COMMUNICATION WITH AN AUTOMATIC COMPUTER

ACADEMISCH PROEFSCHRIFT

TER VERKRIJGING VAN DE GRAAD VAN DOCTOR IN DE WIS- EN NATUURKUNDE, AAN DE UNIVERSITEIT VAN AMSTERDAM, OP GEZAG VAN DE RECTOR MAGNIFICUS DR. J. KOK, HOOGLERAAR IN DE FACULTEIT DER WIS- EN NATUURKUNDE, IN HET OPENBAAR TE VERDEDIGEN IN DE AULA DER UNIVERSITEIT OP WOENSDAG 28 OCTOBER 1959 DES NAMIDDAGS TE 3 UUR PRECIES

DOOR

EDSGER WYBE DIJKSTRA
GEBOREN TE ROTTERDAM

UITGEVERIJ EXCELSIOR - FRIJDASTRAAT 19 - RIJSWIJK (Z.H.)

Promotor: Prof. Dr Ir A. van Wijngaarden

CONTENTS

1 INTRODUCTION

In his work as programmer the author had the privilege of being closely associated with the development of the order code of the X1 computer. This machine, and its order code, was designed by Mr B.J. Loopstra and Mr C.S. Scholten, both of Amsterdam.

In the final selection of the facilities to be included in the machine, I played only a small part. Where technical considerations were not decisive I was left the choice between alternatives; in some minor points I was able to suggest modifications.

When the internal code was fixed it was my task to build up a suitable notation for the code. In my turn I derived benefit from their criticism and also from that of Mr A.W. Dek of the Hague.

This work is the main theme of this thesis. In it I present a rather one-sided picture of the X1 as I restrict myself solely to the order code. Those technical and economic considerations which played a role in the development of the X1 fall outside the scope of this thesis. Similarly we do not discuss problems that arise when the computer is incorporated in an organization or questions that present themselves in the construction of a library of subroutines, of interpretative and self-coding systems etc.

In comparison with all these problems a "microscopic" interest in the internal order code may seem somewhat futile. But in which ever way and how ever intensely a computer may affect an organization, no matter how complicated or large the programs may be, the programs will always consist of the molecules, which we call orders.

The author is greatly indebted to the "Mathematical Centre" for providing him with the opportunity of engaging in this work and particularly to Prof.Dr Ir A. van Wijngaarden, Head of the Computation Department, for his cooperation and continuous interest.

2 A GENERAL DESCRIPTION OF THE X1

2.1 Introduction

The X1 is a high speed electronic computer, intended for both clerical and scientific work. It consists of three main parts, the basic machine, the memory and the input and output apparatus.

In principle the basic machine is always the same; it comprises the arithmetic unit, the control, the console, etc.

Up to a certain limit the size of the memory may be chosen for every installation; in this way it can be adapted to suit the needs of the user.

Similarly one can choose, within a certain range, which input and output apparatus are to be coupled to the machine.

The fact that the X1 is very suitable for both scientific and clerical work is due, not only to its flexibility, but also to its relatively high speed. Amongst others the speed of the machine manifests itself in administrative work when the X1 uses a number of punched card machines to their maximum capacity simultaneously.

2.2 A short description of the three main parts

2.2.1 The basic machine

The basic machine comprises the arithmetic unit, the various registers, the control, etc. Its external appearance is similar to that of an office desk; the various switches and indicator lights for the operator are on the surface of the desk and also on a small vertical panel behind this section.

The basic machine is fully transistorized and the power it requires is only a few hundred watt. As a result no cooling apparatus is necessary.

The arithmetic unit operates internally on fixed point binary numbers. All transports, as well as addition and subtraction, are parallel operations. By these and other means the designers attained a time of considerably less then 100 μs for all operations, except those of multiplication and division, which each require 500 μs.

In the X1 the "natural" unit of information, the word, consists of 27 bits; words have two main interpretations, viz. numbers and orders.

A word, regarded as a number, consists of a sign bit, followed by 26 binary digits.

Alternatively, a word, regarded as an order, defines an operation to be carried out by the machine. For this purpose the 27 bits are divided into two groups: the most-significant twelve (the "function") define the nature of the operation according to fixed conventions, the least significant fifteen (the "address" or "numerical part") may serve for the specification of an operand.

