stichting
mathematisch
centrum

AFDELING INFORMATICA

P. TEN HAGEN & H. NOOT

A DIGISET SIMULATOR

2e boerhaavestraat 49 amsterdam
Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

The Mathematical Centre, founded the 11-th of February 1946, is a non-profit institution aiming at the promotion of pure mathematics and its applications. It is sponsored by the Netherlands Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O), by the Municipality of Amsterdam, by the University of Amsterdam, by the Free University at Amsterdam, and by industries.
A DIGISET SIMULATOR

by

P. ten Hagen & H. Noot

ABSTRACT

In this report, it is shown, that the Digiset-40 T1 photo-typesetter can be used with good results as a device for producing two dimensional hard-copy output generated by a computer program.

The hardware characteristics and instruction set of the Digiset are described in sufficient detail, to enable the use of this report as a Digiset manual.

A Digiset simulation program is given, which translates "shorthand" Digiset code in machine code, performs a syntactical analysis of this code and produces papertape output that can be used as input to the Digiset. The program has some possibilities for the simulation of Digiset output on a line printer or plotter.

A character set is designed in order to use the Digiset as a line-drawing machine. Examples of line drawings made with the simulator and/or with the real Digiset, are shown and discussed.

KEY WORDS & PHRASES: Computer graphics, Digiset, Photo-typesetting, Line drawings, Simulation.
CONTENTS

ABSTRACT

0. INTRODUCTION 1

1. THE DIGISSET 2
1.1 Survey of the functioning of the Digiset 2
1.2 Storage of characters and microprograms 3
1.3 The coding of the characters 5
1.4 The instruction set of the Digiset 10
1.4.1 Direct instructions 11
1.4.2 Indirect instructions 11
1.4.2.1 Data transfer instructions 11
1.4.2.2 Layout instructions 14
1.4.2.3 Start and stop instructions 16

2. THE SIMULATOR 16
2.1 Input to the simulator 16
2.2 Simulator output 18
2.3 Specification of simulator properties 19
2.4 Errors 20
2.5 Description of the simulator 21
2.5.1 Level 1 22
2.5.2 Level 2 23
2.5.2.1 Auxiliary procedures 23
2.5.2.2 Digiset instruction procedures 24
2.5.2.3 Input translation 25
2.5.3 Level 3 26

3. EXERCISES 31
3.1 Exercise 1 31
3.2 Exercise 2 37
3.3 Exercise 3 43
3.4 Exercise 4 50
3.4.1 The alphabet 50
3.4.2 The line drawings 55
APPENDIX A
The Digiset simulator in ALGOL 60

APPENDIX B
1. Syntax of the Digiset machine code
2. Syntax of the shorthand code
3. Syntax of the Digiset code

APPENDIX C
Sample programs
"Crystal Structure"

after L. Geurts and L. Meertens,
produced on the Digiset.
0. INTRODUCTION

The underlying report contains a description of an ALGOL 60 program that simulates most of the behaviour of a Digiset-40 T1 electronic phototypesetter.

The writing of the program is part of a careful study of the possibilities of the photo-typesetter for on-line computer applications (computer graphics). The program in its present state is used to execute programs written or generated in Digiset code. This allows us to continue our study, without using the real apparatus, or alternatively, using it with better results, in the following ways:
- The design of standard Digiset programs (e.g. characters or microprograms) for on-line applications, like the production of line drawings, grayscale pictures and other computer graphics.
- To provide a testing facility for software that generates Digiset programs.
- The program checks the syntactic (and some of the semantic) correctness of Digiset programs. This is necessary because the Digiset itself does not provide any error detection whatsoever.

The report is divided in three parts:

In the first part (chapter 1) a short description of the Digiset-functions and internal organisation is given. The description is meant to allow further reading of the report, as well as the use of the Digiset and the Digiset simulator, without knowledge of the rather technical manuals.

In the second part (chapter 2) the simulator is discussed. First the set of input/output specifications is given. This information is sufficient for any programmer who wants to run an exercise on the simulator. The section also describes how the output is to be interpreted in order to get an idea about the real Digiset performance. The next section gives a sketch of the structure of the ALGOL 60 program (which is added as appendix A). Discussion of the simulator follows in three levels of detail. At each level there exists an error detection mechanism that gets full attention.

In the third part (chapter 3), a selection of exercises is given as an illustration of the possibilities of both simulator and simulated
machine. The collection of exercises will extend far beyond the contents of this report since every step in the development of the computer graphics project will be accompanied, and partly described by such a set of exercises.

The reader should be aware of the following typographic conventions throughout the report: Notions that are used to describe the Digiset are underlined at first occurrence. Usually some kind of definition is then contained in the context (e.g. typefont). Names of Digiset instructions are denoted in capital letters (e.g. KUR, SEQ, UAT), and are the same as those from the official manuals (german mnemonics!). Names of identifiers from the ALGOL 60 program are denoted in a different type (e.g. inpcore, first white element).

1. THE DIGISET

1.1. Survey of the functioning of the Digiset

The Digiset photo-typesetter generates hard-copy representation of graphical information by displaying drawings on the screen of a cathode ray tube, which are recorded on photographic paper or film. For a general overview of the Digiset hardware, see fig. 1.1.1.

![Diagram](image)

fig. 1.1.1. Flow of control in the Digiset.

The dimensions of the area of photographic material that can be exposed without transporting it, the usable recording area, are shown in fig. 1.1.2. In the same figure, the screen coordinates are shown, which
will be used in this report. The unit of length, is the point:
1 point = 0.376065 mm.

![Diagram](image)

fig. 1.1.2. Usable recording area.

The width \( w \) of the recording area depends on the photographic material used: \( w \leq 310 \text{ mm} \).

The recording takes place with a camera which can rewind the photographic material it contains. This offers the possibility of repeated exposure of the same piece of film, alternated with exposure of different pieces. The recording of drawings which are larger then the usable recording area is made quite easy this way.

The Digiset is equipped with a **core store** of 32 to 64 segments of 256 words. A word consists of 3 bytes of 8 bits each. The **disc store** of the Digiset is divided into 3247 addressable sectors, containing 744 non addressable words.

Drawings are composed from elementary characters like lines, text symbols etc. which are stored in the core- or disc store. Only characters stored in core can immediately be displayed, characters stored on the disc have to be transferred to core first. The input of the core address of a character (a **character call**) causes this character to be displayed. Positioning and transformation of characters are brought about by instructions which enter the Digiset through its input point. The Digiset permits the use of another programming unit for describing and drawing pictures, the **microprograms**. These are sequences of character calls and instructions which are stored and can be called like characters.

1.2. **Storage of characters and microprograms**

In order to be able to call characters and microprograms from core with a single one byte instruction, indirect addressing is necessary.
Therefore, the core store is divided into segments which can be addressed as a whole by a segment instruction which remains valid until replaced by a different one. Words in a segment thus selected, can be addressed by a one byte word address, the primary address. To make use of this mechanism, microprograms and characters (which must be defined on the same recording grid, see 1.3.) are grouped together in typefounts of at most 251 elements. These elements are stored in the image area of core, which may consist of several parts of different segments. The data of single characters or microprograms have to be loaded in contiguous parts of the image area however. The segment- and word address of the beginning of such an area are each stored in one word (the secondary address word) of a selected core segment, the address area of the typefount. A character thus stored, can be addressed by the word address of its secondary address, its so called primary address, as long as the appropriate segment instruction is valid.

The three bytes of the secondary address are used as follows (see table 1.2.1.)

<table>
<thead>
<tr>
<th>byte 1</th>
<th>byte 2</th>
<th>byte 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>word address</td>
<td>rel.seg.address</td>
<td>0,1 or 32 character</td>
</tr>
<tr>
<td>&quot;</td>
<td>&quot;</td>
<td>2 microprogram</td>
</tr>
</tbody>
</table>

Table 1.2.1. The secondary address word.

The first byte of the secondary address word contains the word address of the beginning of the character data in the image area.

The second byte contains the relative segment address of the character. This address has to be added to the segment address of the address area to find the segment address of the beginning of the character (or microprogram) data. The relative segment address has to be positive or zero. In the last case the first byte of the secondary address word has to be greater than 62, i.e. if the image area lies in the same segment as the address area, it has to lie above the word with address 62 (and above the address area).

The third byte of the secondary address word has value 2, if the addressed element is a microprogram. In the other case, it has the values
0,1 or 32 (see 1.3).
A special role is reserved for the secondary address word with primary address 62. This word contains the **typeface constant** which specifies among other things the dimensions of the recording grid on which the character is displayed (see 1.3).

The disc can only be loaded with complete typefounts i.e. a typeface constant followed by characters and/or microprograms. If elements of these typefounts are needed for typesetting, they have to be transported to core first. It is possible to transport typefounts as a whole as well as single characters or microprograms. While the disc is loaded with typefounts, the first core segment serves as an input buffer.

### 1.3. The coding of characters

Characters are defined on a square, in which a **recording grid** composed of rectangular cells is inscribed (see fig. 1.3.1a.)

![Recording Grid](image)

**fig. 1.3.1a.** Recording grid. **fig. 1.3.1b.** Character position.

In this square, three different **character base lines** (denoted by VBSII, I and 0) are defined which lie at the base of the square, at 3/16 l and 1/4 l above this base respectively, if l is the length of an edge of the square. If a character is displayed it is positioned in such a way, that the VBS 0 line coincides with the current position of the **type base line** on the screen. This type base line intersects the current beam position and runs parallel to the x-axis of the screen coordinate system.
When a character is displayed, the current beam position moves from point P to point Q in fig. 1.3.16. The amount by which a character can extend below the type base line is determined by its character base line (see fig. 1.3.16.).

The cells, and hence the grids are classified according to their size in three types, the B-, C- and D type (see table 1.3.1.).

<table>
<thead>
<tr>
<th></th>
<th>length in x direction</th>
<th>length in y direction</th>
</tr>
</thead>
<tbody>
<tr>
<td>B type</td>
<td>4/50 pt</td>
<td>4/120</td>
</tr>
<tr>
<td>C type</td>
<td>4/50 pt</td>
<td>4/60</td>
</tr>
<tr>
<td>D type</td>
<td>4/25 pt</td>
<td>4/60</td>
</tr>
</tbody>
</table>

table 1.3.1. Cell types and cell sizes.

According to the length of an edge of the square, recording grids are distinguished in grid types I, II, III, IV, and V. The size of these grids is given in table 1.3.2. below.

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>cell type</th>
</tr>
</thead>
<tbody>
<tr>
<td>grid type</td>
<td>x</td>
<td>y</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>50</td>
<td>120</td>
<td>50</td>
<td>60</td>
</tr>
<tr>
<td>II</td>
<td>100</td>
<td>240</td>
<td>100</td>
<td>120</td>
</tr>
<tr>
<td>III</td>
<td>200</td>
<td>480</td>
<td>200</td>
<td>240</td>
</tr>
<tr>
<td>IV</td>
<td>400</td>
<td>960</td>
<td>400</td>
<td>480</td>
</tr>
<tr>
<td>V</td>
<td>800</td>
<td>1920</td>
<td>800</td>
<td>960</td>
</tr>
</tbody>
</table>

table 1.3.2. Size of recording grids.

The distance (expressed in numbers of cells) from the base of the square to the character base lines, for different cell- and grid types is given in table 1.3.3. below.
<table>
<thead>
<tr>
<th>Character base line</th>
<th>B</th>
<th>C, D</th>
<th>Cell type</th>
<th>Grid type</th>
</tr>
</thead>
<tbody>
<tr>
<td>VBS II</td>
<td>0</td>
<td>0</td>
<td>0 0 0 0 0</td>
<td>0 0 0 0 0</td>
</tr>
<tr>
<td>VBS I</td>
<td>22</td>
<td>45</td>
<td>90 180 360</td>
<td>11 22 45 90 180</td>
</tr>
<tr>
<td>VBS 0</td>
<td>30</td>
<td>60</td>
<td>120 240 480</td>
<td>15 30 60 120 240</td>
</tr>
</tbody>
</table>

Table 1.3.3. Position of character base lines.

Which character line is used in the definition of a character by its character data, is specified in the third byte of its secondary address word, see table 1.3.4.

decimal value of third byte of secondary address word: 1 0 32
character base line VBS 0 VBS I VBS II

Table 1.3.4. Secondary address word and character base line.

Remark: The data of tables 1.3.1. through 1.3.3. are only valid when the GROa instruction has been given, otherwise they must be multiplied by the appropriate enlargement factors (see GRO and DAN instructions: 1.4.).

Although characters are defined on a rectangular grid, they can be displayed as italics. In this case, the recording grid is deformed to a parallelogram, as shown in fig. 1.3.2.

fig. 1.3.2. Italic characters.
Their exist three italic positions, corresponding with values of $\alpha$ (fig. 1.3.2.) of $22.5^\circ$, $25^\circ$ and $27.5^\circ$.

The cell type, grid type and italicy of a character are specified in the typeface constant. This constant consists of one word, the second and third byte of which are zero. The value of the first byte is given by the following formula:

$$\text{byte 1 typeface constant} = NO + 4NI + N2$$

in which NO specifies the cell type, NI the grid type and N2 the italicy of the character. NO, NI and N2 can have the values listed in table 1.3.5.

<table>
<thead>
<tr>
<th>NO cell type</th>
<th>NI grid type</th>
<th>N2 italicy</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>B</td>
<td>0</td>
</tr>
<tr>
<td>32</td>
<td>C</td>
<td>1</td>
</tr>
<tr>
<td>64</td>
<td>D</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>3</td>
</tr>
<tr>
<td></td>
<td></td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1.3.5. Constants, defining the typeface constant.

We only give a short description of the Digiset machine code for characters, since as input to the simulator, a much simpler, but just as powerful code can be used (see 2.1.), that will be translated in machine code before the simulation starts. This translated machine code can be used as input to the real Digiset.

Characters are coded as a sequence consisting of image lines, empty image line commands (LBL commands) and image line repetition commands (BLW commands). An image line is a column of the recording grid, in which for each cell it is defined whether it is black or white. An empty image line contains only white cells. The specification of the white and black elements in a non-empty image line consists of a sequence of alternating white and black elements. These elements are integer values, which denote the length of the pieces of the same colour in cell units (see fig. 1.3.3.). An image line has to start with a white element and to end with a black
element. An image line is displayed by alternately recording white- and black elements, starting from the character base line. However, the recorded length of the first white element, is one less than its coded length. This makes it possible to define characters that begin at the character base line, by giving their first white elements unit length.

![Image line diagram](image)

**Fig. 1.3.3. Character coding.**

The height of line elements can be specified in one byte if it is smaller than $2^7$, otherwise two bytes can be used. In this coding one bit (of the first byte) is used to indicate whether the line element is the last one of an image line or not. In case of two byte coding a second bit of the first byte indicates whether both available bytes are really used, or only the first one. This last possibility can be used for line elements shorter than $2^6$ cell units enclosed by lines longer than $2^7 - 1$ units. In table 1.3.6. the modes in which line elements of different length can be coded, are shown.

<table>
<thead>
<tr>
<th>length of element in cell units</th>
<th>element can be coded in one byte</th>
<th>element can be coded in two bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>$1 &gt; 127$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>$63 &lt; 1 \leq 127$</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>$1 \leq 63$</td>
<td>yes</td>
<td>yes</td>
</tr>
</tbody>
</table>

**Table 1.3.6. Coding modes for line elements.**
The LBL- and BLW commands have parameter parts in which the number of empty lines that have to be recorded, respectively the number of times the image line following the command has to be repeated, is specified. This parameter part can have values between 0 and 255. In addition, one bit in these commands is used to indicate the mode in which the image lines following the command are coded, i.e. to indicate, whether they are coded in one- or two-byte mode. In order to change the coding mode without introducing empty lines or repetitions, the LBL- or BLW commands with zero-valued parameter parts are used.

The ambiguity in the coding possibilities for line elements gives rise to the problem of finding the shortest possible coding for a character. If for instance, a character contains both lines with some long elements, as well as lines with only medium elements, every time these latter lines occur, the question arises if the coding mode has to be changed in order to minimize the memory area occupied by the character.

The answer to this question depends on:

1) The number of medium lines not separated by long lines.
2) The position of LBL- and BLW commands (if present) relative to the position of the medium lines in the coding sequence.
3) The position of the medium lines with respect to the beginning or end of the coding sequence.

The translation of the character code used as input to the simulator in the representation of machine code used during simulation, is such that the shortest possible code is generated.

1.4. The instruction set of the Digiset

The term instruction is used in this report, to denote all commands that control the functioning of the Digiset, with the exclusion of character data. Instructions enter the Digiset through its input point, or they are elements of microprograms. In the first case they are executed on-line, in the second, when the microprogram is activated. Instructions can be divided in direct instructions (consisting of one byte) and indirect instructions (of up to three bytes).
1.4.1. Direct instructions

The direct instructions are: NUL, KEN, UAK, ZWR and UAT.
NUL: This instruction does not cause any action of the Digiset.
KEN: The KEN instruction causes the three bytes following it to be ignored. These bytes may not contain the instruction sequences UAK SPE or KEN KEN however. In the simulator input, KEN may not occur in character- or microprogram data. The KEN instruction is only of use, when a magnetic tape unit provides the input, a situation not discussed here.
UAK: This instruction must precede every indirect instruction.
ZWR: The ZWR instruction is a spacing instruction. It causes the recording position to be shifted in the + x direction by an amount that must have been devised with the indirect instruction ZWQ.
UAT: This instruction serves as an end marker for character- and microprogram data, and for instruction sequences controlling data transfer from disc to core. It has always to be given twice in succession.

All other one byte commands are interpreted as primary addresses and cause the character or microprogram thus addressed, to be displayed or executed. The existence of this 5 direct instructions explains, why a typefount can have at most 251 elements although a core segment has room for 256 secondary addresses. The five values 0, 28, 62, 252 and 255 denote direct instructions and hence can not be used as primary addresses, without causing ambiguities.

1.4.2. Indirect instructions

The indirect instructions which are preceded by the UAK instruction can be divided in the following categories:
- Instructions which control data transfer in the Digiset and between the Digiset and the outside world.
- Layout instructions
- The START and STOP instructions.

1.4.2.1. Data transfer instructions

The data transfer instructions are: UBL, SEG, SPE, PLA, PLE and TRE.
UBL: The UBL instruction causes the Digiset to ignore all input until the
instruction sequence UAK START is encountered in the input. When the
simulator is used, UBL may not occur in microprogram- or character
data in the input stream.

SEG: By means of the SEG instruction, a core segment is addressed. This
address is given in the quantification byte following the instruction,
which can have values between 0 and 64. All word addresses that occur
in the input or in microprograms refer to the segment, last addressed
with SEG. There is, however, one exception to this rule. If a segment
instruction S0 is followed by a different segment instruction S1,
which itself precedes a microprogram call, which is not separated from
S1 by a character call, it is only used for the selection of the micro-
program. Inside the microprogram S0 is valid again, until other SEG
instructions are given there. After the execution of the microprogram,
S0 becomes active again, regardless of the SEG instruction given with-
in the microprogram. If one wishes S1 to remain active during and
after the microprogram call it has to be given twice in succession.

SPE: This instruction is used in two ways:
In the first place, it has to precede the indirect instructions PLA,
PLE and TRE (see below). In the second place, it causes the input in
core of the character, microporgram or type-face constant (with pri-
mary address 62) that follows the SPE instruction.
The instruction sequences are in these cases:

<UAK> <SPE> <62> <type-face constant> <UAT> <UAT> seq. 1.4.1.

<UAK> <SPE> <primary address> <secondary address>
<character data> <UAT> <UAT> seq. 1.4.2a.

<UAK> <SPE> <primary address> <secondary address>
<microprogram data> <UAT> <UAT> seq. 1.4.2b.

PLA: These instructions control the data transfer from the input point to
the disc and from the disc to the core store respectively.

TRE: If we denote seq. 1.4.1. by TC and seq. 1.4.2a,b. by CM, a typefount
is defined as:

<typefount>:: = <TC> <alphabet>
<alphabet> :: = <CM> | <alphabet> <CM>
The instruction sequence for the transfer of a typefount from the input point to the disc is:

<UAK> <SPE> <UAK> <PLE> <disc address> <UAT> <UAT>  
<typefount> <UAK> <SPE> <UAK> <TRE> <UAT> <UAT>  seq. 1.4.3.

This is the only way in which the instructions PLE and TRE can be used. It should be noticed, that during the input of data to the disc, segment 0 of core store is used as a buffer. Thus a PLE instruction causes the loss of the data stored there.

The transfer of a typefount from disc to core is brought about by the following instruction sequence.

<UAK> <SPE> <UAK> <PLA> <disc address> <UAT> <UAT>  seq. 1.4.4.

