therefore we declare two arrays a and b, a representing the value 2^k and b representing 2^{-k}. Each element stands for one decimal digit, a's digits ranging (from right to left) from 1 to m, and b's digits ranging from 1 to k (from left to right). The main program consists of the statement

```
WHILE k < N DO
  k ← k+1; compute a;
  output leading blanks; ouput digits of a;
  output 2 digits of k; output decimal point;
  compute and output digits of b; output CR
ENDLOOP
```

The following is the output file obtained with N = 48.

```
     2 01 .5
     4 02 .25
     8 03 .125
    16 04 .0625
    32 05 .03125
    64 06 .015625
   128 07 .0078125
   256 08 .00390625
   512 09 .001953125
  1024 10 .0009765625
  2048 11 .00048828125
  4096 12 .000244140625
  8192 13 .0001220703125
 16384 14 .00006103515625
 32768 15 .000030517578125
 65536 16 .0000152587890625
131072 17 .00000762939453125
262144 18 .000003814697265625
524288 19 .0000019073486328125
```

```
      1048576 20 .00000095367431640625
      2097152 21 .00000047683715820312 5
      4194304 22 .00000023841857910156 25
      8388608 23 .00000011920928955078125
     16777216 24 .00000005960464477539 0625
     33554432 25 .00000002980232238769 53125
     67108864 26 .00000001490116119384 765625
    134217728 27 .00000000745058059692 3828125
    268435456 28 .00000000372529029846 19140625
    536870912 29 .00000000186264514923 095703125
   1073741824 30 .00000000093132257461 5478515625
   2147483648 31 .00000000046566128730 77392578125
   4294967296 32 .00000000023283064365 386962890625
   8589934592 33 .00000000011641532182 6934814453125
  17179869184 34 .00000000005820766091 346740722656 25
  34359738368 35 .00000000002910383045 673703613281 25
  68719476736 36 .00000000001455191522 836685180664 0625
 137438953472 37 .00000000000727595761 418342590332 03125
 274877906944 38 .00000000000363797880 709171295166 015625
 549755813888 39 .00000000000181898940 354585647583 0078125
1099511627776 40 .00000000000090949470 177292823791 50390625
2199023255552 41 .00000000000045474735 088646411895 751953125
4398046511104 42 .00000000000022737367 544323205947 8759765625
8796093022208 43 .00000000000011368683 772161602973 93798828125
17592186044416 44 .00000000000005684341 886080801486 968994140625
35184372088832 45 .00000000000002842170 943040400743 4844970703125
70368744177664 46 .00000000000001421085 471520200371 74224853515625
140737488355328 47 .00000000000000710542 735760100185 871124267578125
281474976710656 48 .00000000000000355271 367880050092 93556213378906 25
```

```
DIRECTORY StreamDefs: FROM "streamdefs",
          SegmentDefs: FROM "segmentdefs";
DEFINITIONS FROM StreamDefs, SegmentDefs;

Powers: PROGRAM =
BEGIN
  N: INTEGER = 48;  N1: INTEGER = 16;  -- N1 = N*log(2)
  SP: CHARACTER = 40B;  CR: CHARACTER = 15B;
  Digit: TYPE = [0..9];
  i,k,m,c: INTEGER;  t: Digit;
  a: ARRAY [1..N1] OF Digit;
  b: ARRAY [1..N]  OF Digit;
  s: StreamHandle;

  PutDigit: PROCEDURE [d: Digit] =
    BEGIN s.put[s, d + 60B]
    END ;

  PutChar: PROCEDURE [ch: CHARACTER] =
```

```
    BEGIN s.put[s, ch]
    END ;

  k ← 0; m ← 1; a[1] ← 1; b[1] ← 10;
  s ← CreateByteStream[
       NewFile["powers.output", Write+Append, DefaultVersion],
              Write+Append];
  WHILE k < N DO
    k ← k+1; i ← 0; c ← 0;
    WHILE i < m DO
      i ← i+1; t ← 2*a[i] + c;
      IF t < 10 THEN c ← 0 ELSE
        BEGIN t ← t - 10; c ← 1
        END ;
      a[i] ← t
    ENDLOOP ;
    IF c > 0 THEN
      BEGIN m ← m+1; a[m] ← c
      END ;
    i ← N1;
    WHILE i > m DO
      PutChar[SP]; i ← i-1    -- blanks
    ENDLOOP ;
    WHILE i > 0 DO
      PutDigit[a[i]]; i ← i-1
    ENDLOOP ;
    PutChar[SP];
    t ← k/10; PutDigit[t];
    t ← k - 10*t; PutDigit[t];
    PutChar[SP]; PutChar['.];
    c ← 0; i ← 1;
    WHILE i < k DO
      c ← (10*c + b[i]); t ← c/2;
      c ← c - 2*t; b[i] ← t;
      PutDigit[t]; i ← i+1
    ENDLOOP ;
    b[k] ← 5; PutDigit[5]; PutChar[CR]
  ENDLOOP ;
  s.destroy[s]
END .
```