The arithmetic unit comprises three registers, called A, S and B. The registers A and S both have 27 bits, while B has only 16 bits (of which the most significant one acts as sign bit).

The B-register, being shorter than A and S, cannot store numbers as large as the other two registers; otherwise the three registers are equivalent with respect to additive operations: the X1 can add and subtract in all three registers, their contents can be added to or subtracted from the numbers stored in the memory, etc.

Multiplication and division are always executed in the A- and S-registers together; each then has a distinct function.

The distinctive feature of the B-register is the following: as soon as an order is extracted from the memory (see below) but before its execution, its address may be increased by the contents of B. The function of the order itself indicates whether or not this addition is to take place. The length of the B-register was chosen to be 16 bits, so as to accommodate an address increment which can have both signs.

Two other registers of some importance to the programmer are the order register OR and the order counter T.

Before an order is executed it is placed in the register OR, where the bits that form the order are analysed; these bits remain in OR throughout the execution of the order. As one requires a whole word for the definition of an order, OR consists of 27 bits.

Orders placed in OR, i.e. orders to be executed, come from the memory. The words in the store are numbered and the order counter T always contains the number of the word to be transported to OR as the next order. The order counter T is a 15-bit register.

Furthermore we mention three small registers, each of which has a capacity of only one bit; they record "the condition", "the last sign" and "the overflow indication".

They will be discussed in more detail later.

Finally the machine is equipped with a number of flip-flops, that play an important role in the synchronization between the X1 on the one hand and the coupled communication apparatus on the other.

The contents of all the above registers can be read from indication lamps on the control panel.

2.2.2 The memory

The maximum number of words the memory of the X1 can accommodate is $2^{15} = 32768$. These words are numbered from 0 to 32767 inclusive, the numbers being called the addresses of the storage locations, as they are used for reference to a particular word.

Information is sent from the arithmetical unit to the memory in units of one word. Sending a word to the memory is called writing a word in the memory, extracting a word from the memory is called reading a word out of the memory. For these operations to take place it is essential that the address of the storage location concerned be specified.

When a word is written in a storage location its previous contents are destroyed; in a "pure" reading operation the contents of the storage location remain unaltered. Finally the order code of the X1 includes combined operations, in which, firstly, a word is read from a storage location and, secondly, a new word is written in that same location immediately afterwards; in this case the new word is always derived from the original one by means of an additive process.

The memory of the X1 is divided into two parts, viz. the "living memory" and the "dead memory". Both make use of ferrite cores and have the same access time for reading operations.

The machine can write in the living memory, but not in the dead memory. In the living memory the information is recorded by the magnetic states of the ferrite cores, in the dead memory, however, by permanent wiring. The dead memory has the advantage of being relatively inexpensive; this makes it particularly suitable for the storing of standard programs, especially those for the input and the output.

The order code of the X1 is a so-called single-address code, i.e. each order refers, at most, to one word in the memory: the storage location for the word concerned is specified by the address digits of the order. The maximum capacity mentioned earlier ($2^{15} = 32768$ words) refers to living

and dead memory combined. This limitation is due to the fact that in the order 15 bits are available for the address.

Every time the memory delivers a word of information, this operation is subject to a so-called parity check. In all writing operations a 28th bit is added to the word; the value of this bit is such that an odd number of ones is stored in the location. Every time a word is read the memory delivers the corresponding parity bit too; the computer then checks whether the number of ones actually is odd. If not, it stops. Words in the dead memory are also provided with a parity bit.

2.2.3 Input and output apparatus

The input and output apparatus coupled to the X1 depend on the needs of the user.

In this thesis we restrict ourselves to a tape reader (150 characters per second) and a tape punch (25 characters per second) both mechanisms being for five-hole punched tape, and an electric typewriter operated by the machine (10 characters per second).

At the moment of writing many other possibilities for faster input and output of data have been wholly or partly developed, e.g. fast punched tapes, magnetic tapes, fast printers and punched card apparatus of all kinds.

The X1 can utilize a large number of such varied mechanisms simultaneously. The maximum number depends on the complexity of the processing: by the time that the operational speed of the X1 becomes the bottleneck of the process, it is of little use coupling the machine to further apparatus!