The disc address that always occupies two bytes specifies the number of the disc sector where the typefount begins. Typefounts are always stored without gaps on the disc. For the format of the disc address see fig. 1.4.1.

```
byte 2   byte 1

<table>
<thead>
<tr>
<th>T</th>
<th>0</th>
<th>0</th>
<th>D</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>9</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

sector number

fig. 1.4.1. The disc address.

The bit D serves to distinguish the two discs that can be simultaneously used. In the simulator only one disc is simulated and bit D is always set zero. In seq. 1.4.3. bit T is always set to zero. If bit T is set in the disc address of seq. 1.4.4., this sequence causes the transfer of only the disc addresses of the elements of the typefount to the selected address area in core. Every time such an element is called up for typesetting it is transferred from the disc to the segment of core, that lies immediately above the segment that contains
the address area with disc addresses. After this transfer it is displayed or executed. Data transfer from disc to core is the only data transfer possible during the execution of microprograms.

1.4.2.2. Layout instructions

The layout instructions that are part of the Digiset machine code are: VERA, VERO, VERU, ZWQ, RUCK, TAB, ZEV, KUR1, KUR2, KUR3, NOR, DAN, GROa through GROm, WID, SCHN and DREHO. The last two instructions are not incorporated in the Digiset simulator and will not be discussed here. Some of the instructions can be followed by at most two quantification bytes. In this case, the first byte always contains the least significant bits.

VERO: The instructions VERO and VERU cause the type base line to move up and down respectively, by an amount specified in the next two quantification bytes. The unit in which this displacement is measured is 1/32 point. The VERA instruction causes the type base line to move back to its neutral position (see fig. 1.1.2.). For reasons to be explained later, the VERA instruction has in the simulator the effect, of moving the type base line to the centre of the usable recording area.

ZWQ: The instruction ZWQ fixes the amount of spacing in the + x direction caused by the direct instruction ZWR. It is followed by two quantification bytes, which specify the displacement in units of 1/50 point.

RUCK: The RUCK instruction causes displacement in the - x direction of the recording beam. It has two quantification bytes, which specify the displacement in units of 1/50 point.

TAB: By means of the TAB instruction a line perpendicular to the x-axis is defined in absolute screen coordinates. The two quantification bytes specify the constant C in the line equation x = C, in units of 1/50 point.

ZEV: The ZEV and ZER instructions displace the type base line in the same way and have the same quantification bytes as the instructions VERO respectively VERU. In addition, the recording beam is moved to the point of intersection of the displaced type base line and the line defined by TAB.

KUR1: The instructions KUR1, KUR2 and KUR3 cause the image lines to be
KUR2: rotated clockwise round the points of intersection of these lines
KUR3: with the type base line. The rotation takes place over angles of 27.5°, 25° and 22.5° respectively. A KUR instruction remains valid until replaced by a different one, or a NOR instruction. When this last instruction is given, the italicy of the displayed character becomes the one, specified by the typeface constant.

GRO: These instructions cause the elementary rectangles of the recording grid to be enlarged by a definite factor. When a GRO instruction is used enlargement takes place both in the x- and y direction. In case of a DAN instruction, enlargement in the x direction is defined, while the enlargement in the y direction remains the one, defined by the GRO instruction last given. The enlargement factor is fixed by selecting a particular GRO instruction, or by the quantification byte of the DAN instruction, according to table 1.47.

<table>
<thead>
<tr>
<th>GRO instruction</th>
<th>DAN quantification</th>
<th>enlargement factor</th>
</tr>
</thead>
<tbody>
<tr>
<td>GROa</td>
<td>8</td>
<td>1.</td>
</tr>
<tr>
<td>GROb</td>
<td>9</td>
<td>1.125</td>
</tr>
<tr>
<td>GROc</td>
<td>10</td>
<td>1.25</td>
</tr>
<tr>
<td>GROd</td>
<td>11</td>
<td>1.275</td>
</tr>
<tr>
<td>GROe</td>
<td>12</td>
<td>1.5</td>
</tr>
<tr>
<td>GROf</td>
<td>14</td>
<td>1.75</td>
</tr>
<tr>
<td>GROg</td>
<td>16</td>
<td>2</td>
</tr>
<tr>
<td>GROh</td>
<td>18</td>
<td>2.25</td>
</tr>
<tr>
<td>GROI</td>
<td>20</td>
<td>2.50</td>
</tr>
<tr>
<td>GROj</td>
<td>22</td>
<td>2.75</td>
</tr>
<tr>
<td>GROk</td>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td>GROI</td>
<td>28</td>
<td>3.5</td>
</tr>
<tr>
<td>GROM</td>
<td>32</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 1.4.7. Enlargement factors.

An enlargement remains valid until a new GRO or DAN instruction occurs. When a GRO instruction is given, the typeface constant stored in the segment addressed at this time, is evaluated. This typeface constant
will be used for the display of characters called after the GRO instruction. From, this it follows that, before the first character call a GRO instruction has to be given and before the first GRO instruction, a SEG instruction.

WID: The WID instruction causes a character call or microprogram call following it, to be repeated the number of times specified in the two quantification bytes of the instruction.

Remark: The character is displayed (or the microprogram executed) once more, than the number of times it is repeated.

A microprogram that contains a WID instruction may not be repeated itself by a WID instruction.

1.4.2.3. Start and stop instructions.

START: A Digiset program has to begin with the instruction sequence: UAK
STOP : START and ends with UAK STOP. In addition, the START instruction is used to terminate the effect of the UBL instruction (see 1.4.2.1.).

2. THE SIMULATOR

2.1. Input to the simulator

The input to the simulator has the following form:

<parameter setting> <Digiset code representation>

The first part of this sequence will be discussed in section 2.3. after the description of the simulator output.

A sequence of Digiset instructions and parameters is recognized by the simulator if:

- The representation of a byte is separated from the representation of another one by a semicolon.
- Instructions are represented by their names (see 1.4.).
- Microprogram data are enclosed between square brackets.
- Parameters and addresses are represented by decimal numbers.
Examples:
- UAK; START; UAK; DAN; 8; UAK; ZWQ; 10; 0; ...
- UAK; SPE; UAK; PLA; 21; 0; UAT; UAT; 63; 68; 112; ...
- UAK; UBL; byte; ..... byte; UAK; START; ....

The instruction sequences of microprograms are coded in the same way. Character data, as defined in seq. 1.4.2a., are coded as follows:

\[
\text{<character data> ::= <character body> ;}
\]

\[
\text{<character body> ::= <char element> | <char element> <char body>}
\]

\[
\text{<char element> ::= ( repeated empty line ) | (<repeated line>)}
\]

\[
\text{<repeated empty line> ::= <number of empty lines>}
\]

\[
\text{<repeated line> ::= <number of lines>, <first white element>,}
\]

\[
\text{ <black white list>, <last black element>}
\]

\[
\text{<black white list> ::= <black element>, <white element> |}
\]

\[
\text{ <black element>, <white element> , <black white list>}
\]

\[
\text{<first white element> ::= <positive integer>}
\]

\[
\text{<last black element> ::= <positive integer>}
\]

\[
\text{<white element> ::= <positive integer>}
\]

\[
\text{<black element> ::= <positive integer>}
\]

\[
\text{<number of lines> ::= <positive integer>}
\]

\[
\text{<number of empty lines> ::= <non zero integer>}
\]

\[
\text{<number of lines> (respectively <number of empty lines>) denotes the number of times the line (respectively the empty line) has to be displayed in succession. If <number of empty lines> is positive the empty lines are displayed in the recording direction, otherwise in the opposite direction. Thus an empty line command with a negative parameter gives backspacing within a character. The value of <first white element> is the length in cell units plus one of the first white element of the image line. All other white or black elements denote exactly the length of the corresponding parts of an image line in cell units.}
\]

Example: (3) (4,2,1);

This character is built up out of three empty image lines, followed by 4 image lines each consisting of a white element of length one followed by a black element of length one. Characters coded this way, are translated by the simulator in a representation of Digiset machine code (see 1.3.).
This translation takes place in such a way, that the resulting code is the shortest possible one. Hence there is no need to consider the "shortest code problem" (see 1.3.) while using the simulator.

Characters and microprograms from a single typefount that is loaded without being interrupted by the input of elements from a different typefount, can be coded in shorthand code, without specifying secondary word addresses and relative segment addresses. These two addresses must then be replaced by the single pseudo address 256. In this case, the data are loaded consecutively, starting at word address 0 of the segment, following the address area of the typefount. The secondary word addresses and segment addresses are then calculated by the simulator.

2.2. Simulator output

The simulator can produce its output on the plotter, on the lineprinter and on paper tape. When the lineprinter is used, every cell of a C type recording grid corresponds to one print position on the lineprinter paper. This situation fixes the scale of lineprinter drawings. The C type grid has been chosen, because its cells are almost square. It is not possible to display characters coded on B- or D type grids with the lineprinter. This would not make sense anyhow, for it would change the scale of characters, while the scale of layout instructions would remain the same. The height of the part of the usable recording area that is simulated on the lineprinter is 144 C type cell units, while the height of the real recording area is 1440 units (see fig. 2.2.1.). However, the length can be chosen freely. (see 2.3).

The instructions GRO, KUR, NOR and DAN cannot be used in lineprinter drawings. In addition, italic characters are displayed upright.

When using the plotter, every C type cell is displayed as a rectangle of 12 x 1mm, B- and D type cells can be used too, and are displayed on the same scale. (B type cell 12 x 0.5mm, D type 2.4 x 1mm). Black elements are displayed black, by shading them.

The maximal height of a plotter drawing is 275 C type cell units, the maximal length 1250 units. The length of the real usable recording area is 9900 units (see fig. 2.2.1.).
Fig. 2.2.1. Simulated recording area.
Units: C type cell units.

When using the plotter, every instruction described in this report, can be used. The range of the parameters of the positioning instructions that can be simulated is much smaller than the range for the real Digiset, because of the smaller recording areas. For the same reason the effect of the VERA instruction is changed into positioning the type base line as a line, that lies halfway between the base and the top of the simulated recording area.

Apart from drawings, the output contains a numbered listing of all input elements. This listing is used for error identification (see 2.4.).

Furthermore, the simulator can produce a paper tape with the Digiset machine code that corresponds to the short hand code used as input. In this way, tapes that can directly be used as input to the Digiset are produced.

2.3. Specification of simulator properties

The input to the simulator has to be preceded by some simulation parameters. For most simulations it is not necessary to supply the full capacity of the real machine. Program time and space can be saved by specifying the smallest possible configuration. Output time can be reduced by directing output to the lineprinter instead of to the plotter for small exercises. The format of the specification is:

\[ <N_1>; <N_2>; <N_3>; <N_4>; <N_5>; <N_6>; \]

in which \( N_1, \ldots, N_6 \) are integers.

\( N_1 \) specifies the number of core segments \((1 \leq N_1 \leq 64)\), \( N_2 \) the number of disc sectors \((1 \leq N_2 \leq 3274)\). \( N_3 \) is an upper bound for the number of image lines
that can occur in a character \((1 \leq N_3 \leq 15000)\) and \(N_4\) is an upper bound for the number of bytes that the input would occupy, if coded in Digiset machine code \((4 < N_4)\).

If \(N_5 = 1\), the output is plotted, else it is displayed on the lineprinter. In the latter case, the simulated recording area has a length of \(N_5\) image lines \((1 \leq N_5 \leq 15000)\).

The syntax of \(<N_6>\) is:

\[
<N_6>:: = p; \mid <\text{empty}>
\]

When \(<N_6> = p\), a papertape with Digiset machine code is produced.

2.4. Errors

If the input to the simulator contains syntactic errors or semantic errors that are discovered during simulation (an attempt to draw outside the recording area, for example), an error message is given. The same happens if the simulation parameter setting contains errors.

The error detecting mechanism has the property that input errors are discovered before they can cause the simulation program to be stopped by the machine on which it runs. In this way it is possible to localize errors, without knowledge of the functioning of the simulation program.

Errors can be classified in three types:

1) Parameter setting errors.
2) Errors discovered during translation of the input in Digiset machine code.
3) Errors discovered during execution of the translated Digiset program.

All errors of type 1 or 2 are discovered in one scan, after which they are listed, and the simulation stops. If a type 3 error occurs, an error message is given, and the output which has already been generated is displayed, after which the program stops.

An error message for type 2 or type 3 errors consists of:
- A description of the error.
- The number of the input element that causes the error. This number
corresponds to the number of the element in the listing of the input.
- If the error is caused by a microprogram instruction, the number of this
  instruction, relative to the beginning of the microprogram, is also
given.
- If the error is caused by character data, the number of the image line
  that contains the error, is also given.

Examples:

1) input listing: 31 (10) (20,0,1) (3,1,10)
   error message: zero valued line element
                   number input element 31
                   line number 2

2) input listing: 11 PLA;
   12 10;
   13 0;
   14 UAK;
   error message: PLA not followed by UAT; UAT;
                   number input element 14

If a parameter setting error occurs, the error message consists only of a
description of the error.

There is one important semantic error which does not always cause an
error message. If a core area is addressed that has not been loaded with
correct character or microprogram data, it is nevertheless possible that
the data present can be interpreted as microprogram or character data.
Only if this is not the case, an error message will be given.

2.5. Description of the simulator

The simulation program will be discussed with the aid of fig. 2.5.1.
In this figure, the block structure of the program is shown by rectangles.
If a block is a procedure body, this fact is indicated by the text in the
block. In fig. 2.5.1, from left to right three levels of detail can be
seen. We will describe the program by discussing these levels in the same
order.
2.5.1. Level 1

The simulator consists of the simulation parameter setting followed by the simulation program proper. In the parameter setting part the output mode is specified, as well as upper bounds for some array indices. This part contains the procedure parameter error, which gives error messages in case of errors in the specification parameters of the input.

The block that contains the real simulator, begins with the declaration of the integer arrays dtape, core, disc and the boolean array line-printer output. Array dtape will be filled with the input in a representation of Digiset machine code. This representation, which is a result of translation of the input will be used throughout the simulation. It is defined as follows:

Every instruction or character byte is coded as a decimal number of at most eight digits. Each digit corresponds to one bit of machine code and can have the values zero or one. Instruction quantification bytes and addresses, which occur in the simulator input as decimal numbers with values between 0 and 255, are left unchanged, however. The arrays disc and core simulate the disc and core memory of the Digiset. The boolean array line-printer output represents the usable recording area, in case the output has to be displayed on the lineprinter. Every array element corresponds to one cell of a C type grid. It gets the value "true" when a cell is found to be part of the output drawing, otherwise the value "false". At the end of the simulation for every element with value "true", a $ sign is printed.

The procedures which follow the array declarations can be classified as auxiliary procedures and procedures that bear the names of Digiset instructions and simulate them.

The machine code table links the names of the instructions with the representation of their machine code, while the enlargement factor table contains factors used in the GRO and DAN instructions.

The simulation starts with the translation of the input into machine code and the storage of this code in the array dtape. Thereafter this translated input is processed by the procedure interpret (see 2.5.2.1.).
2.5.2. Level 2

2.5.2.1. Auxiliary procedures

The integer procedures **read dtape, read core and read disc** read one byte from **dtape, core** or **disc** respectively, every time they are called. They increase array indices in such a way, that the next time they are called they will read the next byte (unless indices are reinitialized between calls). The integer procedures **rdtod (bdisc), rdtoc (mic)** and **rdtodoc (bddenisc)** read from **dtape or disc**, from **dtape or core** and from **dtape or disc or core** respectively, depending on the values of their boolean parameters. They are used in procedures that must have access to different data sources.

The procedure **error (string, mic, char, stop)** prints its string parameter when it is called. This string contains the description of the error which caused the procedure call. In addition, the number of the input element that caused the error is printed. If the boolean parameters **mic and/or char** are "true" the number of the microprogram instruction and/or the number of the image line which contains the error is/are also printed. If the parameter **stop** is "true", the simulation stops and the output sofar obtained is printed, otherwise the simulation continues.

The procedure **print relative (init linenumerator, init printpos, deltay, deltay, linenumerator max, print)** is used to store drawings in the array line printer output. If **init linenumerator** and/or **init printpos** are "false" **deltay and/or deltay** are absolute screen coordinates, otherwise they are relative to the point of the drawing that is defined by the previous execution of **print relative**. If the boolean **print** has the value "true" the array element corresponding to the point of the drawing defined by the parameters of **print relative** gets the value "true". The variable **linenumerator max** is used to check whether the print lies inside or outside the usable recording area. In the last case, an error message is given.

Procedure **plotrelative (initial value, deltay, deltay, ipen, kurv)** is analogous to **print relative**. However, it causes direct movements of the plotter pen instead of storing the output. The coordinates **deltay and deltay** are absolute or relative simultaneously depending on the value of the boolean **initial value**. The integer **ipen** is used to indicate, whether
the pen movement has to take place with pen up or down. The value of the integer \textit{kur} indicates which of the KUR or NOR instructions is valid at the time of the procedure call. If this is a KUR instruction, movements in the \textit{y} direction occurring during character display are changed into sloped movements. In addition an error message is given if pen movements are towards points outside the recording area.

Procedure \textit{interpret (instruction, bool rddisc, mic, plotter)} is used to interpret machine code bytes, not belonging to character data. It interprets \textit{dtape} data, microprogram data from \textit{core} and \textit{disc} data during execution of the PLA instruction. The procedure is called with the decimal value of an instruction or address byte as value of its parameter \textit{instruction}. If this value is the coding for UAK, another byte is read from \textit{dtape}, \textit{disc} or \textit{core} depending on the values of the booleans \textit{boolrddisc} and \textit{mic}. This reading takes place with procedure \textit{rdtodoc (boolrddisc, mic)}. This second byte will cause a call from the procedure that corresponds with the instruction that is coded by it. If the value of \textit{instruction} does not designate the UAK instruction, a direct instruction is executed, or a character or microprogram is called. The boolean \textit{plotter}, which has the value "true" when the output has to be plotted, is passed to the instruction simulating procedures.

2.5.2.2. Digiset instruction procedures

The procedures \textit{seg, dan, gro, kur, nor, ken, ubl, init, zwr, ruck, vero, veru, vera, zer, zev, zwo} and \textit{tab} simulate the effect of the Digiset instructions with the same name. This simulation is straightforward, and will not be discussed here.

The procedure \textit{stop} stops the simulation. If line printer output has been generated, \textit{stop} causes a print-out of the array \textit{lineprinter output}.

The procedure \textit{inpcore} stores typefount elements in \textit{core}. It reads the data from \textit{dtape} or \textit{disc} with procedure \textit{rdtod (boolrddisc)}. These data are stored in \textit{core} by the local procedure \textit{storec (x)}. Data are read and stored until the first UAT instruction sequence not belonging to UAK PLA byte byte UAT UAT is recognized. During storage, the values of secondary address word bytes and typeface constant bytes are checked. If these bytes have non allowed values an error message is given. Procedure \textit{inpcore} is used
for direct storage into core, during execution of the PLA instruction and when a typefount element is addressed that has to be brought in from disc by the single character transfer method.

The procedure spe first reads the next byte from dtape, disc (during execution of pla) or core (during execution of a microprogram). If this byte is not an UAK byte, procedure inpcore is called, otherwise the next instruction byte is read, to determine whether procedure pla, ple or tre has to be called. In case of non allowed data transfer instruction (for instance, PLE within a microprogram) an error message is printed.

The procedure call typefount element is used, whenever interpret encounters a byte, which is a core address. In this procedure, first it is determined whether the data of the addressed element still have to be brought in from disc (single character transfer) and whether the element is a character or a microprogram. In the last case, the data are read by procedure interpret which causes the execution of the microprogram. If the element is a character, the position of the character base line is calculated first. It is determined by the typeface constant and the third byte of the secondary address word. The y coordinate of the character base line is stored in the variable y position. Then the character data are read which cause calls of the procedures lbl, blw and image line. Procedure lbl causes the appropriate number of empty image lines to be displayed, while blw assigns values to repeat (the number of repetitions) and to line repetition ("true"). Procedure image line displays the line a number of times equal to the value of repeat. If the addressed data cannot be interpreted as character data clearly a wrong core area is addressed. In this case, the procedure address error prints an error message.

2.5.2.3. Input translation

In the block labelled input translation (fig. 2.5.1.), the input is translated into the representation of Digiset machine code, already discussed. This is done by procedure code switch. The input is read symbol-wise by integer procedure skip, which delivers different values for different symbols read and which skips blanks and "new card" symbols. As a side effect, skip produces a listing of the input.

The procedure read input element scans input strings enclosed between
semicolons, by successive calls of \textit{skip}. These strings can contain a quantification byte, an address, an instruction name or a character definition (which is recognized by an opening parenthesis). In case of a quantification byte or an address, the corresponding numerical value is stored in \textit{dtape}. If an instruction name is read, its machine representation is stored in \textit{dtape} as an integer consisting of only zeros and ones. A character definition is processed by the procedure \textit{character} (see 2.5.3.), which delivers a sequence of bytes stored in \textit{dtape}. As a side effect of read \textit{input element}, all strings are enumerated. The procedure \textit{listingnumber} prints the number of each input string on a new line, prior to its listing by \textit{skip}. Input strings that cannot be recognized or processed in this way (because of the occurrence of non allowed symbols, for instance), cause error messages. The translation continues, until the \textsc{stop} instruction is recognized in the input.

2.5.3. Level 3

Procedure \textit{pla}, which is called from \textit{spe}, transfers a typefount from \textit{disc} to \textit{core}. First the disc address of the typefount is evaluated, and from its first address byte it is determined whether the transfer mode is single character or not. In the last case, procedure \textit{interpret} scans the typefount data in \textit{disc}, which results in the transfer of the typefount to \textit{core}, by \textit{spe} calls, causing \textit{inc\textit{core}} calls. When the instruction sequence UAK SPE UAK TRE UAT UAT is recognized, the transfer stops. In the case of single character transfer, \textit{pla} scans the typefount for UAK SPE instructions. If these instructions are not followed by UAK, they mark the beginning of a typefount element and are followed by a primary address. The sector address (for the first character) or the sector address -1 for every other character), is then copied in the secondary address word of \textit{core} that has the primary address just found.

Procedure \textit{ple} checks typefount data, after which they are stored in \textit{disc}, or an error message is given and the simulation stops. Data are stored byte-wise by the local procedure \textit{stored} ($\lambda$), which is called with the machine code representation of the byte to be stored as value of $\lambda$. Procedure \textit{ple} starts with evaluating the disc address and checking whether the typefount begins with a correct typeface constant. Then procedure
element onto disc is called. In this procedure, it is checked (by procedure check) if the next instructions are UAK SPE and whether the following instructions are UAK TRE UAT UAT or a legitimate primary and secondary address. If the TRE sequence is found, the transfer is ended. The secondary address is used to assign value "true" or "false" to the boolean microprogram, which determines whether the following data are interpreted as microprogram or character data. Then procedure uatuat (microprogram), which is local to element into disc, is called. If microprogram is "false", all subsequent data are stored, until the first UAT UAT instructions are encountered, otherwise all data are stored until the first occurrence of UAT UAT not being part of UAK PLA byte byte UAT UAT. Finally, uatuat calls element into disc again.

The procedure image line (cellwidth, cellheight, shading, remainder, repeat, y position, plotter, line repetition, kurn) is used to display an image line of a character. The parameters cellwidth and cellheight specify the size of the recording grid, taking into account the GRO and DAN instructions last given. The integers shading and remainder specify the way in which black elements are shaded. These parameters, which are only used when the output is plotted, are calculated using the typeface constant evaluated during the last call of gro. The integer repeat and the boolean line repetition obtain their values from procedure blw. The integer kurn which specifies the italics of a character, gets its value from procedure kurn or from the typeface constant. The boolean plotter specifies, whether the output is on the plotter or the lineprinter. Finally, the real variable y position, specifies the position of the character base line. It is calculated by procedure call typepoint element, which is global to image line. The data defining an image line are read element by element from core by the integer procedure read line element. These data are decoded into the length of successive line elements and are displayed by alternating calls of the procedures black and white. These procedures use the auxiliary procedures plotrelative or printrelative (see 2.5.1). If a line has to be repeated, the length of its elements are stored at the same time by procedure store repeated line (x) in the integer array repeated line. The integer parameter x passes the value to be stored. When the last black ele-
Fig. 2.5.1. The simulator program.
# LEVEL III

**proc pla**
transfer of single characters or of complete typefounds

**proc ple**
- **proc stored**
- **proc check**
- **proc element onto disc**
  - **proc utuat**
  - store typefound element
  - call element onto disc
  - call check
  - call utuat or end transfer
  - check typeface constant,
  - call element onto disc

**proc image line**
- **proc white**
- **proc black**
- **proc character base**
- **proc bie**
  - **proc read repeated line**
  - display last black element
  - and repeat line if necessary
  - call character base
  - call black and white alternately

**proc character**
- **proc code short long list**
- **proc read char element**
- **proc free core**

recursive call of character or of code short long list, followed by a call of character
ment of the line is encountered, procedure *ble* is called. This procedure
displays the last element and repeats the whole line as many times as re-
quired. When repetition has to take place, *ble* uses the data from the array
*repeated line*, which it reads with integer procedure *read repeated line*.
Every time an image line is displayed, procedure *character base* is called
from *ble*, which changes the recording position to the point on the charac-
ter base line where the next image line has to begin.

Procedure *character \([k, \text{start}]\)* is used to translate character input
into the shortest possible machine code. For this purpose image lines of
the character to be coded, are split by the procedure into possibly empty
groups of short lines, followed by one long line. These groups can be
coded independently from each other, because of the fact that their short
lines for which coding is ambiguous, are enclosed between long lines, that
have to be coded in two-byte mode. This process proceeds as follows (see
fig. 2.5.2.). Procedure *character \([k, \text{start}]\)* reads the k-th line of a se-
quence of the type just described and stores the length of the line ele-
ments and the number of times the line has to be repeated in array *AA*.
Number of repetitions is stored in array element *AA* \([k, 1]\) and the length
of the j-th line element in *AA* \([k, j]\). This reading and storage is done by
the local procedure *read char element \((k, j)\)* which recursively calls itself,
thereby increasing \(j\) by one. If the line just read is a long line or the
last line of the character, the sequence of \(k\) lines (whose data are stores
in *AA*) is coded by procedure *code short long list \([k, \text{start}]\)*. We will not
discuss this procedure here because of the fact that this would require
detailed knowledge of Digiset machine code. After execution of *code short
long list*, it is checked whether there are still lines to be coded. If
this is the case, *character \([1, \text{false}]\)* is called, and the whole process
starts all over again, otherwise *character* is left. If the last line read
by *character \([k, \text{start}]\)* is a short line or an empty line however, *charac-
ter \([k+1, \text{start}]\)* is called and hence the next line is read. The translation
starts with a call of *character \([1, \text{true}]\)*. The value "true" of the boole-
ean start indicates that the coding of the character still has to begin,
and hence, that the coding mode is still undefined.

Furthermore, *character* contains the procedure *free core* (word pointer,
seg pointer), which is used to calculate the secondary word address and relative segment address of typefount elements, that are input without specified addresses (see 2.1.).

3. EXERCISES

All exercises are generated in shorthand code, either by hand or by computer programs. Although the syntax of Digiset programs is very simple (cf. appendix B), most exercises have past the simulator several times before the input was syntactically accepted. This convinced us of the fact that a syntactic check on hand written code is necessary, and that a syntactic check on computer generated code is a very useful testing facility for that computer program. Nevertheless we only give as examples simulator output produced by correct input.

Exercises on the lineprinter serve mainly to study single characters and alphabets. The mutual connection, transformation and positioning of characters can be judged on a plotter simulation.

After a succesful run on the simulator, most exercises have also been carried out on the Digiset itself. These pretested exercises proved to be failsafe: each run on the Digiset was faultless. For comparision both results are published here.

The exercises are instructive as:
- examples of Digiset programs
- illustrations of the working principles of the Digiset
- illustrations of the performance of simulator and Digiset
- an illustration of the capacity needed for storage of pictures as well as alphabets.

3.1. Exercise 1:

Define an alphabet containing one character, and display the character. The character chosen is an alto clef taken from music notation.
Each Digiset (sub)program contains the following possibly empty,
sequences of instructions:
- initialisation
- input of character data
- display of characters, mixed with layout instructions (positioning etc.).

The program together with some explaining comment is listed below:

UAK;                 / begin of program
START;
UAK;                 / select segment Ø
SEG;
Ø;                   /
UAK;                 / select base line
VERA;
UAK;                 / input to core segment
SPE;
62;                  / address of typeface constant
32;                  /
Ø;                   / typeface definition
Ø;
UAT;                 / end of input to pa 62
UAT;
UAK;                 / input to pa 63
SPE;
63;
1;                   / word address
1;
Ø;                   / character type
(7,1,86)(4,82,5)(4,1,86) / image line information
(1,42,3)(1,41,5)(1,40,7)  / # of lines followed by
(1,39,9)(1,38,11)(1,9,6,22, / pairs of white and
13,22,6)(1,7,10,19,15,19,10) / black values.
(1,3,17,7,33,7,17)(1,2,18,7,12,9,12,7,18)
(1,2,17,8,10,13,10,8,17)(1,1,17,10,8,15,8,10,17)
(1,1,5,4,7,13,7,13,7,4,5)
(1,1,5,5,5,17,5,11,5,17,5,5,5)
(1,1,6,27,6,7,6,27,6)(1,1,6,27,7,5,7,27,6)
(1,1,7,27,7,3,7,27,7)(1,2,6,27,7,3,7,27,6)
(1,2,7,25,8,3,8,25,7)(1,3,7,21,10,5,10,21,7)
(1,3,9,17,11,7,11,17,9)(1,4,9,13,13,9,13,13,9)
(1,5,33,11,33)(1,5,31,15,31)(1,6,29,17,29)
(1,7,27,19,27)(1,8,25,21,25)(1,9,22,25,22)
(1,11,18,29,18)(1,12,15,33,15)(1,14,10,39,10)
(1,18,2,47,2);

UAT;      / end of input for pa 63
UAT;      / now the alto clef is defined
UAK;      / as character #63 in segment Ø.
VERU;     / position below base line,
12Ø;      / into usable recording
Ø;        / area.
UAK;
ZWØ;      / define space width
2ØØ;      
Ø;
UAK;      / define enlargement factor
GROa;
63;       / display character
UAK;
STOP;     / end of program

The result of this program for the lineprinter is given in fig. 3.1.1. Each line on the printer corresponds with one image line. Fig. 3.1.2. is the plotter simulation. Its shape is exactly equal to the character on the first line of the Digiset result (see fig. 3.1.3). The plotter simulation and the first character of the Digiset output have the same enlargement factor. For the Digiset exercise we have repeated the character over the whole range of enlargement factors (first line), followed by the whole
fig. 3.1.1. Alto clef character,
line printer simulation.
fig. 3.1.2. Alto clef character, plotter simulation.
fig. 3.1.3. Alto clef series,
Digiset output.
range of width enlargements with constant maximal vertical enlargement (second line), followed by enlargement factor 2 (GROg) combined with the four available italics.

The complete program of the lineprinter exercise takes about 300 bytes in Digiset code, namely: 20 bytes for initialisation, 270 bytes for character definition and 10 bytes for positioning and display.

3.2. Exercise 2

The chemical formula

\[
\begin{array}{c}
\text{H} \\
\text{O} - \text{C} - \text{C} - \text{Cl}.
\end{array}
\begin{array}{c}
\text{H} \\
\text{H} \quad \text{NH}_2
\end{array}
\]

The formula is built up with the following alphabet:

- characters: O, H, N, C, 2, L, -, \, ;
- microprogram: - CH\textsubscript{2} \text{H} (namely:

\[
\begin{array}{c}
\text{H} \\
\text{C} \\
\text{H}
\end{array}
\]

Although the smallest possible character size was chosen, the lineprinter result still extends over 5 pages. Therefore only a part of the result is shown here (see fig. 3.2.1.). The plotter simulation takes a considerable amount of plottime. This is about the size of the largest picture one can produce on the plotter (see fig. 3.2.2.).

As far as actual drawing is concerned, the usefulness of the simulator lies mainly in the field of judging individual characters and character connections. Apart from this the best profit from the simulator is obtained by using it for syntactic pre-checking of the Digiset input, and for semantic checking by monitoring the Digiset reactions on the input without actually drawing. This produces error messages of the type: outside recording area, non existent character called, etc.
fig. 3.2.1. Chemical formula, fragment of line printer simulation (reduced size)
fig. 3.2.2a. Chemical formula, plotter simulation

fig. 3.2.2b. Digiset output corresponding to plotter simulation
The Digiset program takes about 350 bytes, this includes the character definitions (150 bytes) and the microprogram (50 bytes). The microprogram is called twice in succession by means of the WID command. Note that the chemical formulae \( \text{H} - (\text{CH}_2)_n - \text{NH} - \text{CL} \), with \( n = 1, 2, \ldots, 2 + 16 \), could have been drawn with a program that differs from the one listed below by not more than two bytes.