Notes: 1. Don't forget to close the output file by destroying the stream (i.e. clearing the internal buffers) at the end of the program. If you do forget, then the output file may be missing some tail, i.e. it will be truncated at some seemingly arbitrary point. 2. The file attributes have to be mentioned as arguments twice, namely to both *NewFile* and *CreateByteStream.* (I haven't been able to figure out why I need both *Append* and *Write,* but it is so). 3. This program does not generate any output on the display.

4. General input and output of streams.

The next program again elaborates on the subject of input and output: instead of merely using the standard display output, we wish to input from and output to arbitrary files that are named by the program's user. The files are newly created and given a name that is input from the keyboard as a string (of characters). The program is an example of the following very general pattern of sequential file processing.

```
UNTIL end of file DO
  input element; process element; output result
ENDLOOP
```

In this example we choose the "process" part to be very small, yet to make some sense. The "processing" consists of counting the characters copied. Separate counters are available according to the category of the copied character. At the end, the counts are output to the display using the standard procedures imported from IODefs.

```
DIRECTORY StreamDefs: FROM "streamdefs",
          SegmentDefs: FROM "segmentdefs",
          IODefs: FROM "iodefs";
DEFINITIONS FROM StreamDefs, SegmentDefs, IODefs;

CopyStream: PROGRAM =
BEGIN
  ch: CHARACTER;
  c0, c1, c2, c3, c4, c5: INTEGER;
  f: NumberFormat = [10, FALSE, FALSE, 8];
  s1: STRING ← [32];  s2: STRING ← [32];
  in, out: StreamHandle;

  c0 ← c1 ← c2 ← c3 ← c4 ← c5 ← 0;
  WriteString["input file: "]; ReadID[s1]; WriteChar[CR];
  WriteString["output file: "]; ReadID[s2]; WriteChar[CR];
  in ← CreateByteStream[NewFile[s1, Read, DefaultVersion], Read];
    out ← CreateByteStream[NewFile[s2, Write+Append, DefaultVersion],
Write+Append];
  UNTIL in.endof[in] DO
    ch ← in.get[in];
    IF ch < ' THEN c0 ← c0 + 1 ELSE
    IF ch = ' THEN c1 ← c1 + 1 ELSE
    IF ch < '0 THEN c2 ← c2 + 1 ELSE
    IF ch < ': THEN c3 ← c3 + 1 ELSE
    IF ch < 'A THEN c2 ← c2 + 1 ELSE
    IF ch < '[ THEN c4 ← c4 + 1 ELSE
    IF ch < 'a THEN c2 ← c2 + 1 ELSE
    IF ch < '{ THEN c5 ← c5 + 1 ELSE c2 ← c2 + 1;
    out.put[out,ch]
  ENDLOOP ;
  WriteNumber[c0+c1+c2+c3+c4+c5,f]; WriteLine[" characters copied"];
  WriteNumber[c0,f]; WriteLine[" control characters"];
  WriteNumber[c1,f]; WriteLine[" blanks"];
```

```
  WriteNumber[c2,f]; WriteLine[" special characters"];
  WriteNumber[c3,f]; WriteLine[" digits"];
  WriteNumber[c4,f]; WriteLine[" capital letters"];
  WriteNumber[c5,f]; WriteLine[" lower case letters"];
  in.destroy[in]; out.destroy[out]
END .
```

Note: You may wonder why the two string variables were not declared as

s1, s2: STRING ← [32]

As a matter of fact, they were declared so in an earlier version of this program. However, this form has a quite unexpected and disastrous effect (try it!).