The X1 only utilizes all its communication mechanisms simultaneously from a macroscopic point of view; from a microscopic viewpoint it employs only one at a time. The X1 automatically divides its attention between the various mechanisms as efficiently as possible. Due to this arrangement the fact that the various mechanisms are completely asynchronous with respect to each other presents no essential difficulty. This is one of the most important aspects of the flexibility of the X1.

The standard programs for tape reading, tape punching and typing will fully illustrate which possibilities, which problems and what types of arrangements result from this facility of the X1.

2.3 The order code

As mentioned earlier (2.2.1) the bits of the binary representation of an order can be split up into "function" and "address" (also referred to as "numerical part"). In principle, the programmer writes each order as function followed by address in accordance with the above subdivision.

In the ensuing description of the order code (a summary of the existing functions!) we will denote the address by the letter n for the time being.

The symbol (n) denotes the contents of address n; similarly (A) denotes the contents of the A-register, etc.

In the description of the operations we will make use of the orientated equality sign "⇒" to be read as "replaces", cf [5].

The following eight orders exist for each of the three arithmetic registers (A, S or B being denoted by R).

0R n	:	(R) + (n) ⇒ (R)
1R n	:	(R) - (n) ⇒ (R)
2R n	:	+ (n) ⇒ (R)
3R n	:	- (n) ⇒ (R)
4R n	:	(n) + (R) ⇒ (n)
5R n	:	(n) - (R) ⇒ (n)
6R n	:	+ (R) ⇒ (n)
7R n	:	- (R) ⇒ (n)

0R and 1R: Addition in. The number in the storage location with address n is added, with or without a change of sign, to the number in the register R; the result is stored in the register. The original contents of the register are therefore lost, whereas the contents of the storage location n remain unchanged.

2R and 3R: Transport in. The number in the storage location n is transported, with or without a change of sign, to the register R. As in the case of "addition in", the original contents of R are lost while the number in the memory is not altered.

4R and 5R: Addition out. As for 0R and 1R respectively, with the roles of register and storage location interchanged.

6R and 7R: Transport out. As for 2R and 3R respectively, with the roles of register and storage location interchanged.

Remark. The classification "in or out" is based upon the direction of the transport of the result with respect to the arithmetic unit.

It is clear that a number in a register or storage location may have either sign, may be multiplied by -1, etc. We therefore include a more detailed description of the number representation; for this purpose we number the consecutive binary digits of a word as follows:

$$d_{26}\ d_{25} \ldots d_1\ d_0 \ .$$

The numerical value of a word, regarded as an integer, is given by

$$\sum_{i=0}^{25} (d_i - d_{26}).2^i \quad .$$

The most significant digit d_{26} is the so-called sign digit; $d_{26} = 0$ indicates that the number is positive. The largest possible positive number that can be represented in one word equals $2^{26} - 1 = 67108863$, in binary digits: $d_{26} = 0$, $d_{25} = d_{24} = \ldots = d_1 = d_0 = 1$. To change the sign of a number, all digits of the word are inverted, i.e. zeros are replaced by ones and ones by zeros. This method of representing negative numbers is often referred to as the one-complement or inverse system, cf [4].

The inverse system implies that the number zero may be represented in two ways, viz. +0 (all digits = 0) and -0 (all digits = 1). When a result equal to zero is formed by the X1, it is usually shown as -0, or, more precisely, the adder only has an output = +0 if both addenda are = +0. The X1 performs a subtraction by adding the inverted subtrahend.

The following logical operations exist for the registers A and S (R may represent either A or S):

0LR n : $+(n) \vee (R) \Rightarrow (R)$

1LR n : $-(n) \vee (R) \Rightarrow (R)$

2LR n : $+(n) \wedge (R) \Rightarrow (R)$

3LR n : $-(n) \wedge (R) \Rightarrow (R)$

Remark. Logical operations exist for the A- and S-registers only, not for the B-register; furthermore they are always "in-operations".

0LR and 1LR: Logical addition. The operations 0LA and 0LS form a result from the contents of the storage location n and the register in question; this result has a 1 in all those binary positions, where (n) and (R) differ. In the remaining binary positions a 0 is formed. This result replaces the original contents of the register, which are therefore lost; the number in address n remains unchanged. The instructions 1LA and 1LS are similar, except that they handle the inverted word -(n) instead of +(n). Logical addition is also called "carryless addition".