```
UAK; START;       / begin of program
UAK; SEG; \( \emptyset \);       / define core segment
UAK; TAB; \( \emptyset \); \( \emptyset \);       / define tab position
UAK; VERA;       / select type base line \( y=\emptyset \)
UAK; ZEV; \( \emptyset \); \( \emptyset \);       / select tab position \( x=\emptyset \)
UAK; SPE; 62;       / define typeface constant:
32; \( \emptyset \); \( \emptyset \);       / grid \( CI = 50 \times 60 \)
UAT; UAT;       / end of typeface data.
UAK; SPE; 63;       / \#63 = letter 0
1; 1; 1;       / address + character base line
\( (4)(2,5,2\emptyset) \)
\( (12,5,2,16,2) \)
\( (2,5,2\emptyset)(4) \);       / 24 image lines
UAT; UAT;
UAK; SPE; 64;       / \#64 = letter H
6,1,1;
\( (4)(2,5,2\emptyset) \)
\( (12,14,2)(2,5,2\emptyset)(4) \)
UAT; UAT;
UAK; SPE; 65;       / \#65 = letter N
1\( \emptyset \); 1; 1;
\( (4)(2,5,2\emptyset)(1,22,2) \)
\( (1,2\emptyset,2)(1,18,2)(1,17,2) \)
\( (1,15,2)(1,14,2)(1,12,2) \)
\( (1,11,2)(1,1\emptyset,2)(1,8,2) \)
\( (1,7,2)(1,6,2)(2,5,2\emptyset) \)
\( (4) \);
UAT; UAT;
```
UAK; SPE; 66; \hspace{1cm} / \#66 = \text{letter C}
21; 1; 1;
(4)(2,5,2\emptyset)(12,5,2)
(2,5,4,12,4)(4);
UAT; UAT;
UAK; SPE; 67; \hspace{1cm} / \#67 = \text{cipher 2}
27; 1; 1;
(2)(1,1,1,5,3)(1,1,2,7,1)
(1,1,1,1,1,6,1)(1,1,1,2,1,5)
(1,1,1,3,1,4,1)(1,1,1,4,5)
(2);
UAT; UAT;
UAK; SPE; 68; \hspace{1cm} / \#68 = -
36; 1; 1;
(1)(2\emptyset,14,2)(1);
UAT; UAT;
UAK; SPE; 69; \hspace{1cm} / \#69 = |
41; 1; 1;
(2,1,2\emptyset);
UAT; UAT;
UAK; SPE; 70; \hspace{1cm} / \#70 = \text{microprogram}
\hspace{1cm} / \text{for - CH}_2
43; 1; 2; \hspace{1cm} / \text{address + code for microprogram}
[ \hspace{1cm} / \text{start of microprogram}
64; \hspace{1cm} / \text{display H}
UAK; VERO; 6\emptyset; \emptyset; \hspace{1cm} / \uparrow \text{(move)}
UAK; RUCK; 52; \emptyset; \hspace{1cm} / \rightarrow \text{(move)}
66; \hspace{1cm} / \text{display C}
UAK; VERO; 6\emptyset; \emptyset; \hspace{1cm} / \uparrow
UAK; RUCK; 52; \emptyset; \hspace{1cm} / \rightarrow
69; \hspace{1cm} / \text{display |}
UAK; VERO; 42; \emptyset; \hspace{1cm} / \uparrow
UAK; RUCK; 52; \emptyset; \hspace{1cm} / \rightarrow
64; \hspace{1cm} / \text{display H}
UAK; VERU; 1Ø2; Ø; / +
68;
UAK; VERU; 1Ø2; Ø
] ; / display -
UAT;
UAT;
/ end of program
UAK; SPE; 71;
8Ø; 1; 1 ;
(2)(2,5,2Ø)(14,5,2)(2); / end of microprogram input
UAT; UAT;
/ end of text data
UAK; GROa;
/ start of actual program
UAK; VERU; 3Ø; Ø;
/ define size + typeface
UAK; WID; 1; Ø;
/ position in common recording
7Ø;
/ area for lineprinter, plotter
65; 64; 68;
/ and Digiset.
UAK; VERU; 1Ø2; Ø;
/ display H 0 -
64;
UAK; WID; 1; Ø;
/ +
UAK; RUCK; 188; Ø;
/ display - CH2 - CH2
7Ø;
65; 64; 67;
/ display NH2
UAK; VERO; 6Ø; Ø;
/ +
UAK; RUCK; 188; Ø;
/ -
69;
UAK; RUCK; 52; Ø;
/ display l
69;
UAK; VERO; 42; Ø;
/ +
66;
/ display C
UAK; VERO; 6Ø; Ø;
/ +
69;
UAK; RUCK; 52; Ø;
/ +
69;
UAK; VERO; 42; Ø;
/ display l
UAK; RUCK; 52; Ø;
64;
/ display H
UAK; VERU; 1Ø2; Ø;
68; 66; 71;
/ display - CL
UAK; STOP;
/ end of program
3.3. Exercise 3; a character set for a table of prime numbers

There exists a table of all prime numbers below $10^7$ (see LEHMER [*]). One could produce a table of primes up to $10^9$ in a volume of the same size, by making use of a special character set.

The idea for this dense notation for the distribution of prime numbers originates with A. VAN WIJNGAARDEN.

A sequence of 5 consecutive numbers beginning with a 5-fold contains at most 2 primes (except for the sequence $\emptyset$, 1, 2, 3, 4). Moreover, the only candidates are the 2nd and 4th number or the 3rd and 5th number, depending on whether or not the first number is even. Therefore, it takes two bits to characterise a prime distribution in such a quintuple.

Let: $\emptyset \emptyset$, denoted by -, indicate no primes;

1 $\emptyset$, denoted by \ , indicate the left candidate is prime;

$\emptyset$ 1, denoted by / , indicate the right candidate is prime;

1 1, denoted by | , indicate both candidates are prime.

Next, consider all characters that one obtains by taking all combinations of 4 elements from the set { -, \ , / , | }. This produces 256 characters. Each character can be interpreted in the obvious way as a notation for a prime distribution in a sequence of 20 consecutive numbers. By leaving out all 3-folds, one obtains 165 valid characters.

For a correct interpretation it is still necessary to know whether the first number in the sequence is even or not. It is possible to construct the table in such a way that each character represents a sequence that starts with a 20-fold (which implies an even five fold!).

We can for instance form blocks of 5 characters for each hundred.

\[
\begin{align*}
\text{e.g.:} & \quad \backslash - - / \\
& \quad \backslash - - \\
& \quad \backslash - \backslash \\
& \quad - / \backslash \\
& \quad - \backslash \\
\end{align*}
\]

fig. 3.3.1.

The size of these blocks can be reduced to a square of 2 by 2 mm. including spaces, and we would still have good readability. On a page in A4 format we can write down $10^4$ of these blocks, that is the distribution in a sequence of $10^6$ numbers. Hence for the primes up to $10^9$ we would need a volume of 1000 pages.

We will now as an illustration compare two ways of building up a character set for this book.

**First solution:** Observe that each character is build up from elements of the set:

-, \, / and |.

We will define these four shapes as Digiset characters. The 165 valid characters for the 20-folds will be defined as micropograms. Each microprogram will contain four character calls to one of the four primitive shapes. Since we intend to group five characters in a column (see fig. 3.2.1.), we can include in the microprogram the movement of the recording beam down to the beginning of the next row. For each block we need 5 microprogram calls followed by a move to the beginning of the next block.

The grouping of the blocks can be done in the same way for both solutions, and will not be discussed here.

The program fragment listed below contains the character definition of the four basic shapes, and two of the 165 microprograms.

```
UAK; START;          / begin of program
UAK; SEG; Ø;         / core address
UAK; SPE; 62;         / store typeface constant
Ø; Ø; Ø;             / 4-pt grid, 5Ø * 12Ø lines (type B)
UAT; UAT;
UAK; SPE; 1;          / #1 = basic shape: -
1; 1; 1;              / word #1 in segment #1
(1)(8,7,5)(1);        / 1Ø image lines
UAT; UAT;
UAK; SPE; 2;          / #2 = basic shape \n```
2∅; 1; 1;
(1)(1,14,4)(1,12,6)
(1,1∅,6)(1,8,6)(1,6,6)
(1,4,6)(1,2,6)(1,2,4)
(1);
UAT; UAT;
UAK; SPE; 3;
4∅; 1; 1;
(1)(1,2,4)(1,2,6)(1,4,6)
(1,6,6)(1,8,6)(1,1∅,6)
(1,12,6)(1,14,4)(1);
UAT; UAT;
UAK; SPE; 4;
6∅; 1; 1;
(4)(2,2,16)(4);
UAT; UAT;

/ #3 = basic shape /

UAK; SPE; 6;
8∅; 1; 2;
[
1; 1; 1; 1;
UAK; RUCK; 16∅; ∅;
UAK; VERU; 43; ∅;
];
UAT; UAT;
.
.
.
UAK; SPE; pa;
wa; sa; 2;
[
3; 3; 4; 2;
UAK; RUCK; 16∅; ∅;

/ by way of example we only give
/ two of the 165 microprograms.

/ microprogram for:
/ ---
/ (i.e. no primes).
/ display: - (4*)

/ move to the beginning of
/ the next row
/ end of microprogram

/ other definitions

/ microprogram for:
/ / | \.
UAK; VERU; 43; \( \emptyset \);
];
UAT; UAT;

/ 20 bytes

It follows that for the coding of the alphabet about 3500 bytes are needed. The actual program takes 6 bytes for each hundred numbers, that is 60,000 bytes / page (1 page covers \( 10^6 \) numbers). One page would take about 1 minute Digiset time.

**Second solution:** The set of 165 characters is directly coded. This takes an average of 40 bytes a character, or ca. 7000 bytes for the alphabet. By displaying the characters "upright",

\[
\begin{array}{c}
\text{e.g.} \\
| \\
\backslash \\
/ \\
\end{array}
\]

each block of 100 also takes 6 calls.

Except for the alphabet, both solutions require an equal amount of coding space. However, the second solution may be faster, due to the fact, that a character contains less image lines and displaying of a character requires only one microprogram call. This gain in speed blances the much simpler coding of the alphabet of the first solution.

As an example we will now give the coding for two of the characters.

UAK; SPE; 1;

/ character for: | |
| |
| |

wa; sa; 1;

/ address + type

(4)
(2,3,2\( \emptyset \),5,2\( \emptyset \),5,2\( \emptyset \),5,20)
(4);
UAT; UAT;
UAK; SPE; n; / character for: /

wa; sa; 1;
(1)
(1,21,2,12,5, 36,2
(1,18,5,12,5, 36,5)
(1,15,5,15,5, 39,5)
(1,12,5,18,5,12,20, 10,5)
(1,9,5,21,5,12,20 13,5)
(1,6,5,24,5, 48,5)
(1,3,5,27,5, 51,5)
(1,3,2,30,5, 54,2)
(1);
UAT; UAT;

This exercise has been run on the plotter (see fig. 3.3.2.), and on the Digiset itself (see fig. 3.3.3.). The plotter result is only partly reproduced.
fig. 3.3.2. A character set for a table of prime numbers, plotter simulation of a fragment.
fig. 3.3.3. A character set for a table of prime numbers.
3.4. **Exercise 4; an alphabet for line drawings**

This exercise proves that the Digiset can be used as a machine that generates line drawings.

First the set up of the alphabet and the way in which a vector is drawn is outlined. Next an ALGOL 60 program is given, that actually generates a sub alphabet in hand code.

The second part of the exercise shows some line drawings produced by ALGOL 60 programs that use character calls as primitive actions.

3.4.1. The alphabet

Each Digiset character must be defined within a square, the so-called reference square or unit square. The description of the character in the unit square is then transformed into a description on an existing square raster (the recording grid). The size of the recording grid and the distance of the grid points in x- and y direction can be chosen from a fixed list of alternatives. The reference point of the raster, that is the grid point that will coincide with the actual position of the recording beam when the character is displayed, also must be specified in the definition.

A character consists of a piece of straight line, that enters and leaves the unit square entirely through the vertical or entirely through the horizontal sides (see fig. 3.4.1.).

![Character Types](image)

**fig. 3.4.1. character types.**

Each line piece starts in the lower left- or upper left corner. Hence there are four types of characters, namely two for each quadrant. Each type has a different reference point, at the position marked R. In case a line is cut at the corner diagonal to R, the square is a little extended.
in one direction, such that a straight cut off can be made, e.g.: 

The straight cuts enables the connection of two characters of the same type without gaps or overlaps. Both would be visible and give poor quality.

We propose to draw an arbitrary straight line by approximating it with the two characters with the closest direction. The method of approximating is called threading. Due to some overlap in the directions, covered by the different types, each threading can be obtained of characters from one type only. (So we have no gaps there). Moreover, the threading is calculated, that gives the best approximation.

In order to make straight lines of arbitrary length, we must have at our disposal a sequence of character sets which contain line pieces of decreasing length.

The alphabet chosen contains the following character sets:
- The area between $-\pi/2$ and $+\pi/2$ is divided into $6\theta$ directions. There exist $6\theta$ line pieces, one for each direction, so that the mutual difference is less than or equal to $3^\circ$. The line pieces are defined as 8 point characters with a line width of .8 point ($=\theta = 0.03$ mm). This means that the overlap of two adjacent line pieces is about .5 line width:

Each type contains 16 characters: 15 to cover all directions and one type overlapping character.
- A set of 34 characters consisting of:
  30 4-point characters, each one consisting of a line piece differing from its neighbour by $6^\circ$, but with the same line width of $0.03$ mm;
4 type-overlapping characters.

- A set containing:
  15 2-point characters, differing 12° and 4 overlapping characters.
- A set with 4 1-point characters and 2 overlapping characters.
- 1 .5-point character (the square point).

The complete character set thus contains 124 characters. For each character a microprogram is added, that contains the character call followed by a positioning of the recording beam at the end of the line piece. Now one can draw a line simply by calling the appropriate microprograms one after another.

The 124 characters together with the 124 microprograms can be contained in one typefont. In this way a typefont can be defined for each line width desired.

In appendix C an ALGOL 60 program is listed that generates the 64 8-point line pieces mentioned above, together with the microprograms. This part of the typefont suffices for our testing purposes. The code for a line piece is shown for a few characters only (the complete set takes 18 pages).
fig. 3.4.2. A character set for line drawings (line alphabet).
fig. 3.4.3. The line alphabet microprograms in "alphabetical" order.
In figure 3.4.2 the result of the printout of the alphabet on the Digiset is given in the following ways:
- on the first line the characterset is printed twice in real size (GROa)
- on the second line the characterset is printed once with a space after each character
- on the third line the characterset is enlarged by a factor 2
- on the 4th and 5th line the set is enlarged by a factor 4.