The above declaration has the same effect as the declaration "s1, s2: STRING" followed by the assignment statement "s1 ← s2 ← [32]". This implies that you assign to both variables the same "thing". Now the important point is that in MESA a string variable is *not* a string variable, but rather a pointer variable pointing to a string variable. This fact has been conveniently hidden by MESA's dereferencing rules. The effect of the above assignment then is to assign the same reference to the same string variable to both s1 and s2. The statement ReadID[s2] therefore overwrites the string read beforehand by ReadID[s1], and as a result a later attempt to write onto the same file from which we are reading leads to a signal about an illegal file operation.

The above details are succinctly described in the MESA Manual on pages 84 and 85 (line 10). The wisdom of declaring a string *pointer* as a string, however, is debatable.

5. Constructing a bitmap.

Considering the strong orientation of the Alto towards use of the display, the construction of bitmaps is a most likely task for a MESA programmer. A bitmap can be figured as a two-dimensional array of bits, each bit representing a point on the screen with values black and white. However, MESA allocates at least one word for each array element, whereas the Alto uses each word to represent 16 points. Since MESA does not feature packed arrays, this packing has to be programmed explicitly. Ways to create, represent, and operate bitmaps are shown by this sample program, where a bit map is first cleared and then supplied with a raster of horizontal and vertical lines.

In Version 1 the bitmap is represented by a *two-dimensional* array, i.e. an array of arrays of "words". Let BM be a map and i, j map-coordinates. Instead of "BM[i,j] ← black", we write

```
BM[i,j/16] ← BITOR[BM[i,j/16], BITSHIFT[1, 15-(j MOD 16)]]
```

which can be rewritten somewhat more efficiently as

```
[k,b] ← DIVMOD[j,16];
BM[i,k] ← BITOR[BM[i,k], BITSHIFT[1, 15-b]]
```

using the function DIVMOD from InlineDefs. Instead of "BM[i,j] ← white", we write

```
BM[i,k] ← BITAND[BM[i,k], BITNOT[BITSHIFT[1, 15-b]]]
```

(It sure looks awkward). The logical bit operators apply to any type (that occupies a single word), hence throwing the type checker overboard. Using these operators, we are on relatively unsafe grounds, and it is wise to remember that fact. I have introduced my own type *Bits* for all variables to which bit operators are applied; although this doesn't help the type checker of the compiler, it my at least help the human type checker.

Address computation in multi-dimensional arrays is usually complicated and consumes time, even if a compiler is clever and recognizes special situations. In Version 2 we therefore represent a bitmap as a *one-dimensional* array, and program the mapping of the abstract two-dimensional map onto a one-dimensional row of "words" explicitly. For example, "BM[i,j] ← black" now becomes

```
[k,b] ← DIVMOD[j,16]; k ← i*WWidth + k;
BM[k] ← BITOR[BM[k], BITSHIFT[1, 15-b]]
```

where *WWidth* is the constant width of the map "in words", i.e. divided by 16. Efficiency is gained compared to Version 1 in particular when in repeated accesses to BM[i,j] the vertical coordinate i stays constant. (This may be viewed by watching how fast these 2 versions clear the screen).

Bitmaps are usually large arrays. It is impossible in MESA to declare large objects as static variables. Rather, they should be allocated dynamically. For some reason, that works better. The penalty is that all references to such objects then must be indirect via a pointer. However, this pointer is conveniently hidden in MESA by virtue of its dereferencing rules. This means, that we can just as well write BM[i,j] instead of BM↑[i,j], although BM is actually a pointer to an array instead of the array itself.

If you do not like automatic dereferencing, you may resort to a third method of representing a bitmap: using a so-called *array descriptor*. It is essentially the same as a pointer (but also contains associated index bounds which are - however - not utilised). The difference is therefore rather conceptual: If BM is declared as an array descriptor (to which an array is allocated by some statement), then BM will *always* automatically refer to the array, and the descriptor is *always* hidden. Hence, we declare

Version 1: `BM: POINTER TO ARRAY [0..Height) OF ARRAY [0..WWidth) OF Bits`

Version 2: `BM: POINTER TO ARRAY [0..BMSize) OF Bits`

Version 3: `BM: DESCRIPTOR FOR ARRAY OF Bits`

and we allocate storage through the statement

Version 1, Version 2: `BM ← AllocateSegment[BMSize]`

Version 3: `BM ← DESCRIPTOR[AllocateSegment[BMSize], BMSize]`

where *BMSize = Height*WWidth*, and where *AllocateSegment* is imported from *SystemDefs*. However, there is a glitch! The Alto hardware requires that any bitmap be allocated at an even address. The same holds for display control blocks (DCB). A possible solution is shown by the procedure *New*: always allocate one word too much.