2LR and 3LR: Logical multiplication. The operations 2LA and 2LS form a result from the contents of the storage location n and the register in question; this result has a 1 in all those binary positions where both (n) and (R) have a 1. The remaining digits of the result are 0. Again the result is placed in the register, etc. The instructions 3LA and 3LS handle the word -(n) instead of +(n). Another name for logical multiplication is "collation".

The order code includes four versions of multiplication:

$$0X\ n : \quad [A] + [n].[S] \Rightarrow [AS]$$
$$1X\ n : \quad [A] - [n].[S] \Rightarrow [AS]$$
$$2X\ n : \quad + [n].[S] \Rightarrow [AS]$$
$$3X\ n : \quad - [n].[S] \Rightarrow [AS]$$

The position of the -binary- point (i.e. of the units) hardly plays a role in addition, provided that the point is in the same place in both numbers. This is not the case in multiplication and division: we therefore make use of square brackets [ ] to indicate explicitly that the word concerned must be regarded as an integer.

The symbol [AS] is used to denote a double length integer; the 26 most significant digits and the sign digit, which precedes them, are stored in the A-register; the S-register contains a copy of the sign digit and the 26 least significant digits. The symbol [AS] can only be used when sgn(A) = sgn(S); its numerical value is given by

$$[AS] = [A].2^{26} + [S] \quad .$$

The X1 multiplies the contents of S by (plus or minus) the contents of storage location n. In the so-called

"additive multiplications" 0X and 1X this product is increased by the original contents of the A-register, the product being increased at the least significant side! In the "clear multiplications" 2X and 3X there is no such increment. As implied by the notation, the most significant half of the result is placed in A, the other half in S; in the final result the signs of A and S are always equal to each other. The original contents of the registers A and S are lost, the number in the memory remains unchanged.

When the result of a clear multiplications is zero, the same zero is placed in both registers, the sign being determined by the signs of the factors according to the usual algebraic rules. The result of an additive multiplication will only be +0, if, firstly, the original (A) = +0 and, secondly, the original (S) and (n) would have had the corresponding clear product +0.

Another common interpretation of a word is as a proper fraction with the binary point between d_{26} and d_{25}. The word thus represents a fraction which is less than one in absolute value.

This interpretation of a word is denoted by braces { } . The following obviously holds:

$$\{n\} = [n] . 2^{-26}$$

Similarly we introduce two further interpretations for the double length number (AS), viz. as compound number

$$[AS\} = [AS] . 2^{-26} = [A] + \{S\} \quad ,$$

i.e. with the point between the registers, or as double length fraction

$$\{AS\} = [AS] . 2^{-52} = \{A\} + \{S\} . 2^{-26} \quad ,$$

i.e. with the point preceding the most significant digit.

In illustration, the order 2X n was defined as

$$+ [n] . [S] \Rightarrow [AS] \quad ;$$

by multiplying both sides of this equality by 2^{-26}, we obtain

$$+ [n] . \{S\} = + \{n\} . [S] \Rightarrow [AS\}$$

and after multiplication by 2^{-52} we obtain

$$+\{n\} . \{S\} \Rightarrow \{AS\} \quad .$$

A programmed rounding-off for multiplication is described at the end of paragraph 2.6.1.

The order code includes four versions of division, viz.

0D n	:	[AS] /+ [n] ,	remainder ⇒ [A] , quotient ⇒ [S]
1D n	:	[AS] /- [n]	" "
2D n	:	$[A]\,2^{26}$/+ [n]	" "
3D n	:	$[A]\,2^{26}$/- [n]	" "

The operations 2D and 3D begin by clearing S under control of the sign digit of the A-register. From then onwards their operation is the same as that of 0D and 1D respectively, which may only be executed when sgn(A) = sgn(S). (This restriction is implied by the use of the symbol [AS] !)

The double length number [AS] is then divided by +[n] . The quotient appears in S, the remainder is left in A. By definition, this remainder has the same sign as the dividend and the smallest absolute value then possible. The original contents of the registers A and S naturally disappear, while the number in the memory remains unchanged.

Neglecting the remainder, a more useful definition of the division 2D n is obtained after multiplying by 2^{-26}:

$$[A] : [n] = \{A\} : \{n\} \Rightarrow \{S\} \quad ;$$

the quotient is then not rounded-off. A programmed rounding-off for division is described at the end of paragraph 2.6.1.