In figure 3.4.3. the characterset is printed twice by calling the microprogram for each character. This is repeated once on the second line with enlargement factor 2.

3.4.2. The line drawings

Each "sundown"-like drawing of figure 3.4.4. is the result of the program listed below, executed once for each character in the alphabet. The characters are called through their microprograms. In this way, one execution of 6 instructions produces a line consisting of 10 equal line pieces.

```
UAK; WID; 9; ø;
mp;
org;
```

/repeat microprogram * 10 times
/primary address of microprogram
/move back to the starting point

The microprogram "org" stands for the sequence:

```
UAK; ZEV; ø; ø;
UAK; VERA;
```

/move to ø x position
/move to ø y position

Each one of the five subpictures shows the directions available within the alphabet. The black kernel at the origin has a radius of about two line pieces. Moreover, the shape of the kernel is exactly equal for all subpictures. Hence the shape is determined by the overlap only, and is not visibly influenced by inaccuracies.
fig. 3.4.4. Repeated microprograms
fig. 3.4.5. Diamonds.
The "diamond" shapes of figure 3.4.5. are obtained by combining two different characters of the alphabet as follows:

The length of the sides is five line pieces, hence nowhere inside the diamond, the connection between two pieces is covered by a crossing line.

None of the diamonds shows any irregularity caused by an improper connection or improper positioning of its constituting pieces. The precision obtained, indeed equals the theoretical precision of the addressable points.

The ALGOL 60 program that generated this figure is also listed in appendix C. It is an example of the programming effort necessary for drawing on the Digiset. One should remember that the characterset needs to be generated only once. Hence we can say that the program listed is really all that is needed for this line drawing.

The short hand code program for one diamond is listed below. The x- and y direction are π/4 and π/2 respectively.

```
UAK; TAB; a1; a2;  / define new origin
org;              / move to origin
UAK; WID; 4; Ø; X;  / draw "X" 5 times (first line)
org;
UAK; VERU; 32; 1;   / move to start of 2nd line
UAK; WID; 4; Ø; X;  / second line
org;
UAK; VERU; 64; 2;    / start of 3rd line
UAK; WID; 4; Ø; X;  / third line
org;
UAK; VERU; 96; 3;
UAK; WID; 4; Ø; X;  / fourth line
org;
UAK; VERU; 128; 4;
UAK; WID; 4; Ø; X;  / fifth line
```
org;
UAK; WID; 4; Ø; Y;       / draw "Y" 5 times (first line)
org;
UAK; ZWQ; 194; 1; ZWR;   / change x position
UAK; VERU; 32; 1;        / change y position
UAK; WID; 4; Ø; Y;       / second line
org;
   .
   .
   .
UAK; WID; 4; Ø; Y;       / fifth line
org;                     / 116 bytes
APPENDIX A

The Digiset simulator in ALGOL 60.
'BEGIN'
  'INTEGER' CORESEGMENTS, DISCSECTORS, MAXLINENUMBERCHAR,
  DTAPELENGTH, LINENUMBERMAX, A;
  'BOOLEAN' PLOTTER, BOOLERROR, PUNCH;

  'PROCEDURE' PARAMETER ERROR (STRING); 'STRING' STRING;
  'BEGIN' PRINTTEXT ("PARAMETER ERROR"); SPACE (5);
    PRINTTEXT (STRING); NLCR; BOOLERROR := 'TRUE'
  'END' PARAMETER ERROR;

  BOOLERROR := 'FALSE'; CORESEGMENTS := READ;
  'IF' CORESEGMENTS < 1 OR CORESEGMENTS > 64
    'THEN' PARAMETER ERROR
      
  "NUMBER OF SEGMENTS OUTSIDE ADMISSIBLE RANGE"
  DISCSECTORS := READ;
  'IF' DISCSECTORS < 1 OR DISCSECTORS > 3274
    'THEN' PARAMETER ERROR
      "NUMBER OF DISCSECTORS OUTSIDE RANGE"
  MAXLINENUMBERCHAR := READ;
  'IF' MAXLINENUMBERCHAR > 15000
    'THEN' PARAMETER ERROR
      "MAX NUMBER OF IMAGE LINES OUTSIDE ADMISSIBLE RANGE"
  DTAPELENGTH := READ; LINENUMBERMAX := READ;
  'IF' LINENUMBERMAX < 1 OR LINENUMBERMAX > 15000
    'THEN' PARAMETER ERROR
      "SIMULATED RECORDING AREA NOT CONTAINED IN REAL
      RECORDING AREA"
  PLOTTER := LINENUMBERMAX = 1;
  A := RESYM; PUNCH := A = 91;
  'IF' (PUNCH = A = 91)
    'THEN' PARAMETER ERROR
      "NON ALLOWED PUNCH OPTION"
  'ELSE' 'IF' PUNCH 'THEN'
    'BEGIN'
      'IF' (RESYM = 91 AND RESYM = 91)
        'THEN' PARAMETER ERROR
          "WRONG NUMBER OF SPECIFICATION PARAMETERS"
      'FOR' A := RESYM 'WHILE' A = 93 'DO'
        'IF' A 'NE' 119
          'THEN' PARAMETER ERROR
            "PARAMETERS NOT ON SEPARATE CARD"
    'END';

  'BEGIN'
    'INTEGER' CD0, CDAN, CROA, CGROA, CGROB, CGROC, CGROD, CGROE, CGROF,
    CGROG, CGROH, CGROJ, CGROK, CGROL, CGROM, CKEN,
    CKUR1, CKUR2, CKUR3, CNOR, CNUL, CPLA, CPLE, CRUCK, CSCHN,
    CSEG, CSPE, CSTART, CTAB, CTR, CUA, CUAT, CUBL, CVERA,
    CVERO, CVERU, CWID, CZER, CZEV, CZWQ, CZWR, CSTOP,
    QW QUANTIFICATION, TAB QUANTIFICATION, KURN, SECTADDR,
    PA, SEG, SEGFM, SECSIG, SECPA, THIRDBYTE, BYTE, SHADING,
    REMAINDER, GRIDTYPE, PRINTPOSITION, LINENUMBER, SECTBYTE,
    INSTR NUMBER, INSTR NUMBER, LINCOUNT,
    BYTENUMBER INPUT, A, B, I, J, N, R, S, T;
    'REAL' CELLSIZE, CELLMHEIGHT, WIDTH, HEIGHT, FACTOR;
    'BOOLEAN' MIC, BUFFERSEG, BOOL RDISC, BOOL WID, CHAR,
    BOOL INPCORE, BOOL CHAR TO DISC, BOOL ZWO;

    'INTEGER' 'ARRAY' DTAE [1 : DTAPELENGTH],
CORE [0 : CORESEGMENTS, 0 : 255, 1 : 3],
DISC [1 : DISCSECTORS, 1 : 744],
'REAL' 'ARRAY' ENLARGEMENT FACTOR [0 : 12],
'BOOLEAN' 'ARRAY' LINEPRINTER OUTPUT
[1 : LINESHIFT, 1 : 144],
'INTEGER' 'PROCEDURE' READ DTape;
'BEGIN' READ DTape := DTape [BYTE NUMBER INPUT];
BYTE NUMBER INPUT := BYTE NUMBER INPUT + 1;
'IF' != BOOL INCPRIOR => BOOL CHAR TO DISC;
'THEN' INSTR NUMBER := INSTR NUMBER + 1;
'END' READ DTape;

'INTEGER' 'PROCEDURE' READ DISC;
'BEGIN'
  'IFDEF' SECTADDR > DISCSECTORS
  'THEN' ERROR
    ('"INSTRUCTION CAUSES ADDRESSING OF NON-EXISTENT
     DISC-SECTOR", MIC, 'FALSE', 'TRUE')
  READ DISC := DISC [SECTADDR, SECTBYTE];
  'IFDEF' SECTBYTE < 744 'THEN' SECTBYTE := SECTBYTE + 1 'ELSE'
  'BEGIN' SECTADDR := SECTADDR + 1; SECTBYTE := 1 'END';
'END' READ DISC;

'INTEGER' 'PROCEDURE' READ CORE;
'BEGIN'
  'IFDEF' SECSSEG > CORESEGMENTS
  'THEN' ERROR
    ('"NON-EXISTENT CORE=SEGMENT", MIC, 'FALSE', 'TRUE')
  'IFDEF' MIC = CHAR
    'THEN' INSTR NUMBER MIC := INSTR NUMBER MIC + 1;
    READ CORE := CORE [SECSSEG, SECPA, BYTE];
  'IFDEF' BYTE < 3 'THEN' BYTE := BYTE + 1 'ELSE'
    'IFDEF' SECPA < 255 'THEN'
      'BEGIN' SECPA := SECPA + 1; BYTE := 1 'END';
    'ELSE'
      'BEGIN' SECSSEG := SECSSEG + 1; SECPA := 0; BYTE := 1 'END';
  'END' READ CORE;

'INTEGER' 'PROCEDURE' RDTOD (BOOL RDDISC);
'BOOLEAN' BOOL RDDISC;
'IFDEF' BOOL RDDISC 'THEN' RDTOD := READ DISC
'ELSE' RDTOD := READ DTAPE;

'INTEGER' 'PROCEDURE' RDCOC (MIC); 'BOOLEAN' MIC
'BEGIN'
  'IFDEF' MIC 'THEN' RDCOC := READ CORE
  'ELSE' RDCOC := READ DTAPE
'END' RDCOC;

'INTEGER' 'PROCEDURE' RDCODOC (BOOL RDDISC, MIC);
'BOOLEAN' MIC, BOOL RDDISC
RDCODOC :=
'IFDEF' BOOL RDDISC 'THEN' READ DISC 'ELSE' RDCOC (MIC);

'PROCEDURE' ERROR (STRING, MIC, CHAR, STOP); 'STRING' STRING;
'BOOLEAN' MIC, CHAR, STOP;
'BEGIN' 'INTEGER' N, M;
NLCR; PRINTTEXT ("STRING") NLCR;
PRINTTEXT ("NUMBER_INPUT-ELEMENT") SPACE (33));
PRINT (INSTR NUMBER = 1); NLCR;
'IF' MIC 'THEN'
'BEGIN' PRINTTEXT ("INSTRUCTION-NUMBER MICROPROGRAM")
SPACE (22); PRINT (INSTR NUMBER MIC); NLCR;
'END'
'IF' CHAR 'THEN'
'BEGIN' PRINTTEXT ("LINENUMBER CHARACTER") SPACE (33);
PRINT (LINECOUNT); NLCR;
'END'
'IF' STOP 'THEN'
'BEGIN' NEW PAGE;
'IF' = PLOTTED 'THEN'
'BEGIN'
'FOR' N = 1 'STEP' 1 'UNTIL' LINENUMBERMAX 'DO'
'FOR' M = 1 'STEP' 1 'UNTIL' 144 'DO'
'IF' LINENEPRINTEROUTPUT [N, M] 'THEN' PRSYM (133)
'ELSE' SPACE (1)
'END';
EXIT
'END'
'ELSE' BooleanError = 'TRUE'
'END' ERROR;
'PROCEDURE' PRINTRELATIVE
(INIT LINENUMBER, INIT PRINTPOS, DELTAX, DELTAY,
LINENUMBERMAX, PRINT);
'INTEGER' DELTAX, DELTAY, LINENUMBERMAX;
'BOOLEAN' PRINT, INIT LINENUMBER, INIT PRINTPOS;
'COMMENT' THE GLOBAL PARAMETERS LINENUMBER AND PRINTPOSITION
ARE CHANGED;
'BEGIN'
'IF' INIT LINENUMBER 'THEN' LINENUMBER = DELTAX
'ELSE' LINENUMBER = LINENUMBER + DELTAX;
'IF' INIT PRINTPOS 'THEN' PRINTPOSITION = DELTAY
'ELSE' PRINTPOSITION = PRINTPOSITION + DELTAY;
'IF' PRINTPOSITION < 1 'OR' PRINTPOSITION > 144
'THEN' ERROR ("Y OUTSIDE PRINT-RANGE", MIC, CHAR, 'TRUE');
'IF' LINENUMBER < 1 'OR' LINENUMBER > LINENUMBERMAX
'THEN' ERROR ("X OUTSIDE PRINT-RANGE", MIC, CHAR, 'TRUE');
'IF' PRINT
'THEN' LINENEPRINTEROUTPUT [LINENUMBER, PRINTPOSITION] =
'TRUE'
'END' PRINTRELATIVE;
'PROCEDURE' PLOTRELATIVE
(INITIAL VALUE, DELTAX, DELTAY, IPEN, KURN);
'INTEGER' DELTAX, DELTAY, IPEN, KURN; 'BOOLEAN' INITIAL VALUE;
'BEGIN' 'OWN' 'INTEGER' X, Y; 'REAL' TG;
'IF' INITIAL VALUE 'THEN' 'BEGIN' X = 0; Y = 1375 'END'
'ELSE'
'IF' KURN = 0 'AND' DELTAY = 0 'THEN'
'BEGIN' X = X + DELTAX; Y = Y + DELTAY 'END'
'ELSE'
'BEGIN'
'IF' KURN = 1 'THEN' TG = 0.5205671 'ELSE'
'IF' KURN = 2 'THEN' TG = 0.4663077
'ELSE' TG = 0.4142136
Y := Y + DELTAY; X := X + TG * DELTAY;

'END';

'IF' = (0 'LE' X 'X' 'LE' 15000)

'THEN' ERROR (**X OUTSIDE PLOT RANGE**, MI, CHAR, 'TRUE');

'IF' = (0 'LE' Y 'Y' 'LE' 2750)

'THEN' ERROR (**Y OUTSIDE PLOT RANGE**, MI, CHAR, 'TRUE');

PLOT (X, Y, IPEN)

'END' PLOTR E LATIVE;

'PROCEDURE' INTERPRET (S, MI, BOOL RDDISC, PLO TTER);

'INTEGER' S, BOOL RDDISC, PLO TTER;

'BEGIN' 'INTEGER' R, SEC PAMIC, SECSEMIC, BYTEMIC, N, M;

'IF' S = CUAK 'THEN'

'BEGIN' R := RDTODOC (BOOL RDDISC, MI);

'IF' R = CSPE 'THEN' SPE (MIC) 'ELSE'

'IF' R = CSTOP 'THEN' STOP (PLOTTER) 'ELSE'

'IF' R = CSEG 'THEN' SEG (RDTOC (MIC), SEG, SEGMF)

'ELSE'

'IF' R = CDAN 'THEN'

(DRTOC (MIC), REMAINDER, SHADING, FACTOR, CELLWIDTH, WIDTH, CELLMHEIGHT, HEIGHT, PLOTTER)

'ELSE'

'IF' R = CGROA 'THEN' GRO

(0, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLMHEIGHT, HEIGHT, PLOTTER)

'ELSE'

'IF' R = CGROB 'THEN' GRO

(1, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLMHEIGHT, HEIGHT, PLOTTER)

'ELSE'

'IF' R = CGROC 'THEN' GRO

(2, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLMHEIGHT, HEIGHT, PLOTTER)

'ELSE'

'IF' R = CGROD 'THEN' GRO

(3, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLMHEIGHT, HEIGHT, PLOTTER)

'ELSE'

'IF' R = CGROE 'THEN' GRO

(4, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLMHEIGHT, HEIGHT, PLOTTER)

'ELSE'

'IF' R = CGROF 'THEN' GRO

(5, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLMHEIGHT, HEIGHT, PLOTTER)

'ELSE'

'IF' R = CGROG 'THEN' GRO

(6, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLMHEIGHT, HEIGHT, PLOTTER)

'ELSE'

'IF' R = CGROH 'THEN' GRO

'IF' R = CGROI

'IF' R = CGROJ

'IF' R = CGROK

'IF' R = CGROL

'IF' R = CGROM
(7, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLHEIGHT, HEIGHT, PLOTTER)

ELSE
  IF R = CGRO1
    THEN GRO
(8, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLHEIGHT, HEIGHT, PLOTTER)

ELSE
  IF R = CGROJ
    THEN GRO
(9, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLHEIGHT, HEIGHT, PLOTTER)

ELSE
  IF R = CGROK
    THEN GRO
(10, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLHEIGHT, HEIGHT, PLOTTER)

ELSE
  IF R = CGROL
    THEN GRO
(11, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLHEIGHT, Height, PLOTTER)

ELSE
  IF R = CGROM
    THEN GRO
(12, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH, WIDTH, CELLHEIGHT, Height, PLOTTER)

ELSE
  IF R = CNOR 'THEN' KURNOR (0, KURN, PLOTTER) 'ELSE'
  IF R = CKUR1 'THEN' KURNOR (1, KURN, PLOTTER) 'ELSE'
  IF R = CKUR2 'THEN' KURNOR (2, KURN, PLOTTER) 'ELSE'
  IF R = CKUR3 'THEN' KURNOR (3, KURN, PLOTTER) 'ELSE'
  IF R = CUBL 'THEN'
BEGIN
  IF MIC 'THEN' ERROR ('"UBL IN MIC", "TRUE", "FALSE", "TRUE")
  ELSE UBL
END
ELSE
  IF R = CRUCK
    THEN RUCK (RDTOC (MIC), RDTOC (MIC), PLOTTER)
ELSE
  IF R = CVERO
    THEN VERO (RDTOC (MIC), RDTOC (MIC), PLOTTER)
ELSE
  IF R = CVERU
    THEN VERU (RDTOC (MIC), RDTOC (MIC), PLOTTER)
ELSE
  IF R = CVERA 'THEN' VERA (PLOTTER) 'ELSE'
  IF R = CZWO
    THEN ZWO
    (RDTOC (MIC), RDTOC (MIC), ZWO QUANTIFICATION,
    BOOZWO)
ELSE
  IF R = CTAB
    THEN TAB
(RDTOC (MIC), RDTOC (MIC), TAB QUANTIFICATION, MIC)

'ELSE'

'IF' R = CZER
'THEN'
ZER
(RDTOC (MIC), RDTOC (MIC), TAB QUANTIFICATION, PLOTTER)

'ELSE'

'IF' R = CZEV
'THEN'
ZEV
(RDTOC (MIC), RDTOC (MIC), TAB QUANTIFICATION, PLOTTER)

'ELSE'

'IF' R = CWID
'THEN'
WID (RDTOC (MIC), RDTOC (MIC), MIC, BOOL WID)

'ELSE'

'IF' R = CPLA ∨ R = CPLE ∨ R = CTR E
'THEN'
ERROR
("PLA,PLE OR TRE NOT PRECEDED BY UAG,Spej", MIC, 'FALSE', 'TRUE')

'ELSE'
ERROR
("INSTRUCTION CANNOT BE PRECEDED BY UAG", MIC, 'FALSE', 'TRUE')

'END'

'ELSE'

'IF' S = CZWR
'THEN'
ZWR (PLOTTER, BOOLZWO, ZWO QUANTIFICATION)

'ELSE'

'IF' S = CKEN 'THEN'

'BEGIN'

'IF' MIC
'THEN'
ERROR ('KEN IN MIC', 'TRUE', 'FALSE', 'TRUE')

'ELSE'
KEN

'END'

'ELSE'

'IF' S = CNUL
'THEN'
ERROR
("NUL-INSTRUCTION NOT SIMULATED", MIC, 'FALSE', 'TRUE')

'ELSE'

'IF' S = CUAT
'THEN'
ERROR
("UAT-INSTRUCTION NOT ALLOWED IN THIS PLACE", MIC, 'FALSE', 'TRUE')

'ELSE'

'BEGIN'

'IF' MIC 'THEN'

'BEGIN' SECPAMIC := SECPA; SECSAMIC := SECSEG)
BYTEMIC := BYTE)
CALL TYPEFOUNT ELEMENT
(S, PLOTTER, CELLSIZE, CELLDWIDTH, SHADING,
REMAINDER, GRIDTYPE, KURN);)
SECSAMIC := SECSEG; SECPAMIC := SECPAMIC;
BYTE := BYTEMIC)

'END'

'ELSE'
CALL TYPEFOUNT ELEMENT
(S, PLOTTER, CELLSIZE, CELLDWIDTH, SHADING,
REMAINDER, GRIDTYPE, KURN)

'END'

'END' [INTERPRET]
'PROCEDURE: STOP (PLOTTER); 'BOOLEAN: PLOTTIER;
'BEGIN: INTEGER N, M;
'IF':=PLOTTER 'THEN':
'BEGIN:'
'FOR' N = 1 'STEP' 1 'UNTIL' LINESNUMBERMAX 'DO'
'FOR' M = 1 'STEP' 1 'UNTIL' 144 'DO'
'IF' LINEPRINTEROUTPUT [N, M] 'THEN' PRSYM (133)
'ELSE' SPACE (1);
'END';
'END' STOP;

'PROCEDURE: SEG (BYTE, SEG, SEGMF);
'INTEGER: BYTE, SEG, SEGMF;
'BEGIN: SEGMF := SEG; SEGMI := BYTE;
'IF' SEG > CORESEGMENTS 'THEN' ERROR
("NON-EXISTENT CORE-SEGMENT", MIC, 'FALSE', 'TRUE');
'END' SEG;

'PROCEDURE: DAN
(BYTE, REMAINDER, SHADING, FACTOR, CELLWIDTH, WIDTH,
CELLHEIGHT, HEIGHT, PLOTTER);
'INTEGER: BYTE, REMAINDER, SHADING;
'REAL: FACTOR, CELLWIDTH, WIDTH, CELLHEIGHT, HEIGHT;
'BOOLEAN: PLOTTER;
'BEGIN: FACTORDAN;
'IF':= PLOTTER 'THEN':
'BEGIN:
'IF' MIC
'THEN' INSTRNUMBER MIC := INSTRNUMBER MIC = 1;
'ELSE' INSTRNUMBER MIC = INSTRNUMBER = 1;
ERROR
("DAN INSTRUCTION AND LINEPRINTEROUTPUT", MIC, 'FALSE', 'TRUE');
'END';
'IF' 7 < BYTE & BYTE < 13 'THEN' FACTORDAN := ENLARGEMENT FACTOR [BYTE = 8]
'ELSE'
'IF' 13 < BYTE & BYTE < 25 'THEN' FACTORDAN := ENLARGEMENT FACTOR [BYTE / 2 = 2]
'ELSE'
'ELSE'
'IF' BYTE = 32 'THEN' FACTORDAN := ENLARGEMENT FACTOR [12]
'ELSE' ERROR ('QUANTIFICATION DAN', MIC, 'FALSE', 'TRUE');
'FACTOR := 1; CELLWIDTH := WIDTH * FACTORDAN;
CELLHEIGHT := HEIGHT; SHADING := ENTER (CELLWIDTH / 4);
REMAINDER := CELLWIDTH = 4 * SHADING;
'IF' REMAINDER = 4 'THEN'
'BEGIN: SHADING := SHADING + 1; REMAINDER := 0 'END';
'END' DAN;