It is noteworthy that this program would not call for any application of pointers, were it not for considerations of implementation. After all, its variables are purely static. In the case of the bitmap, the pointer became necessary due to the allocation mechanisms available, and in both cases (BM and DCB), dynamic allocation is necessary because there is

no other way to force a variable to be allocated at an even address.

Actually, there exist two mechanisms for dynamic storage allocation (see <MESA-doc>storage). The routines *AllocateSegment* and *AllocatePages* are used for allocating relatively large objects that are (more or less) permanent. *AllocateHeapNode* is used to allocate relatively small objects that "come and go", and are therefore subject to a more sophisticated storage retrieval strategy. In this program, we use the latter for both occasions.

The program has three steps: clearing the screen (bitmap), drawing horizontal lines, drawing vertical lines. Hence it generates a raster of squares with width H. (Note that 7*73 = 511 = 2**9 - 1).

```
DIRECTORY                       -- Version 1.  N.Wirth  15. Feb. 77
  InlineDefs: FROM "inlinedefs",
  SystemDefs: FROM "systemdefs",
  StreamDefs: FROM "streamdefs";
DEFINITIONS FROM InlineDefs, SystemDefs, StreamDefs;

Raster: PROGRAM =
BEGIN  -- Constants --
  H: INTEGER = 73;              -- size of raster square
  Width: INTEGER = 512;          -- in no. of points
  Height: INTEGER = 512;         -- in no. of scan lines
  WWidth: INTEGER = Width/16;    -- in words
  BMSize: INTEGER = WWidth * Height;
  ScreenWidth: INTEGER = 606;
  ScreenHeight: INTEGER = 808;

  -- Pointer Constants --
  DCBnil: DCBptr = @MEMORY[0];
  DCBorg: POINTER TO DCBptr = @MEMORY[420B];

  -- Types --
  Bits: TYPE = UNSPECIFIED;
  BitMap: TYPE = ARRAY [0..Height) OF ARRAY [0..WWidth) OF Bits;
  BMptr: TYPE = POINTER TO BitMap;
  DCBptr: TYPE = POINTER TO DCB;
  DCB: TYPE = MACHINE DEPENDENT RECORD
       [next: DCBptr,
        resolution, background: [0..1],
        indentation: [0..77B],  -- in units of 16 bits
        width: [0..377B],       -- likewise, even
        bitmap: BMptr,          -- even
        height: INTEGER];       -- in double scan lines

  -- Variables --
  dcb1, dcb2: DCBptr;
  BM: BMptr;
  i,j: INTEGER;
  ch: CHARACTER;
```

```
key: StreamHandle;

New: PROCEDURE [n: INTEGER] RETURNS [POINTER] =
  -- allocate n words with even address
  BEGIN p: POINTER;
    p ← AllocateHeapNode[n+1]; p ← p + BITAND[LOOPHOLE[p], 1];
    RETURN [p]
  END ;

vertical: PROCEDURE [x: INTEGER] =
  -- draw vertical line at coordinate x
  BEGIN i,j,b: INTEGER;  t: Bits;
    [j,b] ← DIVMOD[x, 16]; t ← BITSHIFT[1, 15-b]; i ← 0;
    WHILE i < Height DO
      BM[i][j] ← BITOR[BM[i][j], t]; i ← i+1
    ENDLOOP
  END ;

horizontal: PROCEDURE [y: INTEGER] =
  -- draw horizontal line at coordinate y
  BEGIN i,j: INTEGER;
    i ← y; j ← 0;
    WHILE j < WWidth DO
      BM[i][j] ← 177777B; j ← j+1
    ENDLOOP
  END ;

-- Start Main Program: initialize bitmap and descriptors --
key ← GetDefaultKey[];
dcb1 ← New[4];
BM ← New[BMSize];