A necessary condition for a correct result after division is that the quotient does not exceed the capacity of one word.

The above description is incomplete in one respect; every operation includes the substitution

$$(T) + 1 \Rightarrow (T) \quad ,$$

i.e. one is added to the contents of the order counter T at the beginning of the execution of every order. As mentioned before (see 2.2.1) (T) is the address of the order to be executed. Hence the instructions are obeyed in the order in which they are stored in the memory.

The above only holds if we restrict ourselves to the orders that have been dealt with so far. Besides these, there is a special group of orders that may cause the machine to start selecting orders at a different point in the memory. They are called jump orders; from a technical point of view their function is to give (T) a new value.

0T n : $(T) + (n) \Rightarrow (T)$
1T n : $(T) - (n) \Rightarrow (T)$
2T n : $+ (n) \Rightarrow (T)$
} and stop the X1 if $(n) \leqslant -0$

4T n m : $(m) - 1 \Rightarrow (m)$; $n \Rightarrow (T)$; $0 \leqslant m \leqslant 7$

6T n m : $(T) \Rightarrow (m+8)$; $n \Rightarrow (T)$; $0 \leqslant m \leqslant 15$

Here the symbol (T) represents the contents of the order counter after the increase

$$(T) + 1 \Rightarrow (T)$$

has been effected. This addition always takes place, regardless whether the current order is a jump or not.

0T and 1T: Additive jump. When (n) =+0, both orders have no effect. They act as skips, i.e. the control proceeds to the next instruction without altering the contents of the memory or of the registers. When (n) = +1, the order 1T n acts as a dynamic stop.

2T: Normal jump. The substitution $(n) \Rightarrow (T)$ applies to the 15 least significant digits of the word (n); they are copied into the order counter. The jump has then been executed and the next order will be read from the storage location indicated by (n).

After execution of the 0,1,2 T jump the X1 stops if $(n) \leqslant -0$. This makes it possible for the X1 to execute a subroutine at full speed during the testing of a program and to stop automatically when control returns to the main program (see 2.8).

4T: Counting jump. In this case the address must be followed by an index m which can have any value from 0 to 7 inclusive. The 4T-order is a jump, since its address is copied into the order counter. (NB. The actual address digits and not the contents of address n are copied into the order counter!)

Furthermore the number in storage location m is decreased by one. In this context that location is usually referred to as r_m and the counting is described by

$$(r_m) - 1 \Rightarrow (r_m) \quad .$$

The full meaning of the counting jumps will become clear as soon as we are familiar with a number of additional facilities included in the order code; in the discussion of the so-called variants (see 2.4) we return to this subject.

6T: Subroutine jump. The subroutine jump is also provided with an index m; here m satisfies the inequalities $0 \leqslant m \leqslant 15$. As in the case of the counting jump the address n, and not (n), is copied into T. Before this takes place, however, the contents of T, already increased by one, are copied into storage location m + 8. The location m + 8 is usually referred to as s_m in this context. (The sixteen (s_m)'s therefore occupy the addresses 8 to 23 inclusive.) The previous value of (s_m) is obviously lost.

The word (s_m) - also called "the link" - records the address of the order that follows the subroutine jump in the memory; this order would have been the next order to be executed, were it not for the fact that the 6T-order, being a jump, interrupted the normal sequence. The value of (s_m) indicates where the interrupted sequence can be continued; for this reason the 6T-order is called a subroutine jump.

The subroutine jump therefore "preserves" the current contents of T by putting them in safety in the 15 least significant digits of (s_m). The subroutine jump preserves still more information: the remaining bits of (s_m) are used to record the current contents of a number of small registers. The details are postponed until these registers have been discussed (see 2.4.3 and 2.7.2).

The remaining orders form a special group, known as "shift and communication orders". They are of a different nature and will be discussed later (see 2.6 and 2.7).

2.4 The variants

The description of the orders as given in 2.3 holds in so-called "normal cases". In actual fact there are different versions of the orders. The function of the orders can be altered and extended in three respects. These modifications are controlled by the three so-called "variants". Roughly speaking, the variants cover "address modification", "condition-setting", and "condition reaction".