'PROCEDURE: GRO
(N, REMAINDER, SHADING, GRIDTYPE, KURN, FACTOR, CELLWIDTH,
WIDTH, CELLHEIGHT, HEIGHT, PLOTTER);
'INTEGER: N, REMAINDER, SHADING, GRIDTYPE, KURN;
'REAL' FACTOR, CELLPWT:, WIDTH, CELPHGHT, HEIGHT;

'BOOLEAN' PLOTTER;

'BEGIN' 'INTEGER' R

'IF' SEGM = 1

'THEN' ERROR

("GRO NOT PRECEDED BY SEG", MIC, 'FALSE', 'TRUE');

R := CORE [SEG, 62, 1];

'IF' (0 'LE' R & R 'LE' 19) V (32 'LE' R & R 'LE' 51) V

(64 'LE' R & R 'LE' 83))

'THEN' ERROR

("TYPEFACE-CONST. HAS NON-ALLOWED VALUE", MIC, 'FALSE', 'TRUE');

'IF' PLOTTER 'THEN'

'BEGIN' KURN := R = 4 * ENTER (R / 4)

'IF' 0 'LE' R & R 'LE' 19 'THEN'

'BEGIN' WIDTH := 12; HEIGHT := 5 'END'

'ELSE'

'IF' 32 'LE' R & R 'LE' 51 'THEN'

'BEGIN' WIDTH := 12; HEIGHT := 10; R := R = 32 'END'

'ELSE'

'IF' 64 'LE' R & R 'LE' 83 'THEN'

'BEGIN' WIDTH := 24; HEIGHT := 10; R := R = 64 'END'

'ELSE' ERROR

("TYPEFACECONSTANT", MIC, 'FALSE', 'TRUE');

'FACTORS ENLARGEMENT FACTOR [N]

CELLWIDTH := WIDTH * FACTOR;

CELLHEIGHT := HEIGHT * FACTOR;

SHADING := ENTER (CELLWIDTH / 4));

REMAINDER := CELLWIDTH = 4 * SHADING;

'IF' REMAINDER = 4 'THEN'

'BEGIN' SHADING := SHADING + 1; REMAINDER := 0 'END'

'END';

'ELSE'

'BEGIN'

'IF' = N = 0

'THEN' ERROR

("NOT GROA AND LINEPRINTEROUTPUT", MIC, 'FALSE', 'TRUE');

'ELSE' KURN := 0;

'IF' 32 'LE' R & R 'LE' 51 'THEN' R := R = 32

'ELSE' ERROR

("B-OR D-GRID AND LINEPRINTEROUTPUT", MIC, 'FALSE', 'TRUE');

'END';

GRIDTYPE := 2 * ENTER (R / 4);

'END' GRO;

'PROCEDURE' KURNOR (N, KURN, PLOTTER); 'INTEGER' N, KURN;

'BOOLEAN' PLOTTER;

'BEGIN'

'IF' PLOTTER 'THEN' KURN := N

'ELSE' ERROR

("KUR/NOR-INSTRTION AND LINEPRINTEROUTPUT", MIC, 'FALSE', 'TRUE');

'END' KURNOR;

'PROCEDURE' KEN;

'COMMENT' THE KEN-INSTRTION IS ONLY SIMULATED IN

TEXT-DATA, NOT IN CHARACTOR-OR MICROPROGRAM-DATA)

'BEGIN' 'INTEGER' R, S, T
R1: READ DTAPE; S1: READ DTAPE; T1: READ DTAPE;
477  'IF' ((R = CUAK) ∧ (S = CSPE)) ∧ ((S = CUAK) ∧ (T = CSPE)) ∨ (R = CKEN ∧ S = CKEN ∧ T = CKEN)
478  'THEN' ERROR
479  "(CUAK SPE OR KEN KEN KEN IN KEN", 'FALSE', 'FALSE',
480  'TRUE')
481
482  'END' KEN;
483
484  'PROCEDURE' UBL;
485  'COMMENT' THE UBL-INSTRUCTION IS ONLY SIMULATED IN
486  TEXT-DATA, NOT IN CHARACTER-OR MICROPROGRAM-DATA;
487  'BEGIN' 'INTEGER' Y, Z;
488  Y1: READ DTAPE;
489  'FOR' Z1: READ DTAPE 'WHILE' - ((Y = CUAK) ∧ (Z = CSTART))
490  'DO' Y1: Z
491  'END' UBL;
492
493  'PROCEDURE' WID (BYTE1, BYTE2, MIC, BOOLWID); 'INTEGER' BYTE1, BYTE2; 'BOOLEAN' BOOLWID, MIC;
495  'BEGIN' 'INTEGER' N, M, PA;
496  'IF' BOOLWID
497  'THEN' ERROR
498  "REPEATED MICROPROGRAM CONTAINS WID-INSTRUCTION", 'TRUE', 'FALSE', 'TRUE');
499  BOOLWID:= 'TRUE'; N:= BYTE1 + 256 * BYTE2;
500  PA:= RDOTC (MIC);
501  'IF' PA < 1 ~ PA = 28 ~ PA = 62 ~ PA = 281 ~ PA > 254
502  'THEN' ERROR
503  "INSTRUCTION NOT ALLOWED AS WORD-ADDRESS", MIC,
504  'FALSE', 'TRUE');
505  'FOR' M:= 0 'STEP' 1 'UNTIL' N
506  'DO' CALL TYPEFOUNT ELEMENT
507  (PA, PLOTTER, CELLEIGHT, CELLEWIDTH, SHADING,
508  REMAINDER, GRIDTYPE, KURN);
509  BOOLWID:= 'FALSE'
510  'END' WID;
511
512  'PROCEDURE' ZWQ (BYTE1, BYTE2, ZWQ QUANTIFICATION, BOOLZWQ); 'INTEGER' BYTE1, BYTE2, ZWQ QUANTIFICATION; 'BOOLEAN' BOOLZWQ;
513  'BEGIN' 'INTEGER' N;
514  'IF' ZWQ QUANTIFICATION:= 3 * RDOTC (MIC) + 768 * RDOTC (MIC);
515  BOOLZWQ:= 'TRUE';
516  'END' ZWQ;
517
518  'PROCEDURE' ZWR (PLOTTER, BOOLZWQ, ZWQ QUANTIFICATION); 'BOOLEAN' PLOTTER, BOOLZWQ; 'INTEGER' ZWQ QUANTIFICATION;
519  'BEGIN' 'INTEGER' R;
520  'IF' = BOOLZWQ
521  'THEN' ERROR
522  "ZWQ-INSTRUCTION BEFORE FIRST ZWQ-INSTRUCTION", MIC, 'FALSE', 'TRUE');
523  'IF' PLOTTER
524  'THEN' PLOTRelative
525  ('FALSE', ZWQ QUANTIFICATION, 0, - 2, 0)
526  'ELSE' PRINTRelative
527  ('FALSE', 'FALSE', ZWQ QUANTIFICATION / 12, 0,
528  LENUMBERMAX, 'FALSE')
529
530  'END' ZWR;
531
532  'PROCEDURE' RUCK (BYTE1, BYTE2, PLOTTER);
'INTEGER' BYTE1, BYTE2; 'BOOLEAN' PLOTTER;
'BEGIN' 'INTEGER' RU;
RU := 3 * BYTE1 + 768 * BYTE2;
'IF' PLOTTER
'THEN' PLOTRELATIVE ('FALSE', RU, 0, -2, 0)
'ELSE' PRINTRELATIVE
('FALSE', 'FALSE', RU / 12, 0, LINENUMBERMAX, 'FALSE')
'END' RUCK;

'PROCEDURE' VERO (BYTE1, BYTE2, PLOTTER);
'INTEGER' BYTE1, BYTE2; 'BOOLEAN' PLOTTER;
'BEGIN' 'REAL' VE;
VE := 4.6875 * BYTE1 + 1200 * BYTE2;
'IF' PLOTTER 'THEN' PLOTRELATIVE ('FALSE', 0, VE, -2, 0)
'ELSE' PRINTRELATIVE
('FALSE', 'FALSE', 0,_VE / 10, LINENUMBERMAX, 'FALSE')
'END' VERO;

'PROCEDURE' VERU (BYTE1, BYTE2, PLOTTER);
'INTEGER' BYTE1, BYTE2; 'BOOLEAN' PLOTTER;
'BEGIN' 'REAL' VE; 'INTEGER' RU
VE := 4.6875 * BYTE1 + 1200 * BYTE2;
'IF' PLOTTER
'THEN' PLOTRELATIVE ('FALSE', 0, -VE, -2, 0)
'ELSE' PRINTRELATIVE
('FALSE', 'FALSE', 0, -VE / 10, LINENUMBERMAX, 'FALSE')
'END' VERU;

'PROCEDURE' VERA (PLOTTER); 'BOOLEAN' PLOTTER;
'BEGIN'
'IF' PLOTTER
'THEN' PLOTRELATIVE
('FALSE', 0, 1375 - PLOT' (0, 0, 16), -2, 0)
'ELSE' PRINTRELATIVE
('FALSE', 'TRUE', 0, 73, LINENUMBERMAX, 'FALSE')
'END' VERA;

'PROCEDURE' TAB (BYTE1, BYTE2, TAB QUANTIFICATION, M1C);
'INTEGER' BYTE1, BYTE2, TAB QUANTIFICATION; 'BOOLEAN' M1C;
TAB QUANTIFICATION := 3 * RDTOC (M1C) + 768 * RDTOC (M1C)

'PROCEDURE' ZER (BYTE1, BYTE2, TAB QUANTIFICATION, PLOTTER);
'INTEGER' BYTE1, BYTE2, TAB QUANTIFICATION; 'BOOLEAN' PLOTTER;
'BEGIN' 'REAL' ZE;
ZE := 4.6875 * BYTE1 + 1200 * BYTE2;
'IF' PLOTTER 'THEN' PLOTRELATIVE ('FALSE', 0, -ZE, -2, 0)
PLOTRELATIVE ('FALSE', TAB QUANTIFICATION - PLOT (0, 0, 15), 0, -2, 0)
'END'
'ELSE' PRINTRELATIVE
('TRUE', 'FALSE', TAB QUANTIFICATION / 12 + 1, ZE / 10, LINENUMBERMAX, 'FALSE')
'END' ZER;

'PROCEDURE' ZEV (BYTE1, BYTE2, TAB QUANTIFICATION, PLOTTER);
'INTEGER' BYTE1, BYTE2, TAB QUANTIFICATION; 'BOOLEAN' PLOTTER;

'BEGIN' 'REAL' ZE;
ZE := 4.6875 * BYTE1 + 1200 * BYTE2;

'IF' PLOTTER 'THEN'

'BEGIN' PLOTRELATIVE ('FALSE', 0, = ZE, - 2, 0)
PLOTRELATIVE ('FALSE', TAB QUANTIFICATION = PLOT (0, 0, 15), 0,
= 2, 0)

'END'

'ELSE' PRINTRELATIVE ('TRUE', 'FALSE', TAB QUANTIFICATION / 12 + 1,
- ZE / 10, LINENUMBERMAX, 'FALSE')

'END' ZE;

'PROCEDURE' INPDEC (SINGLE CHARACTER);

'BOOLEAN' SINGLE CHARACTER;

'BEGIN' 'INTEGER' SECSEG, SECPA, BYTE, R, S, T, U, V, W, N;

'PROCEDURE' STOREC (X); 'INTEGER' X;

'BEGIN'

'IF' SECSEG 'LE' CORESEGMENTS

'THEN' CORE [SECSEG, SECPA, BYTE] := X;

'ELSE' ERROR

('TYPEF, ELEMENT INTO NON-EXISTENT CORE-SEG,",
M鹤, 'FALSE', 'TRUE')

'IF' BYTE < 3 'THEN' BYTE := BYTE + 1 'ELSE'

'IF' SECPA < 255 'THEN'

'BEGIN' SECPA := SECPA + 1; BYTE := 1 'END'

'ELSE'

'BEGIN' SECSEG := SECSEG + 1; SECPA := 0; BYTE := 1

'END'

'END' STOREC;

SEGM := SEGM; 'IF' SEGM = 0 'THEN' BUFFERSEG := 'FALSE';

'IF' PA = 62 'THEN' SINGLE CHARACTER 'THEN'

'BEGIN' SECSEG := SEGM; SECPA := 62; BYTE := 1;

R := RDTOD (BOOL RDISC); S := RDTOD (BOOL RDISC);

T := RDTOD (BOOL RDISC);

'IF' = BOOL RDISC 'THEN'

'BEGIN'

'IF' T = ((O 'LE' R = R 'LE' 19) +

(32 'LE' R = R 'LE' 51) +

(64 'LE' R = R 'LE' 83))

'THEN' ERROR

('TYPEF, ELEMENT INTO NON-ALLOWED VALUE", 'FALSE', 'FALSE', 'TRUE')

'IF' S 'NE' 0

'THEN' ERROR

('SEC, BYTE TYPEFACECONST, NOT EQUAL ZERO", 'FALSE', 'FALSE', 'TRUE')

'IF' T 'NE' 0

'THEN' ERROR

('THIRD BYTE TYPEFACECONST, NOT EQUAL ZERO", 'FALSE', 'FALSE', 'TRUE')

'END';

STOREC (R); STOREC (S); STOREC (T);

'IF' = ((RDTOD (BOOL RDISC) = CUAT) +

(RDTOD (BOOL RDISC) = CUAT))

'THEN' ERROR

('TYPEFAECONST NOT FOLLOWED BY UAT UAT",

STROKEMODE = STROKETOG, 'TRUE')
'BEGIN' BYTE:= 1;

'IF' SINGLE CHARACTER 'THEN'

'BEGIN' SECPA:= 0; SECSG:= SEGm + 1; 'END'

'ELSE'

'BEGIN'

CORE [SEGm, PA, 1]:= SECPA:= RDTOD (BOOL RDDISC);
CORE [SEGm, PA, 2]:= RDTOD (BOOL RDDISC);
SECSG:= CORE [SEGm, PA, 2] + SEGm;

'IF' SECSG > CORESEGMENTS
THEN' ERROR

(""TYPEF,EL, INTO NON-EXISTENT CORE=SEGm",
M1C, "FALSE", "TRUE");

CORE [SEGm, PA, 3]:= THIRDBYTE:=
RDTOD (BOOL RDDISC);

'IF' ~ BOOL RDDISC 'THEN'

'BEGIN'

'IF' ~ (THIRDBYTE = 0 ~ THIRDBYTE = 1 ~
THIRDBYTE = 2 ~ THIRDBYTE = 32)

'THEN' ERROR

(""THIRDBYTE SEC,ADDRESS=WORD", "FALSE",
"FALSE", "TRUE");

'IF' SECSG = SEGm ~ SECPA < 63 'THEN'

'BEGIN' INSTR NUMBER:= INSTR NUMBER + 2
ERROR

(""IMAGE-AREA BELOW TYPEFACECONSTANT",
"FALSE", "FALSE", "TRUE");

'END'

'END'

'END'}

'END'

'BEGIN'

'IF' THIRDBYTE = 1 ~ THIRDBYTE = 0 ~ THIRDBYTE = 32
THEN'

'BEGIN' R:= RDTOD (BOOL RDDISC); BOOL!NPCORE:= 'TRUE'

'FOR' S:=RDTOD(BOOL RDDISC)

'WHILE' ~ ((R = CUAT) ~ (S = CUAT))

'DO' 'BEGIN' STOREC (R); R:= S 'END'

STOREC (R); BOOL!NPCORE:= 'FALSE'

'IF' ~ BOOL RDDISC

'THEN' INSTR NUMBER:= INSTR NUMBER + 2

'IF' RDTOD (BOOL RDDISC) = CUAT

'THEN' STOREC (CUAT)

'ELSE'

'IF' BOOL RDDISC 'THEN'

'BEGIN'

'IF' SECTBYTE > 1

'THEN' SECTBYTE:= SECTBYTE - 1

'ELSE'

'BEGIN' SECTBYTE:= 744;

SECTADDR:= SECTADDR - 1

'END'

'END'

'ELSE'

'BEGIN' BYTENUMBER INPUT:= BYTENUMBER INPUT + 1;

INSTR NUMBER:= INSTR NUMBER + 1

'END'

'ELSE'

'BEGIN' R:= RDTOD (BOOL RDDISC);
FOR S := RDTOD (BOOL RDDISC)
WHILE R = CUAT \& S = CUAT
DO
BEGIN
IF R = CUAK \& S = CPLA THEN
BEGIN STOREC (R); STOREC (S);
FOR NI := 1 STEP 1 UNTIL 4
DO STOREC (RDTOD (BOOL RDDISC));
R := RDTOD (BOOL RDDISC)
END;
ELSE BEGIN STOREC (R); R := S END;
END;
STOREC (R)
END;
END; IMPCORE;

'PROCEDURE' SPE (MIC); 'BOOLEAN' M sometimes
BEGIN
'PROCEDURE' PLA (MIC); 'BOOLEAN' M sometimes
BEGIN
'INTEGER' BYTE1 SECTADDR, BYTE2 SECTADDR,
REMEMBERSECTADDR, R, S, N;
'BOOLEAN' SINGLE CHARACTER;
SINGLE CHARACTER := 'FALSE';
BYTE1 SECTADDR := RDTOC (MIC);
BYTE2 SECTADDR := RDTOC (MIC);
IF BYTE2 SECTADDR > 127 THEN
BEGIN
IF SEGm = CORESEGMENTS THEN ERROR
("SINGLE-CHAR,TRANSFER TO LAST CORE-SEGm",
MIC, 'FALSE', 'TRUE');
BYTE2 SECTADDR := BYTE2 SECTADDR - 128;
SINGLE CHARACTER := 'TRUE';
END;
REMEMBERSECTADDR := SECTADDR :=
BYTE1 SECTADDR + 256 := BYTE2 SECTADDR
IF SECTADDR > DISCSECTORS THEN ERROR
("NON-EXISTENT DISC-SECTOR ADDRESSED", MIC,
'FALSE', 'TRUE');
IF (RDTOC (MIC) = CUAT) \& (RDTOC (MIC) = CUAT)
THEN
BEGIN
IF MIC THEN INSTR NUMBER MIC := INSTR NUMBER MIC = 1
ELSE INSTR NUMBER := INSTR NUMBER = 1
ERROR
("PLA NOT FOLLOWED BY UAT\&UAT", MIC, 'FALSE',
'TRUE');
END;
SECTBYTE := 3; BOOL RDDISC := 'TRUE'
IF SINGLECHARACTER THEN
BEGIN IF SEGm = 0 THEN BUFFERSEGm := 'FALSE';
SECTBYTE := 6;
FOR NI := 1, 2, 3
DO CORE [SEGm, 62, N] := READ DISC;
776  'COMMENT' TYPEFACECONSTANT;
777  'FOR' N:= 1, N + 1 'WHILE' BOOL RDDISC 'DO'
778  'BEGIN' R:= READ DISC
779  'FOR' S:=READ DISC
780  'WHILE' ¬ (R = CUAK ∨ S = CSPE)
781  'DO' R:= S; R:= READ DISC;
782  'IF' ¬ R = CUAK 'THEN'
783  'BEGIN'
784  'IF' REMEMBERSECTADDR = SECTADDR
785  'THEN' CORE [SEG, R, 1]:= SECTADDR
786  'ELSE' CORE [SEG, R, 1]:= SECTADDR + 1
787  CORE [SEG, R, 2]:= ¬ 1
788  'END'
789  'ELSE' 'IF' READ DISC = CTRE 'THEN' TRE
790  'END'
791  'END'
792  'ELSE'
793  'FOR' S:= READ DISC 'WHILE' BOOL RDDISC
794  'DO' INTERPRET (S, 'FALSE', 'TRUE', PLOTTER)
795  'END' PLA)
796  'END' PLA)
797  'PROCEDURE' PLE;
798  'BEGIN' 'INTEGER' SECTBYTES;
799  'PROCEDURE' STORED (X);
800  'BEGIN'
801  'PROCEDURE' STORED (X);
802  'BEGIN'
803  'IF' SECTADDR > DISCSECTORS
804  'THEN' ERROR
805  "NON-EXISTENT DISC-SECTOR ADDRESSED",
806  'FALSE', 'FALSE', 'TRUE')
807  DISC [SECTADDR, SECTBYTES] := X
808  'IF' SECTBYTES < 744
809  'THEN' SECTBYTES := SECTBYTES + 1
810  ELSE
811  'BEGIN' SECTADDR := SECTADDR + 1; SECTBYTES := 1;
812  'END'
813  'END' STORED;
814  'PROCEDURE' CHECK;
815  'BEGIN' R:= READ DATAPE; S:= READ DATAPE;
816  'IF' ¬ (R = CUAK) ∨ (¬ (S = CSPE))
817  'THEN' ERROR
818  "TYPEF, EL, DOES NOT BEGIN WITH UAK SPE",
819  'MIC', 'FALSE', 'TRUE')
820  ELSE 'BEGIN' STORED (R); STORED (S) 'END'
821  'END'
822  'PROCEDURE' ELEMENT INTO DISC;
823  'PROCEDURE' UATUAT (MICROPROGRAM);
'ELSE'
'BEGIN'
'IF' ~ MICROPROGRAM 'THEN'
'BEGIN'
'IF' READ DTAPE = CUAT
'THEN' STORED (CUAT)
'ELSE' BYTENUMBER [INPUT]= 1
'BEGIN'
BYTENUMBER [INPUT] = 1
BOOL CHAR TO DISC; = 'FALSE'
'ELSE' INSTR NUMBER; = INSTR NUMBER + 2
'BEGIN'
'ELSE'
'SET DISC'; = 'FALSE'
'END'
'BEGIN'
'THEN' READ DTAPE
'SET DISC'; = 'FALSE'
'ELSE'
'BEGIN'
'SET DISC'; = 'FALSE'
'END'
'BEGIN'
'BEGIN'
'R' = SECTADDR = 1
'S' = 744 + SECTBYTES = 6
'BEGIN'
'T' = SECTADDR = 5
'ELSE'
'BEGIN'
'T' = SECTADDR = 1
'U' = 744 + SECTBYTES = 5
'ELSE'
'DISC'; = 'FALSE'
'DISC'; = 'FALSE'
'DISC'; = 'FALSE'
'DISC'; = 'FALSE'
         'TRUE')
'ELSE'
'BEGIN'
'S' = STORED (R); 
'S' = STORED (S); 
'S' = STORED (T)
'ELSE'
'BEGIN'
'R' = READ DTAPE; STORED (R)
'R' = CUAK 'THEN'
'BEGIN'
'R' = READ DTAPE; 
'S' = READ DTAPE
'T' = READ DTAPE
'IF' ~ (R # CTRE) ~ (S = CUAT) ~ (T = CUAT)
'T' = SECTADDR
'THERE' ERROR
("SPE UAK NOT FOLLOWED BY TRE UAT UAT", 
FALSE', 'FALSE', 'TRUE'))
ELSE'
'BEGIN'
'R' < 1 ~ R = 28 ~ R = 62 ~ R = 251 ~
'R' > 254 
'THERE' ERROR
("INSTR. NOT ALLOWED AS WORD-ADDRESS", 
'FALSE', 'FALSE', 'TRUE'))
R := READ DTAPE;
STORED (R); S := READ DTAPE;
'IF' S = 0 ∧ R < 63
'THEN' ERROR
("IMAGE-AREA BELOW TYPEFACEMENT", 'FALSE', 'FALSE', 'TRUE');
'IF' S > CORESEGMENTS
'THEN' ERROR
("REL. SEG. ADDRESS TOO LARGE", 'FALSE', 'FALSE', 'TRUE');
STORED (S); T := READ DTAPE;
'IF' T = (T = 0 ∧ T = 1 ∧ T = 2 ∧ T = 32)
'THEN' ERROR
("THIRDBYTE SECONDARY ADDRESS-WORD", 'FALSE', 'FALSE', 'TRUE');
STORED (T); MICROPROGRAM := T = 2;
BOOL CHAR TO DISC := MICROPROGRAM;
'IF' BOOL CHAR TO DISC
'THEN' INSTR NUMBER := INSTR NUMBER + 1,
UATUAT (MICROPROGRAM);