dcb1↑ ← DCB[DCBnil, 0, 0, (ScreenWidth-Width)/32, WWidth, BM, Height/2];
dcb2 ← New[4];
dcb2↑ ← DCB[dcb1, 0, 0, 0, 0, BM, (ScreenHeight-Height)/4];
dcb1 ← DCBorg↑; DCBorg↑ ← dcb2;

ch ← key.get[key];  -- wait
FOR i IN [0..Height) DO
  FOR j IN [0..WWidth) DO
    BM[i][j] ← 0
  ENDLOOP
ENDLOOP ;

ch ← key.get[key];  -- wait
i ← 0;
WHILE i < Height DO
  horizontal[i]; i ← i + H
ENDLOOP ;

ch ← key.get[key];  -- wait
```

```
  j ← 0;
  WHILE j < Width DO
    vertical[j]; j ← j + H
  ENDLOOP ;

  ch ← key.get[key];  -- wait
  DCBorg↑ ← dcb1      -- restore display
END .


DIRECTORY                       -- Version 2.  N.Wirth  15. Feb. 77
  InlineDefs: FROM "inlinedefs",
  SystemDefs: FROM "systemdefs",
  StreamDefs: FROM "streamdefs";
DEFINITIONS FROM InlineDefs, SystemDefs, StreamDefs;

Raster: PROGRAM =
BEGIN  -- Constants --
  H: INTEGER = 73;              -- size of raster square
  Width: INTEGER = 512;         -- in no. of points
  Height: INTEGER = 512;        -- in no. of scan lines
  WWidth: INTEGER = Width/16;   -- in words
  BMSize: INTEGER = WWidth * Height;
  ScreenWidth: INTEGER = 606;
  ScreenHeight: INTEGER = 808;

  -- Pointer Constants --
  DCBnil: DCBptr = @MEMORY[0];
  DCBorg: POINTER TO DCBptr = @MEMORY[420B];

  -- Types --
  Bits: TYPE = UNSPECIFIED;
  BitMap: TYPE = ARRAY [0..BMSize) OF Bits;
  BMptr: TYPE = POINTER TO BitMap;
  DCBptr: TYPE = POINTER TO DCB;
  DCB: TYPE = MACHINE DEPENDENT RECORD
       [next: DCBptr,
        resolution, background: [0..1],
        indentation: [0..77B],  -- in units of 16 bits
        width: [0..377B],       -- likewise, even
        bitmap: BMptr,          -- even
        height: INTEGER];       -- in double scan lines

  -- Variables --
  dcb1, dcb2: DCBptr;
  BM: BMptr;
  i,j: INTEGER;
  ch: CHARACTER;
  key: StreamHandle;
```

```
New: PROCEDURE [n: INTEGER] RETURNS [POINTER] =
  -- allocate n words with even address
  BEGIN p: POINTER;
    p ← AllocateHeapNode[n+1]; p ← p + BITAND[LOOPHOLE[p], 1];
    RETURN [p]
  END ;

vertical: PROCEDURE [x: INTEGER] =
  -- draw vertical line at coordinate x
  BEGIN k,b: INTEGER;  t: Bits;
    [k,b] ← DIVMOD[x, 16]; t ← BITSHIFT[1, 15-b];
    WHILE k < BMSize DO
      BM[k] ← BITOR[BM[k], t]; k ← k + WWidth
    ENDLOOP
  END ;

horizontal: PROCEDURE [y: INTEGER] =
  -- draw horizontal line at coordinate y
  BEGIN k,m: INTEGER;
    k ← y * WWidth; m ← k + WWidth;
    WHILE k < m DO
      BM[k] ← 177777B; k ← k+1
    ENDLOOP
  END ;

-- Start Main Program: initialize bitmap and descriptors --
key ← GetDefaultKey[];
dcb1 ← New[4];
BM ← New[BMSize];
dcb1↑ ← DCB[DCBnil, 0, 0, (ScreenWidth-Width)/32, WWidth, BM, Height/2];
dcb2 ← New[4];
dcb2↑ ← DCB[dcb1, 0, 0, 0, 0, BM, (ScreenHeight-Height)/4];
dcb1 ← DCBorg↑; DCBorg↑ ← dcb2;

ch ← key.get[key];  -- wait
FOR i IN [0..BMSize) DO
  BM[i] ← 0
ENDLOOP ;

ch ← key.get[key];  -- wait
i ← 0;
WHILE i < Height DO
  horizontal[i]; i ← i + H
ENDLOOP ;

ch ← key.get[key];  -- wait
j ← 0;
WHILE j < Width DO
  vertical[j]; j ← j + H
ENDLOOP ;
```

```
  ch ← key.get[key];  -- wait
  DCBorg↑ ← dcb1      -- restore display
END .
```