2.4.1 Address modification

The variant controlling address modification can take one of four forms. In the normal case the address digits of the order are processed as described in 2.3, the other three possibilities are indicated by writing one of the three letters A, B or C behind the address.

A("Absolute") In this case the address modification is a change in interpretation of the address digits: the numerical part n of the order is not interpreted as an address here, but as a number. For this purpose 12 zero's are added to the 15 digits of n at the most significant side.

For the absolute version "the contents of storage location n" in the above description of the orders should be read as "the number n".

For example, the normal order 2S n has the function $(n) \Rightarrow (S)$; by contrast, the order 2S n A has the function $n \Rightarrow [S]$.

The absolute version makes it possible to record constants less than 2^{15} directly in the address of the order; in this way space in the memory is saved.

Furthermore the absolute version is, as a rule, faster than the normal version: when the order has been selected, no second memory contact is necessary to obtain the number.

We note:

1. that the absolute version has no meaning in the case of out-operations.
2. that for the orders 4T and 6T (counting and subroutine jump respectively) the address is automatically interpreted as an absolute address (see 2.3). The variant for address modification does not apply to these orders, the address may not be followed by either A or B or C. The obligatory index m follows it instead.

3. that the absolute version of the 2T-jump will be made use of more frequently than the normal version.

B and C ("B- and C-correction" respectively) In the case of a B- or C-correction, the contents of the B-register are added to the address of the order immediately after it has been read from the store. The order is then executed with the modified address; the latter is interpreted in the normal way and never as an absolute address.

If (B) is negative care must be taken that the B- or C-correction gives rise to a non-negative address; similarly, if (B) is positive, the resulting address must be less than 2^{15}.

The difference between the B- and C-correction is the following: the B-correction leaves the order stored in the memory intact, while the C-correction also substitutes the newly formed address in the memory. For orders stored in the dead memory a C-correction is interpreted as a B-correction.

Finally a word of caution must be added with regard to shift and communication orders. Here the use of B- or C-correction may give rise to unexpected results, because the address digits act more or less as additional function digits. By using a B-correction the character of such an order can radically change.

2.4.2 Condition reaction

The variant of the condition reaction can take one of four forms. In the normal case the order is executed as described above (see 2.3). To indicate one of the three remaining possibilities, one of the letters U, Y or N is placed before the function. The name of this variant is derived from the last two versions (Y and N), as they cause the order to react to the so-called condition. The latter is recorded in a separate one bit register. When this bit is zero, we say that "the condition is affirmative", if not, we say that "the condition is negative".

Y ("Yes-conditional") In this case the order is only executed of the condition is affirmative, otherwise it is skipped.

N("No-conditional") In this case the order is only executed if the condition is negative, otherwise it is skipped.

We point out again that a skipped order leaves the contents of the registers and the memory unaltered: if, for

example, a C-corrected order is skipped on account of the condition, the order itself also remains unchanged in the memory.

U ("Undisturbed destination"). When the U-version is used, transfer of the result of the operation to its final destination is suppressed, although it is formed. The result of the operation is thus lost, whereas the original data remain.

This version would obviously be pointless, were it not for the fact that the result obtained can be used to set the condition (see 2.4.3).

The U-version does not apply to multiplication, division and shift and normalize orders (see 2.6), i.e. not to those instructions in which the registers are used repeatedly in forming the result.

The U-version has a different meaning for T-orders (jumps), viz. "Execute the jump order if the overflow indication = 1 and clear it (i.e. make it = 0), otherwise skip".

The overflow indication is held in a separate one bit register (comparable to the condition). Irrespective of its previous value, this bit is given the value one as soon as an overflow of capacity occurs in addition or subtraction.

Overflow is detected when the addition of two numbers with equal signs results in a sum with the opposite sign. This applies only to numbers of full word length, the orders 0B to 7B therefore require a more detailed description.

In the orders 0B, 1B, 4B, 5B, 6B and 7B the original contents of the B-register are supplemented by eleven copies of its sign digit at the most significant side; the 27 bit number thus obtained is processed.

In the orders 0B, 1B, 2B and 3B only the 16 least significant digits of the result are transferred to the B-register.

The overflow detection, however, reacts to the addition performed on numbers of full length. By addition of two positive numbers, it is therefore possible that the sign digit of the B-register d_{15} becomes = 1, without the overflow indication being set.