'END';
'END' ELEMENT ONTO DISC;
SECTORAR := READ DTAPE + 256 * READ DTAPE;
'IF' SECTORAR > DISCSECTORS
'THEN' ERROR
("NON-EXISTENT DISC-SECTOR ADDRESSED", 'FALSE', 'FALSE', 'TRUE');
SECTBYTES := 1; BUFFERSEG := 'TRUE';
'FOR' N := 1, 2 'DO';
'BEGIN' R := READ DTAPE;
'IF' R = CUA'T THEN' STORED (R)
'ELSE' ERROR
("PLE NOT FOLLOWED BY UAT UAT", 'FALSE', 'FALSE', 'TRUE');
'END';
CHECK; R := READ DTAPE;
'IF' R = 62 'THEN';
'BEGIN' STORED (R); R := READ DTAPE; S := READ DTAPE;
T := READ DTAPE;
'IF' T = (0 ∧ 'LE' R ∧ 'LE' 19) ∨
(32 'LE' R ∧ 'LE' 51) ∨
(64 'LE' R ∧ 'LE' 83))
'THEN' ERROR
("BYTE ONE TYPEFCONST, NOT ALLOWED VALUE", 'FALSE', 'FALSE', 'TRUE');
'S' 'NE' 0
'THEN' ERROR
("SEC. BYTE TYPEFACEMENT, NOT EQUAL ZERO", 'FALSE', 'FALSE', 'TRUE');
'T' 'NE' 0
'THEN' ERROR
("THIRD BYTE TYPEFACEMENT, NOT EQUAL ZERO", 'FALSE', 'FALSE', 'TRUE');
STORED (R); STORED (S); STORED (T);
'END';
'ELSE' ERROR
("TYPECOUNT WITHOUT TYPEFACEMENT-CONSTANT", 'FALSE', 'FALSE', 'TRUE');
'FOR' N := 1, 2 'DO';
"BEGIN" R := READ DTape
   "IF" R = QUAT "THEN" STORED (R)
   "ELSE" ERROR
   ("TYPEFACE CONSTANT NOT FOLLOWED BY UAT UAT",
    "FALSE", "FALSE", "TRUE")
"END"

ELEMENT ONTO DISC
"END" PLE

"PROCEDURE" TRE; BOOL RDISC := "FALSE"
PA := RDTODOC (BOOL RDISC, MIC);
"IF" PA = CUAK "THEN"
"BEGIN" R := RDTODOC (BOOL RDISC, MIC)
   "IF" R = CPLA "THEN" PLA (MIC) "ELSE"
   "IF" MIC "THEN" ERROR
   ("INSTRUCTION NOT PERMITTED IN MICROPROGRAM",
    "TRUE", "FALSE", "TRUE")
   "ELSE"
   "IF" R = CPLE "THEN" PLE "ELSE"
   "IF" R = CTRR "THEN" TRE
   "ELSE" ERROR
   ("INSTRUCTION FOLLOWING SPE NOT PERMITTED",
    "MIC", "FALSE", "TRUE")
"ELSE" INPCORE ("FALSE")
"END"

"IF" MIC "THEN" ERROR
   ("INSTRUCTION NOT PERMITTED IN MICROPROGRAM",
    "TRUE", "FALSE", "TRUE")
"ELSE"
   "IF" PA < 1 || PA = 28 || PA = 251 || PA > 254
   "THEN" ERROR
   ("INSTRUCTION NOT ALLOWED AS WORD-ADDRESS",
    "MIC", "FALSE", "TRUE")
   "ELSE" INPCORE ("FALSE")
"END" SPE

"PROCEDURE" CALL TYPEFOUNT ELEMENT
   (PRIMARY ADDRESS, PLOTTER, CELLHEIGHT, CELLWIDTH, SHADING,
    REMAINDER, GRIDTYPE, KURN)
   "INTEGER" PRIMARY ADDRESS, SHADING, REMAINDER, GRIDTYPE, KURN;
   "REAL" CELLWIDTH, CELLHEIGHT; BOOLEAN PLOTTER;
"BEGIN"
   "INTEGER" B0, B1, B2, B3, B4, B5, B6, B7, INDEX, REPEAT,
   SECPAMIC, SECESEGIC, BYTEMIC, LINESTART, N, R, S, T
   "REAL" YPOSITION;
   "BOOLEAN" SINGLE BYTE, LINE REPETITION, BOOLUAT,
   FIRST WHITE ELEMENT;

"INTEGER" PROEDURE READ BYTE;
"BEGIN" "INTEGER" N, M "INTEGER ARRAY" BB [0 : 7]
M := READ CORE
"FOR" N := 7 "STEP" - 1 "UNTIL" 0 "DO"
"BEGIN" BB [N] := ENTER (M / 10 ** N)
"IF" BB [N] "NE" 0 "AND" BB [N] "NE" 1
"THEN" ADDRESS ERR
M := M - BB [N] * 10 ** N
"END"
1016 'IF' BB [7] = 0 'THEN' B7:= 0 'ELSE' B7:= 128;
1017 'IF' BB [6] = 0 'THEN' B6:= 0 'ELSE' B6:= 64;
1018 'IF' BB [5] = 0 'THEN' B5:= 0 'ELSE' B5:= 32;
1019 'IF' BB [4] = 0 'THEN' B4:= 0 'ELSE' B4:= 16;
1020 'IF' BB [3] = 0 'THEN' B3:= 0 'ELSE' B3:= 8;
1022 'IF' BB [1] = 0 'THEN' B1:= 0 'ELSE' B1:= 21;
1023 'IF' BB [0] = 0 'THEN' B0:= 0 'ELSE' B0:= 1;
1024 READ BYTE:= B0 + B1 + B2 + B3 + B4 + B5 + B6 + B7;
1025 'END! READ BYTE;
1026 'INTEGER', 'PROCEDURE', 'READ LINE ELEMENT';
1027 'BEGIN', 'INTEGER' BB;
1028 BB:= B5 + B4 + B3 + B2 + B1 + B0;
1029 'ELSE'
1030 'IF' B6 = 64 'THEN' READ LINE ELEMENT:= BB + B6
1031 'ELSE'
1032 'IF' B6 = 64 'THEN' READ LINE ELEMENT:= BB
1033 'ELSE' READ LINE ELEMENT:= READ BYTE + 64 + BB
1034 'END! READ LINE ELEMENT;
1035 'PROCEDURE', 'LBL', 'ONE BYTE', 'CELLWIDTH'; 'BOOLEAN' 'ONE BYTE;
1036 'REAL' 'CELLWIDTH';
1037 'BEGIN', 'INTEGER' R, S;
1038 'BEGIN' 'INTEGER' R, S;
1039 SINGLE BYTE:= B0 = 0;
1040 'IF' B6 = 64 'THEN' S:= -1 'ELSE' S:= 1;
1041 'IF' ONE BYTE 'THEN' R:= (B5 + B4 + B3) / 8
1042 'ELSE' R:= READ BYTE;
1043 'IF' R > 0 'THEN' LINECOUNT:= LINECOUNT + 1;
1044 'END! PLOTTER
1045 'THEN' PLOTRELATIVE
1046 ('FALSE', 'CELLWIDTH = R * S, 0, -2, 0')
1047 'ELSE' PRINTRELATIVE
1048 ('FALSE', 'FALSE', R * S, 0, LINENUMBERMAX, 'FALSE')
1049 'END! LBL;
1050 'PROCEDURE', 'BLW', 'ONE BYTE', 'REPEAT'; 'INTEGER' 'REPEAT;
1051 'BOOLEAN' 'ONE BYTE;
1052 'BEGIN' 'INTEGER' B0 = 0;
1053 'IF' ONE BYTE 'THEN' 'REPEAT':= (B6 + B5 + B4 + B3) / 8
1054 'ELSE' 'REPEAT':= READ BYTE;
1055 'LINE REPETITION':= 'TRUE';
1056 'END! BLW;
1057 'PROCEDURE', 'IMAGE', 'LINE
1058 'CELLWIDTH', 'CELLHEIGHT', 'SHADING', 'REMAINDER', 'REPEAT,' 'YPERPOSITION, PLOTTER, LINE REPETITION, KURNJI
1059 'INTEGER', 'SHADING', 'REMAINDER', 'REPEAT, KURN;
1060 'REAL' 'CELLWIDTH', 'CELLHEIGHT', 'YPERPOSITION;
1061 'BOOLEAN', 'PLOTTER, LINE REPETITION;
1062 'BEGIN' 'INTEGER' INDEX, N; 'REAL' YTOTAL, XPOSITION, Y;
1063 'BOOLEAN' 'WHITE ELEMENT';
1064 'INTEGER', 'ARRAY', 'REPEATED LINE [1 : 276];
1065 'PROCEDURE', 'STORE REPEATED LINE (X); 'INTEGER' X;
1066 'BEGIN' 'REPEATED LINE [INDEX]:= X; INDEX:= INDEX + 1
1067 'END';
1068 'PROCEDURE', 'WHITE
1069 'Y, 'CELLHEIGHT, 'PLOTTER, FIRST WHITE ELEMENT, KURN);
1076 'VALUE Y; 'REAL Y, CELLMANHT;
1077 'BOOLEAN PLOTTER, FIRST WHITE ELEMENT;
1078 'INTEGER KURN;
1079 'BEGIN'
1080 'IF' FIRST WHITE ELEMENT 'THEN'
1081 'BEGIN' Y := Y - 1; FIRST WHITE ELEMENT := 'FALSE'
1082 'END'
1083 'IF' PLOTTER 'THEN'
1084 'BEGIN' Y := Y + CELLMANHT; YTOTAL := YTOTAL + Y
1085 PLOTRELATIVE ('FALSE', 0, Y, = 2, KURN)
1086 'END'
1087 'ELSE' PRINTRELATIVE
1088 ('FALSE', 'FALSE', 0, Y, LINENUMBERMAX, 'FALSE')
1089 'END' WHITE;
1090
1091 'PROCEDURE' BLACK
1092 (Y, SHADING, REMAINDER, XPOSITION, CELLMANHT,
1093 PLOTTER, KURN);
1094 'VALUE' Y; 'REAL' Y, XPOSITION, CELLMANHT;
1095 'INTEGER' SHADING, REMAINDER, KURN; 'BOOLEAN' PLOTTER;
1096 'BEGIN' 'INTEGER' N;
1097 'IF' PLOTTER 'THEN'
1098 'BEGIN' Y := Y + CELLMANHT; YTOTAL := YTOTAL + Y
1099 PLOTRELATIVE ('FALSE', 1, 0, = 1, KURN)
1100 'FOR' N := 1 'STEP' 1 'UNTIL' SHADING 'DO'
1101 'BEGIN'
1102 PLOTRELATIVE ('FALSE', 0, Y, = -1, KURN)
1103 PLOTRELATIVE ('FALSE', 2, 0, = -1, KURN)
1104 PLOTRELATIVE ('FALSE', 0, Y, = -1, KURN)
1105 PLOTRELATIVE ('FALSE', 2, 0, = -1, KURN)
1106 'END'
1107 'IF' REMAINDER = 0 'THEN'
1108 'BEGIN'
1109 PLOTRELATIVE ('FALSE', 0, Y, = -2, KURN)
1110 PLOTRELATIVE ('FALSE', -4 * SHADING - 1, 0, = -2, 0)
1111 'END'
1112 'ELSE'
1113 'BEGIN'
1114 'IF' REMAINDER := 1 'THEN'
1115 'BEGIN'
1116 PLOTRELATIVE ('FALSE', -1, 0, = -1, 0)
1117 PLOTRELATIVE ('FALSE', 0, Y, = -1, KURN)
1118 PLOTRELATIVE ('FALSE', -4 * SHADING, 0, = 2, 0)
1119 'END'
1120 'ELSE'
1121 'BEGIN'
1122 'IF' REMAINDER := 2 'THEN'
1123 'BEGIN'
1124 PLOTRELATIVE ('FALSE', 0, Y, = -1, KURN)
1125 PLOTRELATIVE ('FALSE', -4 * SHADING - 1, 0, = 2, 0)
1126 'END'
1127 'ELSE'
1128 'BEGIN'
1129 'IF' REMAINDER := 3 'THEN'
1130 'BEGIN'
1131 PLOTRELATIVE ('FALSE', 0, Y, = -1, KURN)
1132 PLOTRELATIVE ('FALSE', 1, 0, = -1, 0)
1133 PLOTRELATIVE ('FALSE', 0, Y, = -1, KURN)
1134 PLOTRELATIVE ('FALSE', 0, Y, = -2, KURN)
PLOTRELATIVE
('FALSE', -4 * SHADING - 2, 0, -2, 0)
'END'
'END'
'ELSE'
'FOR' N := 1 'STEP' 1 'UNTIL' Y
'DO' PRINTRELATIVE
('FALSE', 'FALSE', 0, 1, LINENUMBERMAX,
'TRUE')
'END' BLACK)

PROCEDURE CHARACTER BASE
(PLOTTER, YPOSITION, YTOTAL, KURN)
'REAL' YPOSITION, YTOTAL; 'INTEGER' KURN
'BOOLEAN' PLOTTER;
'BEGIN'
'IF' PLOTTER 'THEN'
'BEGIN'

PLOTRELATIVE
('FALSE', 0, YPOSITION - YTOTAL, = 2, KURN));

PLOTRELATIVE ('FALSE', CELLWIDTH, 0, = 2, 0)
'END'
'ELSE' PRINTRELATIVE
('FALSE', 'TRUE', 1, YPOSITION,
LINENUMBERMAX, 'FALSE');

FIRST WHITE ELEMENT = 'TRUE';

'END' CHARACTER BASE)

PROCEDURE BLE
(CELLWIDTH, CELLHEIGHT, SHADING, REMAINDER, REPEAT,
YPOSITION, YTOTAL, PLOTTER, LINE REPETITION, KURN)
'INTEGER' SHADING, REMAINDER, REPEAT, KURN;
'REAL' CELLWIDTH, CELLHEIGHT, YPOSITION, YTOTAL;
'BOOLEAN' PLOTTER, LINE REPETITION;
'BEGIN' 'INTEGER' N; 'BOOLEAN' WHITE ELEMENT)

'INTEGER' 'PROCEDURE' READ REPEATED LINE;
'BEGIN'

READ REPEATED LINE := REPEATED LINE [INDEX];
INDEX := INDEX + 1
'END';

R := READ LINE ELEMENT;

'IF' LINE REPETITION 'THEN'

'BEGIN' STORE REPEATED LINE (R)
STORE REPEATED LINE (= 1)
'END';

BLACK
(R, SHADING, REMAINDER, XPOSITION, CELLHEIGHT,
PLOTTER, KURN);

CHARACTER BASE (PLOTTER, YPOSITION, YTOTAL, KURN))

'IF' LINE REPETITION 'THEN'

'BEGIN'

'FOR' N := 1 'STEP' 1 'UNTIL' REPEAT 'DO'

'BEGIN' INDEX := 1; WHITE ELEMENT := 'TRUE';

'IF' PLOTTER 'THEN'

'BEGIN' XPOSITION := PLOT (0, 0, 15);

YTOTAL := YPOSITION

'END';

'FOR' R := READ REPEATED LINE 'WHILE' R > 0
'DO'  
'BEGIN'  
'IF'  WHITE ELEMENT  
'THEN'  WHITE  
(R, CELLHEIGHT, PLOTTER,  
FIRST WHITE ELEMENT, KURN)  
ELSE BLACK  
(R, SHADING, REMAINDER,  
XPOSITION, CELLHEIGHT,  
PLOTTER, KURN)  
WHITE ELEMENT = ¬ WHITE ELEMENT;  
'END';  
CHARACTER BASE  
(PLOTTER, YPOSITION, YTOTAL, KURN)  
'END')  
'END';  
LINE REPETITION = 'FALSE';  
'END';  
'END' BLE;  
LINECOUNT = LINECOUNT + 1;  
'IF' PLOTTER 'THEN'  
'BEGIN' XPOSITION = PLOT (0, 0, 15);  
YTOTAL = YPOSITION  
'END';  
WHITE ELEMENT = 'FALSE';  
'IF' LINE REPETITION 'THEN'  
'FOR' INDEX = 1 'STEP' 1 'UNTIL' 276  
'DO' REPEATED LINE [INDEX] = 0  
INDEX = 1;  
'FOR' N = 1, N + 1 'WHILE' B7 = 0 'DO'  
'BEGIN' WHITE ELEMENT = ¬ WHITE ELEMENT;  
Y = READ LINE ELEMENT;  
'IF' LINE REPETITION  
'THEN' STORE REPEATED LINE (Y);  
'IF' WHITE ELEMENT  
'THEN' WHITE  
(Y, CELLHEIGHT, PLOTTER,  
FIRST WHITEELEMENT, KURN)  
ELSE BLACK  
(Y, SHADING, REMAINDER, XPOSITION,  
CELLHEIGHT, PLOTTER, KURN);  
READ BYTE)  
'END';  
BLE  
(CELLWIDTH, CELLHEIGHT, SHADING, REMAINDER, REPEAT,  
YPOSITION, YTOTAL, PLOTTER, LINE REPETITION, KURN)  
'END' IMAGE LINE;  
'PROCEDURE' ADDRESS ERR;  
ERROR  
("ADDRESS CORE-AREA NOT LOADED WITH ERROR=FREE  
CHARACTER-DATA", MIC, 'FALSE', 'TRUE');  
'IF' BUFFERSEG = SEGNUM = 0  
'THEN' ERROR  
("CHAR.CALL FROM SEG.0 WHICH HAS BEEN USED AS  
BUFFER FOR DISC-INPUT", MIC, 'FALSE', 'TRUE');  
'IF' SEGNUM = ¬ 1  
'THEN' ERROR  
("CHARACTERCALL BEFORE FIRST SEG INSTRUCTION",  


1256 'FALSE', 'FALSE', 'TRUE')"
1257 'IF' CORE [SEG, PRIMARY ADDRESS, 2] = 1 'THEN'
1258 'BEGIN' SECTADDR := CORE [SEG, PRIMARY ADDRESS, 1]
1259 SECTBYTE := 1; R := READ DISC; S := READ DISC
1260 'FOR' T := READ DISC 'WHILE'
1261 (R = CUAK ∧ S = CSPE ∧ T = PRIMARY ADDRESS)
1262 'DO' 'BEGIN' R := S; S := T 'END'
1263 SECTBYTE := SECTBYTE + 2; THIRDBYTE := READ DISC;
1264 BOOL RDDISCI := 'TRUE'; INCORE ('TRUE')
1265 BOOL RDDISCI := 'FALSE'; SECPL := 0; SECSEG := SEG + 1
1266 'END'
1267 'ELSE'
1268 'BEGIN' SECPL := CORE [SEG, PRIMARY ADDRESS, 1]
1269 SECSEG := CORE [SEG, PRIMARY ADDRESS, 2] + SEG;
1270 'IF' SECSEG > CORESEGM + 'THEN' ADDRESS ERR
1271 THIRDBYTE := CORE [SEG, PRIMARY ADDRESS, 3];
1272 'IF' ... (THIRDBYTE = 0 ∧ THIRDBYTE = 1 ∧ THIRDBYTE = 2
1273 ∧ THIRDBYTE = 32)
1274 'THEN' ADDRESS ERR
1275 'END'
1276 FIRST WHITE ELEMENT := 'TRUE'
1277 BYTE := '1'; LINE REPTITION := 'FALSE'
1278 'IF' THIRDBYTE = 2 'THEN'
1279 'BEGIN' 'INTEGER' REMEMBERSEGFM
1280 REMEMBERSEGFM := SEGFM
1281 SEG := SEGFM
1282 'IF' MIC
1283 'THEN' ERROR
1284 ('MICROPROGRAM CONTAINS MICROPROGRAM=CALL'
1285 'TRUE', 'FALSE', 'TRUE')
1286 MIC := 'TRUE'; INSTR NUMBER MICI := 0
1287 'FOR' S := READ CORE 'WHILE' S = CUAAT
1288 'DO' INTERPRET (S, 'TRUE', 'FALSE', PLOTTER)
1289 MIC := 'FALSE'; SEGFM := SEG := REMEMBERSEGFM
1290 'END'
1291 'ELSE'
1292 'BEGIN'
1293 SEGFM := SEGFM
1294 'IF' KURN = 1
1295 'THEN' ERROR
1296 ('CHARACTER=CALL BEFORE FIRST GRO=INSTRUCTION'
1297 MIC, 'FALSE', 'TRUE')
1298 CHAR := 'TRUE'; LINECOUNT := 0;
1299 R :=
1300 ('IF' THIRDBYTE = 0 'THEN' 3.75 ELSE 15) * GRIDTYPE;
1301 'COMMENT' ROUNDING OFF;
1302 LINESTART :=
1303 ('IF' THIRDBYTE = 1 'THEN' 0
1304 'ELSE' ('IF' PLOTTER 'THEN' 10 * FACTOR 'ELSE' 1) + R;
1305 'IF' PLOTTER 'THEN'
1306 'BEGIN' YPOSITION := PLOT (0, 0, 16) - LINESTART;
1307 PLOTRELATIVE ('FALSE', 0, - LINESTART, - 2, KURN)
1308 'END'
1309 'ELSE'
1310 'BEGIN' YPOSITION := PRINTPOSITION - LINESTART;
1311 PRINTRELATIVE
1312 ('FALSE', 'FALSE', 0, - LINESTART, LINESNUMBERMAX,
1313 'FALSE')
1314 'END'
1315 'BOOLUAT := 'FALSE'}
FOR N := 1, N + 1 WHILE NOT BOOLUT 'DO
BEGIN READ BYTE;
  IF B7 = 128 THEN
  BEGIN
    IF B2 = 4 THEN
      BEGIN
        IF B1 = 0 THEN LBL ('TRUE', CELLWIDTH)
        ELSE
          IF B1 = 2 AND B3 = 0 AND B4 = 0 AND B5 = 0 THEN LBL ('FALSE', CELLWIDTH)
          ELSE ADDRESS ERR
      END
    ELSE
      BEGIN
        IF B1 = 0 THEN BLW ('TRUE', REPEAT)
        ELSE
          BEGIN
            IF B3 = 0 AND B4 = 0 AND B5 = 0 AND B6 = 0 THEN BLW ('FALSE', REPEAT)
            ELSE
              IF B3 = 8 AND B4 = 16 AND B5 = 32 AND B6 = 64 THEN
                BEGIN
                  IF PLOTTER THEN PLOTRELATIVE ('FALSE', 0, LINSTART, -2, KURN)
                  ELSE PRINTRELATIVE ('FALSE', 'FALSE', 0, LINSTART, LINENUMBERMAX, 'FALSE')
                  BOOLUT := 'TRUE'
                END
              ELSE ADDRESS ERR
          END
        END
      END
    END
  END
END
'END'
'ELSE' ADDRESS ERR
'END'
'END'
'ELSE' IMAGE LINE
(CELLWIDTH, CELLHEIGHT, SHADING, REMAINDER, REPEAT, YPOSITION, PLOTTER,
LINE REPETITION, KURN)
'END'}
CZWQ:= 00110010; CZWR:= 00000000; CSTOP:= 00100111