Notes: 1. If you must access a fixed location (such as the display chain origin) of type T and with address a, declare a pointer constant

p: POINTER TO T = @MEMORY[a]

then access that location by writing "p↑".
2. If in a program the display chain origin at location 420B is altered, the old value should be restored at the end of the program.
3. *AllocHeapNode* is imported from *SystemDefs, GetDefaultKey* and *StreamHandle* from *StreamDefs*, DIVMOD, BITAND, BITOR, BITXOR, BITNOT, and BITSHIFT from *InlineDefs*.

6. Using cursor, mouse, and BitBlt.

We can easily extend the previous program to illustrate the use of the display cursor, the mouse, and the special BitBlt instruction to operate on bitmaps. The following program

1. inputs an integer n from the keyboard (0 < n < 32).
2. draws a raster with n*n squares.
3. paints the square indicated by the current cursor position black, gray, or white depending on which mouse button is clicked.

Hint: You can use this program to play Tictactoe with n = 3, or Goban with n = 12.

Although the main principles of this program are the same as that of the previous example, it looks quite different. The reason is the use of the BitBlt instruction, whose effects are specified in a descriptor called a BitBlt-Table (BBT). It can be used to copy, paint, invert, and erase any rectangular area in a bitmap. We also use it to draw the horizontal (height = 1) and vertical (width = 1) lines.

```
DIRECTORY                       -- N.Wirth  17. Feb 77
  InlineDefs: FROM "inlinedefs",
  SystemDefs: FROM "systemdefs",
  IODefs:     FROM "iodefs";
DEFINITIONS FROM InlineDefs, SystemDefs, IODefs;

Goban: PROGRAM =
BEGIN  -- Constants --
  nMax: INTEGER = 31;           -- max no. of squares per row
  Width: INTEGER = 512;         -- in no. of raster points
  Height: INTEGER = Width;      -- in no. of scan lines
  WWidth: INTEGER = Width/16;   -- in words
  ScreenWidth: INTEGER = 606;
  ScreenHeight: INTEGER = 808;
  LeftMargin: INTEGER = (ScreenWidth - Width)/32; -- in words
  UpMargin: INTEGER = (ScreenHeight - Height)/2;  -- in points
```

```
BMSize: INTEGER = WWidth * Height;

-- Pointer Constants --
DCBnil: DCBptr = @MEMORY[0];
DCBorg: POINTER TO DCBptr = @MEMORY[420B];
MouseButton: POINTER TO Bits = @MEMORY[177030B];
CursorX: POINTER TO INTEGER = @MEMORY[426B];
CursorY: POINTER TO INTEGER = @MEMORY[427B];
CursorBM: POINTER TO CursorBitMap = @MEMORY[431B];

-- Types --
Bits: TYPE = UNSPECIFIED;
BitMap: TYPE = ARRAY [0..BMSize) OF Bits;
BMptr: TYPE = POINTER TO BitMap;
DCBptr: TYPE = POINTER TO DCB;
CursorBitMap: TYPE = ARRAY [0 .. 16) OF Bits;

DCB: TYPE = MACHINE DEPENDENT RECORD  -- DisplayControlBlock
    [next: DCBptr,
     resolution, background: [0..1],
     indentation: [0..77B],  -- in units of 16 bits
     width: [0..377B],       -- likewise, even
     bitmap: BMptr,          -- even
     height: INTEGER];       -- in double scan lines

BBT: TYPE = MACHINE DEPENDENT RECORD  -- BitBltTable
    [function, unused: INTEGER,
     -- BitBlt function codes:
     --   0: d ← s'  (replace)      0: s' = s
     --   1: d ← s' ior d (paint)   4: s' = ~s
     --   2: d ← s' xor d (invert)  8: s' = s & g
     --   3: d ← ~s' & d (erase)   12: s' = g
     dadr: BMptr,            -- destination address
     dwidth: INTEGER,        -- dest. bitmap width
     dx,dy,dw,dh: INTEGER,   -- dest. block coordinates
     sadr: BMptr,            -- source address
     swidth: INTEGER,        -- source bitmap width
     sx,sy: INTEGER,         -- source block coordinates
     g0,g1,g2,g3: Bits];     -- gray block g

-- Variables --
dcb1, dcb2: DCBptr;
BM: BMptr;                   -- main BitMap
bbt: POINTER TO BBT;
i,j,n,w,h: INTEGER;
mouse: Bits;                 -- mouse buttons

BitBlt: EXTERNAL PROCEDURE [p: POINTER TO BBT];

New: PROCEDURE [n: INTEGER] RETURNS [POINTER] =
```

```
  -- allocate n words with even address
  BEGIN p: POINTER;
    p ← AllocateHeapNode[n+1]; p ← p + BITAND[LOOPHOLE[p], 1];
    RETURN [p]
  END ;

shade: PROCEDURE [x,y,w,h: INTEGER] =
  -- shade square of width w, height h at coordinates x,y
  BEGIN bbt↑.function ← 12;
    bbt↑.dx ← x; bbt↑.dy ← y; bbt↑.dw ← w; bbt↑.dh ← h;
    BitBlt[bbt]
  END ;

SetCursor: PROCEDURE [u: Bits] =
  BEGIN i: INTEGER;
    CursorBM↑[0] ← CursorBM↑[15] ← 177777B;
    FOR i IN [1..14] DO CursorBM↑[i] ← BITOR[u, 100001B]
    ENDLOOP
  END ;

SetGray: PROCEDURE [a,b: Bits] =
  BEGIN bbt↑.g0 ← bbt↑.g1 ← a; bbt↑.g2 ← bbt↑.g3 ← b;
    SetCursor[a]
  END ;

-- Start Main Program: initialize bitmap and descriptors --
dcb1 ← New[4];
BM ← New[BMSize];
dcb1↑ ← DCB[DCBnil, 0, 0, LeftMargin, WWidth, BM, Height/2];
dcb2 ← New[4];
dcb2↑ ← DCB[dcb1, 0, 0, 0, 0, BM, UpMargin/2];
bbt ← New[16];
bbt↑.dadr ← bbt↑.sadr ← BM;
bbt↑.dwidth ← bbt↑.swidth ← WWidth;


DO WriteString["No. of rows = "];
  n ← ReadDecimal[]; WriteChar[CR];
  IF NOT n IN [1 .. nMax] THEN EXIT;
  SetGray[0,0]; shade[0,0,Width,Height];   -- clear display
  dcb1 ← DCBorg↑; DCBorg↑ ← dcb2;          -- start display
  h ← (Width-1)/n; w ← n*h +1;
  SetGray[177777B, 177777B];
  i ← 0;
  WHILE i < w DO
    shade[i,0,1,w]; i ← i + h     -- vertical lines
  ENDLOOP ;
  j ← 0;
  WHILE j < Width DO
    shade[0,j,w,1]; j ← j + h     -- horizontal lines
```

```
    ENDLOOP ;

    DO mouse ← BITXOR[BITAND[MouseButton↑, 7], 7];
      IF mouse # 0 THEN
      BEGIN SetCursor[0];
        WHILE BITXOR[BITAND[MouseButton↑, 7], 7] # 0 DO
          -- wait until button released
        ENDLOOP ;
        i ← CursorX↑ - LeftMargin*16 +8;
        j ← CursorY↑ - UpMargin +8;
        IF i IN [0 .. w) AND j IN [0 .. w) THEN
        BEGIN i ← (i/h)*h; j ← (j/h)*h;
          IF mouse = 4 THEN SetGray[177777B, 177777B] ELSE
          IF mouse = 2 THEN SetGray[0,0] ELSE
            SetGray[114631B, 063146B];
          shade[i+1, j+1, h-1, h-1]
        END ELSE EXIT;
      END
    ENDLOOP ;
    DCBorg↑ ← dcb1      -- restore display
  ENDLOOP
END .
```