ENLARGEMENT FACTOR[0]=1; ENLARGEMENT FACTOR[1]=1.125;
ENLARGEMENT FACTOR[6]=2; ENLARGEMENT FACTOR[7]=2.25;
ENLARGEMENT FACTOR[8]=2.5; ENLARGEMENT FACTOR[9]=2.75;
ENLARGEMENT FACTOR[12]=4;
CHAR:= 'FALSE';

'BEGIN'

'COMMENT' IN THIS BLOCK, THE INPUT IS TRANSLATED INTO
DIGEST MACHINE-CODE, WHICH IS STORED IN THE ARRAY
DTAPE;

INTEGER MEDIUM_LINES, NM, BEFORE_LEFT_COMMAND,
BEFORE_RIGHT_COMMAND, AFTER_RIGHT_COMMAND,
LEFT_COMMAND, RIGHT_COMMAND, I, J, INSTR COUNT,
KENBYTES, WORD POINTER, SEG POINTER, EL START;
REAL A;

INTEGER AARRAY [1 : MAXLINE.NUMBERCHAR, 1 : 20];
BOOLEAN BOOLSTOP, NUMBER, BOOKEN, BOOLUBL, XMEND, ADRCAL;

INTEGER PROCEDURE SKIP (INITIAL VALUE);
BOOLEAN INITIAL VALUE;
COMMENT THE GLOBAL PARAMETER INSTR NUMBER IS CHANGED;
BEGIN INTEGER S;
IF INITIAL VALUE THEN
BEGIN INSTR NUMBER:= 2; SKIP:= 0; NEW PAGE;
PRINT (1) SPACE (40 - PRINTPOS);
END;
ELSE
BEGIN
FOR S:= RESYM WHILE (S = 93) OR (S = 119)
DO;
SKIP:= 5;
IF PRINTPOS = 100 THEN
BEGIN
NLCR; SPACE (40);
END;
PRINTSYM(S);
END;
END SKIP;

PROCEDURE FILL DTAPE (X); VALUE X; INTEGER X;
BEGIN
IF INSTR COUNT > DTAPELength
THEN ERROR
("INPUT CAUSES DTAPE-OVERFLOW", "FALSE", CHAR, "TRUE");
DTAPE [INSTR COUNT]:= X; INSTR COUNT:= INSTR COUNT + 1;
IF PUNCH THEN
BEGIN
IF CHAR = X 'GE' 10 THEN
BEGIN INTEGER A, B, N;
A:= B:= 0;
FOR N:= 7 'STEP' -1 'UNTIL' 0 'DO';
BEGIN
B:= ENTIER(X / 10 ** N);
X:= X - B * 10 ** N;
END;
END;
END;
A := A + (*IF* B = 1 *THEN* 2 **) N *ELSE* 0)

*END* PUMEP(A)

ELSE PUMEP(Y)

*END* FILL DTape

PROCEDURE CHARACTER (K, START); \*VALUE* K, START*

INTEGER K; BOOLEAN START;

BEGIN* INTEGER J; BOOLEAN SINGLE BYTE, UAT, L127*

PROCEDURE CODE SHORT LONG LIST (K, START);

INTEGER K; BOOLEAN START;

BEGIN* INTEGER N;

BOOLEAN FIRST COMMAND, LAST EMPTY LINE,

LAST REPLICATION;

PROCEDURE CODE CHAR ELEMENT (K); INTEGER K;

BEGIN* INTEGER BIT0, BIT6*

PROCEDURE CODE IMAGELINE (K); INTEGER K;

BEGIN* INTEGER J, BYTE1, BYTE2*

PROCEDURE LINE BYTES (A); \*VALUE* A*

INTEGER A;

BEGIN*

IF SINGLE BYTES THEN

BEGIN*

IF A > 63 THEN

BEGIN*

BYTE2 := FORM BYTE (13, 6, A);

BYTE1 := FORM BYTE (5, 0, A);

END*

ELSE*

BEGIN* BYTE2 := 0*

BYTE1 := FORM BYTE (5, 0, A) + 1000000

END*

ELSE*

BEGIN* BYTE1 := FORM BYTE (6, 0, A);

BYTE2 := 0

END*

END* LINE BYTES*

FOR J := 2;

J * 1 * WHILE* AA [K, J - 1] **NE' 0

DO*

BEGIN* LINE BYTES (AA [K, J]);

FILL DTape (BYTE1);

IF BYTE2 = 0 *THEN* FILL DTape (BYTE2)

END*

LINE BYTES (AA [K, J]);

FILL DTape (BYTE1 + 10000000);

IF BYTE2 = 0 *THEN* FILL DTape (BYTE2)

END* CODE IMAGELINE;

BEGIN* INTEGER* PROCEDURE FORM BYTE (NB, NO, A)
'VALUE' A; 'INTEGER' NB, NO, A;
'BEGIN' 'INTEGER' AA, B, NJ;
   BI= 0;
   A:=;
   A:= ENTER (A / 2 ** (NB + 1)) * 
   2 ** (NB + 1);
   'FOR' NJ= NB 'STEP' 1 'UNTIL' NO 'DO'
   'BEGIN' AA:= ENTER (A / 2 ** NJ);
   A:= A - AA * 2 ** NJ;
   BI= B + AA * 10 ** (N = NO)
   'END';
   FORM BYTE:= B;
   'END' FORM BYTE;

'IF' SINGLE BYTE 'THEN' BIT0:= 0
'ELSE' BIT0:= 1;
'IF' AA [K, 2] = 0 'THEN'
   'BEGIN'
   'IF' AA [K, 1] > 0 'THEN' BIT6:= 0
   'ELSE' BIT6:= 10000000
   'IF' ABS (AA [K, 1]) < 8
   'THEN' FILL DTAPE
       (10000100 + BIT0 + BIT6 * 
FORM BYTE (2, 0, ABS (AA [K, 1]))
   = 1000)
   'ELSE'
   'BEGIN'
   FILL DTAPE (10000110 + BIT0 + BIT6)
   FILL DTAPE
       (FORM BYTE (7, 0, ABS (AA [K, 1])))
   'END';
   'END';
   'ELSE'
   'IF' AA [K, 1] > 1 'THEN'
   'BEGIN'
   'IF' AA [K, 1] < 16
   'THEN' FILL DTAPE
       (10000000 + BIT0 + 
FORM BYTE (3, 0, AA [K, 1] - 1) * 
1000)
   'ELSE'
   'BEGIN' FILL DTAPE (10000010 + BIT0)
   FILL DTAPE
       (FORM BYTE (7, 0, AA [K, 1] - 1))
   'END';
   'CODE IMAGELINE (K)
   'END';
   'ELSE' 'CODE IMAGELINE (K)
   'END' 'CODE CHAR ELEMENT';

'PROCEDURE' 'CODE LIST (IC, JC): ' 'INTEGER' IC, JC;
'BEGIN' 'INTEGER' HC;
   'FOR' HC:= IC 'STEP' 1 'UNTIL' JC
   'DO' 'CODE CHAR ELEMENT (HC)
   'END' 'CODE LIST';

'PROCEDURE' 'CODE TAIL (N): ' 'INTEGER' N;
   'BEGIN' SINGLE BYTE:= 'TRUE';
   'IF' UAT 'THEN' 'CODE LIST (N, K) ' 'ELSE'

'END';
1556 'IF' LAST EMPTY LINE 'THEN'
1557 'BEGIN' CODE LIST (N, K - 2);
1558 SINGLE BYTE := 'FALSE';
1559 CODE LIST (K - 1, K)
1560 'END'
1561 'ELSE'
1562 'IF' LAST REPETITION 'THEN'
1563 'BEGIN' CODE LIST (N, K = 1);
1564 SINGLE BYTE := 'FALSE';
1565 CODE CHAR ELEMENT (K)
1566 'END'
1567 'ELSE'
1568 'IF' AFTER RIGHT COMMAND > 1 'THEN'
1569 'BEGIN' CODE LIST (N, K = 1);
1570 FILL TAPE (10000101);
1571 SINGLE BYTE := 'FALSE';
1572 CODE CHAR ELEMENT (K)
1573 'END'
1574 'ELSE':
1575 'BEGIN' CODE LIST (N, RIGHT COMMAND - 1);
1576 SINGLE BYTE := 'FALSE';
1577 CODE LIST (RIGHT COMMAND, K)
1578 'END'
1579 'END' CODE TAIL;
1580
1581 FIRST COMMAND := AA [1, 1] > 1 = AA [1, 2] = 0;
1582 'IF' K > 1 'THEN'
1583 'BEGIN' LAST EMPTY LINE := AA [K - 1, 2] = 0;
1584 LAST REPETITION := AA [K, 1] > 1
1585 'END'
1586 'ELSE' LAST EMPTY LINE := LAST REPETITION :=
1587 'FALSE';
1588 'IF' FIRST COMMAND 'THEN' CODE TAIL (1) 'ELSE'
1589 'IF' LEFT COMMAND > 1 'THEN'
1590 'BEGIN'
1591 'IF' BEFORE LEFT COMMAND > 1 = START 'THEN'
1592 'BEGIN' SINGLE BYTE := 'TRUE';
1593 FILL TAPE (10000100)
1594 'END'
1595 CODE LIST (1, LEFT COMMAND - 1);
1596 CODE TAIL (LEFT COMMAND)
1597 'END'
1598 'ELSE'
1599 'IF' MEDIUM LINE > 1 =
1600 (A'UT = LAST EMPTY LINE = LAST REPETITION)
1601 'THEN'
1602 'BEGIN' SINGLE BYTE := 'TRUE';
1603 FILL TAPE (10000100);
1604 'IF' A'UT 'THEN' CODE LIST (1, K) 'ELSE'
1605 'IF' LAST EMPTY LINE 'THEN'
1606 'BEGIN' CODE LIST (1, K = 2);
1607 SINGLE BYTE := 'FALSE';
1608 CODE LIST (K - 1, K)
1609 'END'
1610 'ELSE'
1611 'IF' LAST REPETITION 'THEN'
1612 'BEGIN' CODE LIST (1, K = 1);
1613 SINGLE BYTE := 'FALSE';
1614 CODE CHAR ELEMENT (K)
1615 'END'
'END'
'ELSE'
'IF' MEDIUM LINES > 2 'THEN'
'BEGIN' FILL DTape (10000101)
'SINGLE BYTE= 'TRUE'; CODE LIST (1, K = 1)
FILL DTape (10000101); SINGLE BYTE= 'FALSE';
CODE CHAR ELEMENT (K)
'END'
'ELSE'
'BEGIN'
'IF' START 'THEN'
'BEGIN' SINGLE BYTE= 'FALSE'
FILL DTape (10000101)
'END'
'END'
CODE LIST (1, K)
'END'
'END' CODE SHORT LONG LIST
PROCEDURE READ CHAR ELEMENT (K, J); 'INTEGER' K, J;
'BEGIN' 'INTEGER' R, S;
S := SKIP ('FALSE');
'IF' S = 65 'THEN'
'BEGIN' S := RESYM; PRSYM (S); S := S 'END';
'IF' S < 10 'THEN'
'BEGIN' R = S;
'FOR' S := SKIP ('FALSE') 'WHILE' S < 10
'DO' R := R * 10 + S
AA [K, J] := R;
'IF' R = 0 'THEN'
'THEN' ERROR
"ZERO-VALUED LINE-ELEMENT", 'FALSE', 'TRUE', 'FALSE');
'IF' J = 1 'THEN'
'BEGIN' L127 := UAT := 'FALSE'; NM := 0 'END'
'ELSE'
'IF' AA [K, J] > 127 'THEN' L127 := 'TRUE'
'ELSE' 'IF' AA [K, J] > 63 'THEN' NM := NM + 1
'IF' S = 87
'BEGIN' read CHAR ELEMENT (K, J + 1)
'ELSE'
'IF' S = 99 'THEN'
'IF' J = 2 * ENTER (J / 2) = 1
'THEN' ERROR
"EVEN NUMBER OF LINE-ELEMENTS",
'FALSE', 'TRUE', 'FALSE');
'END'
'ELSE' ERROR
"LINE-ELEMENT NOT FOLLOWED BY ) OR ",
'FALSE', 'TRUE', 'FALSE');
'END'
'ELSE'
'IF' S = 98 'THEN' READ CHAR ELEMENT (K, 1) 'ELSE'
'IF' S = 91 'THEN' UAT := 'TRUE'
'ELSE' ERROR
"SYMBOL FOLLOWING LINE-ELEMENT NOT ALLOWED",
'FALSE', 'TRUE', 'FALSE');
'IF' BOOLERROR = # UAT 'THEN'
'BEGIN' LINECOUNT := LINECOUNT + 1
READ CHAR ELEMENT (K, 1)
'END'
'END' READ CHAR ELEMENT;

CHAR:= 'TRUE'; SINGLE BYTE:= 'FALSE';
'IF' K = 1 'THEN'
'BEGIN'
BEFORE LEFT COMMAND:= BEFORE RIGHT COMMAND:= 0;
MEDIUM LINES:=0;
LEFT COMMAND:= RIGHT COMMAND:= 1
'END';

READ CHAR ELEMENT (K, 1); LINECOUNT:= LINECOUNT + 1
'IF' = BOOLERROR 'THEN'
'BEGIN'

'IF' UAT 'THEN'
'BEGIN'
AFTER RIGHT COMMAND:=
MEDIUM LINES:= BEFORE RIGHT COMMAND;
CODE SHORT LONG LIST (K = 1, START));
'END';

'ELSE'

'IF' L127 'THEN'
'BEGIN'
AFTER RIGHT COMMAND:=
MEDIUM LINES:= BEFORE RIGHT COMMAND;
CODE SHORT LONG LIST (K, START);
K:= 1; START:= 'FALSE';
CHARACTER (1, 'FALSE');
'END;

'ELSE'

'IF' AA [K, 2] = 0 'THEN'
'BEGIN'

'IF' K > 1 'THEN'
'BEGIN'

'IF' AA [K - 1, 2] = 0 'THEN'
'BEGIN'
AA [K - 1, 1]:= AA [K, 1] + AA [K, 1]
K:= K - 1
'END';

'IF' LEFT COMMAND = 1 'THEN'
'BEGIN' LEFT COMMAND:= RIGHT COMMAND:= K;
BEFORE RIGHT COMMAND:=
BEFORE LEFT COMMAND:= MEDIUM LINES
'END';

'ELSE'

'BEGIN'
BEFORE RIGHT COMMAND:= MEDIUM LINES;
RIGHT COMMAND:= K
'END';

'END'}

CHARACTER (K + 1, START)
'END';

'ELSE';

'BEGIN'

'IF' AA [K, 1] > 1 ^ K > 1 'THEN'
'BEGIN'

'IF' LEFT COMMAND = 1 'THEN'
'BEGIN' LEFT COMMAND:= RIGHT COMMAND:= K;
BEFORE RIGHT COMMAND:=
BEFORE LEFT COMMAND:= MEDIUM LINES
'END'
'ELSE'
'BEGIN'
BEFORE RIGHT COMMAND:= MEDIUM LINES
RIGHT COMMAND:= K
'END'
'END'
MEDIUM LINES:= MEDIUM LINES + NM
CHARACTER (K + 1, START)
'END'
'END'
CHAR:= 'FALSE'
'END' CHARACTER
PROCEDURE: LISTINGNUMBER;
'BEGIN' NLCR; PRINT (INSTR NUMBER); SPACE (40 - PRINTPOS);
INSTR NUMBER:= INSTR NUMBER + 1
'END' LISTINGNUMBER;
PROCEDURE: FREE CORE(WORD POINTER, SEG POINTER);
INTEGER WORD POINTER, SEG POINTER;
'BEGIN' INTEGER LENGTH;
LENGTH:= (INSTR COUNT + 1 - EL START) / 31
WORD POINTER:= WORD POINTER +
('IF' LENGTH = ENTER LENGTH 'THEN' LENGTH
'ELSE' LENGTH + 1)
'IF' WORD POINTER > 255 'THEN'
'BEGIN'
WORD POINTER:= WORD POINTER - 256
SEG POINTER:= SEG POINTER + 1
'END'
'END' FREE CORE
PROCEDURE: UATT CHECK(NUMBER, BOOLSTOP);
BOOLEAN NUMBER, BOOLSTOP;
'BEGIN' INTEGER N; REAL ARRAY AA[1:2];
'FOR' N := 1, 2 'DO'
'BEGIN'
NUMBER:= 'TRUE'; LISTINGNUMBER;
READ INPUT ELEMENT) AA[N]:= A; A:= 0
'END'
'END'
'END'
'BEGIN'
FILL DTAPE(CUART); FILL DTAPE(CUAT)
'END'
'ELSE' ERROR("TYPE OF COUNT EL. NOT FOLLOWED BY UAT/JUAT!");
'FALSE', 'FALSE', 'FALSE');
'END' UATT CHECK
PROCEDURE: READ INPUT ELEMENT;
'BEGIN' 'INTEGER' S;
S:= SKIP ('FALSE'))
'IF' S = 98 'THEN'
'BEGIN'
'IF' A = 0 'THEN'
'BEGIN' A:= S; NUMBER:= 'FALSE'
'END'
'ELSE' ERROR
("INSTRUCTION CONTAINS NON-ALLOWED SYMBOL",
'FALSE', 'FALSE', 'FALSE')
'END'
'ELSE'
'IF' S = 100 'THEN'
'BEGIN'
  'IF' A 'NE' 0 'THEN'
  ERROR("INSTRUCTION CONTAINS NON-ALLOWED SYMBOL", 'FALSE', 'FALSE', 'FALSE');
  READ INPUT ELEMENT
  'END'
'ELSE'
  'IF' S = 101 'THEN'
  'BEGIN'
  'IF' SKIP('FALSE') 'NE' 91 'THEN'
  ERROR("INSTRUCTION CONTAINS NON-ALLOWED SYMBOL", 'FALSE', 'FALSE', 'FALSE');
  'IF' NUMBER 'THEN'
  'BEGIN' FILL DTape(A); LISTINGNUMBER 'END'
  'ELSE'
  'BEGIN'
  MIEEnd= 'TRUE';
  CODE SWITCH;
  MIEEnd= 'FALSE';
  'END';
  A:= 0;
  'IF' ADRCAL 'THEN'
  'BEGIN'
  FREE CORE( WORD POINTER, SEG POINTER);
  ADRCAL:= 'FALSE';
  'END';
  UATT CHECK(NUMBER, BOOLSTOP);
  LISTINGNUMBER;
  NUMBER:= 'TRUE';
  'END'
  'ELSE'
  'IF' S = 91 'THEN'
  'BEGIN'
  'IF' NUMBER 'THEN'
  'BEGIN'
  'IF' A 'NE' 256 'THEN' FILL DTape(A);
  LISTINGNUMBER;
  'IF' A = 62 & DTape[(INSTR COUNT-2) = CSPE & DTape[(INSTR COUNT-3) = CUAK 'THEN'
  'BEGIN' WORD POINTER:= 0; SEG POINTER:= 1; 'END';
  'ELSE' 'IF' A = 256 'THEN'
  'BEGIN'
  ADRCAL:= 'TRUE';
  SPACE(40 = PRINTPOS);
  PRINTTEXT("ADDRESS CALCULATION");
  LISTINGNUMBER;
  EL START:= INSTR COUNT + 2)
  FILL DTape(WORD POINTER);
  FILL DTape(SEG POINTER)
  'END';
  'IF' BOOLKEN 'THEN'
  'BEGIN' KENBYTES:= KENBYTES + 1;
  BOOLKEN:= KENBYTES = 3
  'END'
  'END'
  'ELSE'
  'END';
IF S < 10 THEN BEGIN A := A + 10 + S;
ELSE IF A > 256 THEN ERROR ("INTEGER CANNOT BE CODED IN ONE BYTE", FALSE, FALSE, FALSE);
READ INPUT ELEMENT
END ELSE
BEGIN
NUMBER := FALSE; A := A + 36 + S;
READ INPUT ELEMENT
END
END READ INPUT ELEMENT;

'PROCEDURE CODE SWITCH;
BEGIN
'IF A = 468 THEN FILL DTape (CD0) ELSE
'IF A = 17231 THEN FILL DTape (CDAN) ELSE
'IF A = 782362 THEN FILL DTape (CGROA) ELSE
'IF A = 782363 THEN FILL DTape (CGROB) ELSE
'IF A = 782364 THEN FILL DTape (CGROC) ELSE
'IF A = 782365 THEN FILL DTape (CGROD) ELSE
'IF A = 782366 THEN FILL DTape (CGROE) ELSE
'IF A = 782367 THEN FILL DTape (CGROF) ELSE
'IF A = 782368 THEN FILL DTape (CGROG) ELSE
'IF A = 782369 THEN FILL DTape (CGROH) ELSE
'IF A = 782370 THEN FILL DTape (CGR0I) ELSE
'IF A = 782371 THEN FILL DTape (CGR0J) ELSE
'IF A = 782372 THEN FILL DTape (CGR0K) ELSE
'IF A = 782373 THEN FILL DTape (CGR0L) ELSE
'IF A = 782374 THEN FILL DTape (CGR0M) ELSE
'IF A = 26447 THEN BEGIN
FILL DTape (CKEN);
ELSE BEGIN BOOLEK = TRUE; KENBYTES = 0 END
END
'IF = 972973 THEN FILL DTape (CKUR1) ELSE
'IF A = 972974 THEN FILL DTape (CKUR2) ELSE
'IF A = 972975 THEN FILL DTape (CKUR3) ELSE
'IF A = 36099 THEN FILL DTape (CNOR) ELSE
'IF A = 30909 THEN FILL DTape (CNUL) ELSE
'IF A = 33166 THEN FILL DTape (CPLA) ELSE
'IF A = 33170 THEN FILL DTape (CPLE) ELSE
'IF A = 1299044 THEN FILL DTape (CRUCK) ELSE
'IF A = 1322555 THEN FILL DTape (CSCHN) ELSE
'IF A = 36808 THEN FILL DTape (CSEG) ELSE
'IF A = 37202 THEN FILL DTape (CSEP) ELSE
'IF A = 48396233 THEN FILL DTape (CSTART) IF BOOLUBL = FALSE THEN END
'IF = 37955 THEN FILL DTape (CTAB) ELSE
'IF A = 38570 THEN FILL DTape (CTRE) ELSE
'IF A = 39260 THEN FILL DTape (CUAK) ELSE
'IF A = 39269 THEN FILL DTape (CUAT) ELSE
'IF A = 39297 THEN BEGIN
FILL DTape (CUBL) IF BOOLUBL = TRUE THEN END
ELSE
'IF A = 1465462 THEN FILL DTape (CVERA) ELSE
'IF A = 1465476 THEN FILL DTape (CVERO) ELSE
'IF' A = 1465482 'THEN' FILL DTAPE (CVERU) 'ELSE'
'IF' A = 42133 'THEN' FILL DTAPE (CWID) 'ELSE'
'IF' A = 45891 'THEN' FILL DTAPE (CZER) 'ELSE'
'IF' A = 45895 'THEN' FILL DTAPE (CZEV) 'ELSE'
'IF' A = 46538 'THEN' FILL DTAPE (CZWQ) 'ELSE'
'IF' A = 46539 'THEN' FILL DTAPE (CZWR) 'ELSE'
'IF' A = 1344841 'THEN'
'BEGIN' FILL DTAPE (CSTOP) 'BOOLSTOP' = 'TRUE' 'END'
'ELSE'
'IF' A = 98 'THEN'
'BEGIN'
LINECOUNT := 1; CHARACTER (1, 'TRUE'); AI := 0
'IF' ADRCAL 'THEN'
'BEGIN'
FREE CORE (WORD POINTER, SEG POINTER);
ADRCAL := 'FALSE'
'END';
'END'
UATT CHECK (NUMBER, BOOLSTOP)
'END'
'ELSE'
'IF' BOOLKEN 'THEN'
'BEGIN' KENBYTES := KENBYTES + 1 'FILL DTAPE (A)'
BOOLKEN := KENBYTES < 3
'END'
'ELSE'
'IF' BOOLUBL 'THEN' FILL DTAPE (A)
'ELSE' ERROR
("INSTRUCTION DOES NOT EXIST", 'FALSE',
'FALSE', 'FALSE')
'IF' BOOLSTOP 'THEN' NEW PAGE
'ELSE'
'IF' = MICEND 'THEN' LISTING NUMBER
'END' CODE SWITCH
'SKIP ('TRUE'); INSTR COUNT := 1
BOOLSTOP := BOOLKEN := BOOLUBL := ADRCAL := 'FALSE'
'FOR' I := 1, 1 + 1 'WHILE' = BOOLSTOP 'DO'
'BEGIN' AI := 0 'NUMBER' = 'TRUE'; READ INPUT ELEMENT
'IF' = NUMBER 'THEN' CODE SWITCH
'END'
'IF' BOOLERROR 'THEN' EXIT
'END' THE INPUT IS TRANSLATED INTO DIGISET MACHINE = CODE
BOOLWID := MIGI := BOOL RDDI := BOOL INCORE :=
BOOL CHAR TO DISC := BUFFERSEG := BOOLZWQ := CHAR := 'FALSE'
SEGHI := SECTYET := KURN := 1
BYTE_NUMBER INPUT := NINST NUMBER := 1
'FOR' R := 1 'STEP' 1 'UNTIL' CORESEGMENTS 'DO'
'FOR' S := 0 'STEP' 1 'UNTIL' 255 'DO'
'FOR' T := 1, 2, 3 'DO' CORE (R, S, T) := 0
'COMMENT' THIS INITIALLY IS NEEDED FOR ADDRESS-ERROR
'END' CHECKS;
'IF' PLOTTER 'THEN'
'BEGIN' PLOTRELATIVE ('TRUE', 0, 0, 0, 0)
'PLOTFRAME (0, 0, 15000, 2750, 15000, 2750)
'END'
'ELSE'
'BEGIN'
'FOR' LINENUMBER := 1 'STEP' 1 'UNTIL' LINENUMBER MAX 'DO'
'FOR' PRINTPOSITION := 1 'STEP' 1 'UNTIL' 144
1976 'DO' LINEPRINTEROUTPUT [LINENUMBER, PRINTPOSITION]=
1977 'FALSE';
1978 PRINTRELATIVE
1979 ('TRUE', 'TRUE', 1, 73, LINENUMBERMAX, 'FALSE')
1980 'END';
1981 'IF' ~ READ DTAPE = CUAK ~ READ DTAPE = CSTART
1982 'THEN' ERROR
1983 ('"PROGRAM DOES NOT BEGIN WITH UAK START", 'FALSE',
1984 'FALSE', 'FALSE');
1985 'FOR' S := READ DTAPE 'WHILE' 'TRUE'
1986 'DO' INTERPRET (S, 'FALSE', 'FALSE', PLOTTER)
1987 'END'
1988 'END'
APPENDIX B

1. Syntax of the Digiset machine code

The rules of the syntax listed below should be interpreted as follows:
- Non terminals are denoted in small letters, terminals begin with capital letters.
- The notation is in Backus normal form.

Meta characters are:

";" for "is defined as"
"," for "followed by",
"." for "end of rule",
"|" for "or".

- In each rule, the first occurrence of a non-terminal (except the one defined by that rule) is followed by the number of its defining rule between < and >.
- To emphasize the byte structure of the code, the following redundant notation is used:
  "byte ("should not be read as part of the syntax, but as a comment, that gives information about the byte-structure;")" stands for ",".
N.B. "byte" not followed by "("is the nonterminal for a number between 0 and 255 in binary notation.
- Each term in capitals followed by a ";" stands for a unique value in the 8-bit code of the Digiset.

2. Syntax of the shorthand code

The differences between the shorthand code and the Digiset code are the following:
- The numerical values are given in decimal notation followed by a semi-colon; for example rule 27:
  pa 62: byte (0,0,1,1,1,1,1,0), should be read as pa 62: 62;.
- The definition of single characters is simplified. This is accomplished by dropping the byte structure and putting image lines between brackets. The divergence starts with rule 35. The alternative definitions are marked with an asterisk.
- Each microprogram is embraced by two brackets, in order to improve error recovery. This can be seen from the definition of rule 71 (see rule 71*).

3. Syntax of the Digiset code

1 program : start <2>, body <4>, stop <3>.
2 start : UAK;, START;.
3 stop : UAK;, STOP;.
4 body : inortx <5> | inortx, body.
5 inortx : input <7> | text <73> | neglect <6>.
6 neglect : UAK;, UBL;, "everything except start", start <2>.
7 input : segopt <8>, core <10> | disc <12> | segopt, disc to core <13>.
8 segopt : | seg <9>.
9 seg : UAK;, SEG;, byte <93>.
10 core : core elmt <11> | core elmt, core.
11 core elmt : typeface <26> | char <34> | NUL; | KEN;, byte <93>, byte, byte.
12 disc : uspe <14>, uple <17>, alf <25>, uspe, utre <15>, uatt <16>.
13 distocore: uspe <14>, upla <18>.
14 uspe : UAK;, SPE;.
15 utre : UAK;, TRE;.
16 uatt : UAT;, UAT;.
17 uple : UAK;, PLE;, discad <19>, uatt <16>.
18 upla : UAK;, PLA;, discad <19>, uatt <16>.
19 discad : byte (clyl <20>, k <22>, sekt <24>), byte (alforch <23>, Ø, Ø, cylv <21>).
20 cylv : BIT 4.
21 cylvh : BIT 5.
22 k : BIT.
23 alforch : BIT.
24 sekt : BIT 3.
25 alf : typeface <26>, characterset <33>.
27 pa 62 : byte (Ø, Ø, 1, 1, 1, 1, 1, 0).
28 tfword : zbyte <94>, zbyte, byte (Ø, tftype <29>).
29 tftype : grid <30>, seize <31>, angle <32>.
30 grid : BIT 2.
31 seize : BIT 3.
32 angle : BIT 3.
33 character-set : | char <34>, characterset.
34 char : micropr <40> | properchar <35>.
35 properchar : uspe <14>, pa <36>, chword <37>, linelist <43>, uatt <16>.
36 pa : byte <93>.
37 chword : segad <42>, byte (Ø, Ø, VS1 <38>, Ø, Ø, Ø, VS2 <39>).
38 VS1 : BIT.
39 VS2 : BIT.
40 micropr : uspe <14>, pa <36>, mpword <41>, comlist <71>, uatt <16>.
41 mpword : segad <42>, byte (Ø, Ø, Ø, Ø, Ø, 1, 0).
42 segad : byte (Ø, Ø, BIT 6), byte <93>.
43 linelist : | 1spack <44> | slpack <45>.
44 1spack : longpack <46> | longpack, slpack <45>.
45 slpack : shortpack <47> | shortpack, 1spack <44>.
46 longpack : long <48> | long, longpack.
47 shortpack : short <49> | short, shortpack.
48 long : lbll <53>, lo <51> | blwl <55>, lo <51>.
49 short : lbls <52>, sh <50> | blws <54>, sh.
50 sh : S <63> | S, sh.
51 lo : 1 <64> | 1, lo.
52 lbll : lb1 <56> + spack <58>.
53 lbll : lb1 <56> + 1pack <59>.
54 blws : blw <57> + spack <58>.
55 blwl : blw <57> + 1pack <59>.
56 lb1 + pack : byte (1, dir <60>, num 3 <61>, Ø, 1, pack) | byte (1, dir, Ø, Ø, Ø, 1, 1, pack), byte <93>. 
57 blw + pack: byte (1, num 4 <62>, \emptyset, \emptyset, pack) |
    byte (1, \emptyset, \emptyset, \emptyset, 1, pack), byte <93>.
58 spack : \emptyset.
59 lpack : 1.
60 dir : BIT.
61 num 3 : BIT 3.
63 s : swze <65> | swz <67>, s.
64 1 : lwze <66> | lwz <68>, 1.
65 swze : byte (\emptyset, BIT 7), byte (1, BIT 7).
66 lwze : ww <69>, byte (1,\emptyset, BIT 6) | ww, byte (1,1, BIT 6), byte <93>.
67 swz : byte (\emptyset, BIT 7), byte (\emptyset, BIT 7).
68 lwz : ww <69>, zz <70>.
69 ww : byte (\emptyset, \emptyset BIT 6) | byte (\emptyset, 1, BIT 6), byte <93>.
70 zz : byte (\emptyset, \emptyset, BIT 6) | byte (\emptyset, 1, BIT 6), byte <93>.
71 comlist : comlemlt <72> | comlemlt, comlist.
72 comlemlt : segopt <8>, disc to core <13> | setchar <75> |
    layout <78> | seg <9> | iter <77>.
73 text : texdat <74> | texdat, text.
74 texdat : setchar <75> | domic <76> | layout <78> | seg <9> | iter <77>.
75 setchar : pa <36>.
76 domic : pa <36>.
77 iter : UAK;, WID;, byte <93>, byte, domic | UAK;, WID;, byte <93>,
    byte, setchar.
78 layout : relsize <79> | unsquare <81> | italic <82> | position <84> |
    UAK;, SCHN; | UAK;, DØ;.
79 relsize : UAK;, gro <80>.
80 gro : GROa; | GROb; | GROC; | GROd; | GROe; | GROf; | GROg; |
    GROh; | GROI; | GROj; | GROk; | GROl; | GROM.
81 unsquare : UAK;, DAN;, byte <93>.
82 italic : UAK;, knor <83>.
83 knor : NOR; | KUR 1; | KUR 2; | KUR 3;.
84 position : horp <85> | verp <86> | mixp <88> | defp <92>.
85 horp  : ZWR; | UAK;, RUCK;, byte <93>, byte.
86 verp  : UAK;, VERA; | UAK;, VERO;, tilt | UAK;, VERU;, tilt.
87 tilt  : byte (Ø, Ø, Ø, Ø, BIT 4), byte <93>.
88 mixp  : UAK;, ZER;, shift <89> | UAK;, ZEV;, shift.
89 shift : byte (m <9Ø>, sv <91>), byte <93>.
90 m     : BIT 3.
91 sv    : BIT 5.
92 defp  : UAK;, ZWQ;, byte <93>, byte | UAK;, TAB;, byte, byte.
93 byte  : BIT 8
94 zbyte : byte (Ø, Ø, Ø, Ø, Ø, Ø, Ø).

35* properchar  : uspe <14>, pa <36>, chword <37*>, linelist <43*>, uatt <16>.
37* chword      : addressvalue <94*>, byte (Ø, Ø, vsl <38>, Ø, Ø, Ø, vs2 <39>).
43* linelist    : ; | image line <95*>, linelist.
71* comlist     : [, actual comlist <96>, ].
94* addressvalue : undefined <97*> | segad <42>.
95* imageline   : empty line <98*> | nonempty line <99*>.
96* actual comlist: comlemt <72> | comlemt, actual comlist.
97* undefined   : 256.
98* emptyline   : (, <signed integer>),.
99* nonemptyline : (, <unsigned integer>, wzpack <100*>),.
100* wzpack     : wz <101*> | wz, wzpack.
101* wz         : white <102*>, black <103>.
102* white      : <unsigned integer>,
103* black       : <unsigned integer>. 
APPENDIX C

1. An ALGOL 60 program for the generation of a character set for line drawings (fig. 3.4.2).
2. Main program for the generation of fig. 3.4.5.
BEGIN COMMENT 2665F, PENHAGEN, HNOOT, GENEREREN VAN L'jNALFABET;
    INT M, V, PUT, PA, WD, ALFB;
    REAL VD, HD, D, STEP, FRAC, VFAC, HFAC;
    ARRAY ALFBA[1:256, 1:3];
PROC PRCSYM(S) BEGIN PRSYM(S); IF PUT=1 THEN CSYM(S) END;
PROC AFS : XT(C(N, M, V)); VAL V; REAL V;
BEGIN AFS : XT(N, M, V); IF PUT=1 THEN AFS : XT(C(N, M, V)) END;
PROC F : XT(C(N, M, V)); VAL V; REAL V;
BEGIN F : XT(N, M, V); IF PUT=1 THEN F : XT(C(N, M, V)) END;
PROC PRACTEXT(S); BEGIN PRACTEXT(S); IF PUT=1 THEN CTEXT(S) END;
PROC CTEXT(S); STRING S;
BEGIN INT I, SYM, K = 0;
FOR SYM = STRING'SYMBOL(K, S) WHILE SYM 'E 255 DO
BEGIN CSYM(SYM); K = K + 1
END;
END;
INT PROC: NEXTPA; NEXTPA = PA + IF PA 'E 0 then PA + 1 ELSE PA + 128 THEN 2 ELSE 1
PA + IF PA = 27 THEN PA + 61 ELSE PA + 123;
PROC GENTYPEFONT(CELLTYPE); STRING CELTYPE;
BEGIN INT I = 0;
REAL INC, LB, UB, ANGLE, PI;
CELL = STRING'SYMBOL(0, CELTYPE);
PRACTEXT("UAK; SPE; 62;");
IF CELL = 1 THEN BEGIN H = 400; V = 960; PRACTEXT("12") END ELSE
IF CELL = 12 THEN BEGIN H = 400; V = 480; PRACTEXT("44") END ELSE
IF CELL = 13 THEN BEGIN H = 200; V = 480; PRACTEXT("76") END ELSE
BEGIN NCR; PRINTTEXT("GRID TYPE ERROR"); EXIT END;
ALFB = 256;
FAC = HFAC;
PRACTEXT("O; OUA; UAT;");
PI = 3.14159265358979; INC = PI / 60; UB = PI / 4 + 4 - 4
FOR ANGLE = 0 STEP INC UNTIL UB DO GEN LINE CHAR PLUS MIC(ANGLE, 1);
LB = UB - 4; UPB = PI / 4 + 4
FOR ANGLE = LB STEP INC UNTIL UPB DO GEN LINE CHAR PLUS MIC(ANGLE, 2);
INC = -1 * INC; LB = UB - 4; UPB = UB - 4
FOR ANGLE = LB STEP INC UNTIL UPB DO GEN LINE CHAR PLUS MIC(ANGLE, 3);
UPB = PI / 4; LB = UB = 4
FOR ANGLE = LB STEP INC UNTIL UPB DO GEN LINE CHAR PLUS MIC(ANGLE, 4);
ALFB = PA;
END;
PROC GEN LINE CHAR PLUS MIC(ANGLE, TYPE); VAL ANGLE, TYPE, REAL ANGLE;
INT TYPE;
BEGIN REAL TGA, COSA, SIN, B, P1, P2, I, DH, DV;
INT DSPL, DSPL, DSPL2;
SIN = SIN (ANGLE); COSA = COS(ANGLE); TGA = SIN / COSA
IF TYPE = 1 THEN
BEGIN VOI = TGA; M = I; I = I + 1; STEP;
DV = 0 / COSA;
GEN CHAR HEAD(TYPE);
BEGIN PRCSYM(119); PRACTEXT("UAK;SPE:")); ABSF XTC(3,0,NEXTPA); PRACTEXT(";")); ABSF XTC(3,0,IF TYPE=1 THEN 0 ELSE 1); IF TYPE=2 THEN ELSE 32); PRCSYM(91); ID=0; *END*; PROC I NALFTABLE(A,H,V); VAL A,H,V; REAL A,H,V; BEGIN ALFTAB[PA,1]=A*150/3.14159265; ALFTAB[PA,2]=H*.72; ALFTAB[PA,3]=V*.72; *END*; PROC DUMP ALFTAB; BEGIN INT L; U=PA; PA=-1; NLCRTXT(" PA ANGLE HD UD"); FOR L=1, L+1 WHILE L<LE 66 DO; BEGIN NLCTXT(XT(3,0,L)); ABSF XTC(3,2,ALFTAB[L,1]); ABSF XTC(3,2,ALFTAB[L,2]); ABSF XTC(3,2,ALFTAB[L,3]); *END*; *END*; PROC BYTE(X); VAL X; INT X; BEGIN ABSF XTC(3,0,X); PRCSYM(91); *END*; PROC BYTE(X); VAL X; REAL X; BEGIN INTEGER X1,X2; X1=ENTIER(X/256); X2=X-X1*256; IF X2<256 THEN BEGIN X1=X1+1; X2=0; *END*; BYTE(X2); BYTE(X1); *END*; PROC BCCH(S,V); VAL V; STRING S; REAL V; BEGIN IF PRINTPOS() THEN 59 ELSE 123 THEN PRCSYM(119); PRACTXT(S); BBYTE(V); *END*; PROC GENW D(N,PA); INT N,PA; IF N>0 THEN BEGIN IF N>1 THEN BCCH("UAK;WID;",N-1); BYTE(PA); *END*; PROC GEN WGR(X); VAL X; REAL X; IF X<.01 THEN BEGIN BBCH("UAK;ZWO;",X=.50); PRACTXT("ZWR;"); *END*; IF X<.01 THEN BBCH("UAK;RUCK;",X=.50); PROC GEN UD(X); VAL X; REAL X; IF X=0 THEN ELSE IF X>0 THEN BBCH("UAK;VERU;",X=32) ELSE BBCH("UAK;VERU;",X=32); PROC MOV (X,Y,P); VAL X,Y,P; REAL X,P; BEGIN GEN ZWOR(X); GEN UD(Y); *END*; PROC REPES T; BEGIN IF PRINTPOS() THEN 59 ELSE 123 THEN PRCSYM(119); PRACTXT("UAK;VERA;UAK;ZEV;0;0"); *END*; INT PROC NEWPA(X); VAL X; INT X; BEGIN PA=X; NEWPA=NEXTPA; *END*; PUT=0;
\begin{verbatim}
176  FRAC := .225;
177  VFRAC := 1024 * FRAC;
178  D := .1;
179  PA := -1;
180  GEN TYPE FOUNT("C");
181  JUMP ALF TAB;
182  'END'
\end{verbatim}
BEGIN COMMENT EXERCISE 4;

INTEGER PX, PY, I, J, T, TB, AXIS; REAL ARRAY HD, WD[1:66];

REAL DX, DY;

INTEGER PROCEDURE REMAINDER(A, B); INTEGER A, B;

REMAINDER: = A-B*ENTIER(A/B);

INTEGER PROCEDURE NEXT AXIS;

NEXT AXIS := IF AXIS = 65 THEN 1 ELSE 1;

I:

IF AXIS = 15 THEN AXIS := 27 ELSE AXIS := 48 THEN AXIS := 61 THEN AXIS := 2

ELSE AXIS := 1

END;

PUT := 1;

PROTEXT("UAK;START;UAK;SEG1;UAK;GROA;UAK;TAB1;0;0;UAK;VERA;");

FORMANCE("UAK;ZEV", 450*32);

FOR I := 1 STEP 1 UNTIL 66 DO

BEGIN HD[1] := ALFTAB[1,2];

VD[1] := IF < 3 THEN ALFTAB[1,3] ELSE ALFTAB[1,3];

END;

PX := 34; PY := 50;

FOR T := 1 STEP 1 UNTIL 46 DO

BEGIN

IF REMAINDER(T, 5) = 1 THEN

BEGIN TB := 72; BFORMANCE("UAK;ZEV", 32*144) END ELSE TB := TB + 144;

BBFORMANCE("UAK;TAB", 50*TB);

DX := 1.25*HD(PX); DY := 1.25*VD(PX);

FOR I := 0, 2, 3, 4 DO BEGIN PLACE; MOV(DX*I, DY*I); GEN VID(5, PY + 127) END;

DX := 1.25*HD(PY); DY := 1.25*VD(PY);

FOR I := 0, 1, 2, 3, 4 DO BEGIN PLACE; MOV(DX*I, DY*I); GEN VID(5, PX + 127) END;

AXIS := PX; PX := NEXT AXIS; AXIS := PY; PY := NEXT AXIS

END;

END;

END