2.4.3 Condition-setting

In principle the last variant controls whether the order assigns a new value to the condition. The normal case, in which the bit specifying the condition is left as it stands, is described in 2.3. If we wish the result of an operation to influence the condition, one of the letters P, Z or E is written right at the end of the order.

We formulate the setting of the condition as recording the reply to a question; the nomenclature "affirmative" and "negative" for the two states of the condition is in accordance with the above convention.

P("Positive?") The condition records the reply to the question "Is the result $\geq +0$?", where, by definition, a result = -0 would lead to a negative answer.

Z("Zero?") The condition records the reply to the question "Is the result = 0?"

E("Equal signs?") The condition records the reply to the question "Is the sign of the result equal to the sign of the result of the previous condition-setting order?"

It is possible to construct the answer to the last question thanks to an additional memory element, the so-called "Last Sign Register"; like the registers for condition and overflow indication it consists of one bit.

In the orders 0B to 7B the questions asked refer to the full length result (see 2.4.3, last paragraphs). For multiplication and division we specify that the final contents of the A-register are to be taken as "the result".

The above description of this variant does not apply to jump orders; for them this variant acts as follows.

The symbols P, Z or E do not occur in the 6T-order (subroutine jump).

In the 4T-order (counting jump) P, Z or E may occur, but they do not change the condition. In the description of the counting jump (see 2.3) the counting: $(r_m) - 1 \Rightarrow (r_m)$ and the jump: $n \Rightarrow (T)$ were irrevocably connected. By placing a P, Z or E after the index m, however, one makes the execution of the jump dependent on the new value of (r_m).

P("Jump if (r_m) is positive") The jump only takes place if the new $(r_m) > 0$. (NB. When the counting leaves $(r_m) = 0$, it is always in the form $(r_m) = -0$.)

Z ("Jump if (r_m) equals zero") The jump only takes place if the new (r_m) = 0.

E ("Extra jump") The jump only takes place if the new value (r_m) $\geq$ 0. (If this order is executed repeatedly, one more jump is made than in the case of P, hence the name "Extra jump".)

A counting jump may be conditional to the overflow indication (U) or the condition (Y or N), in which case the counting jump is skipped if the requirements concerned are not fulfilled. This implies no counting and no jumping. If the order is not skipped, the counting always takes place and the order is considered as executed - in the U-version the overflow indication is cleared! -, even if, after counting, it is found that the jump must be suppressed.

The variant only exists in the P-version for the remaining jumps (0T, 1T and 2T). The jump is then called a "restoring jump". For the present a partial description must suffice.

The order "2T n" transfers the 15 least significant bits of (n) to the order counter. The order "2T n P" does this too, but performs a number of secondary substitutions depending on the values of the more significant digits of (n). For the moment we mention the processing of three of these bits.

(n) for the orders 0, 1, 2T n P

d_{18} = 0 make the condition affirmative
= 1 " " " negative

d_{17} = 0 make the "Last Sign" positive
= 1 " " " " negative

d_{16} = 0 place a 0 in the overflow indication (i.e. clear it)
= 1 place a 1 in the overflow indication.

The inverse of the above operations always takes place in subroutine jumps: besides the fact that (T) is preserved in the corresponding s_m (see 2.3), the condition, last sign and overflow indication are recorded in the digits d_{18}, d_{17} and d_{16} of (s_m). It is therefore possible to call in a subroutine, at the end of which the control is returned to the main program, with the restoration of the status quo at the moment of calling in, as regards the condition, last sign and overflow indication.

If the subroutine jump was conditional to the overflow (U-version of the jump order), an already cleared overflow indication is recorded in s_m.

2.5 The order notation

2.5.1 The address notation

In discussing the address notation we regard the memory as being divided into pages, each consisting of 32 consecutive storage locations.

An address is written in two parts, viz., line number followed by page name. For this reason the lines of a page are numbered from 0 to 31 inclusive. Line 31 of a page is followed by line 0 of the next one.

The standard program sheets for the X1 are designed in accordance with this convention and have 32 lines; starting at the top they are numbered from 0 to 31.

In their turn pages are grouped into so-called paragraphs. A page name consists of so-called paragraph letters followed by the page number. The latter may have any value from 0 to 16 inclusive, it numbers consecutive pages of the same paragraph. Thus a paragraph is characterized by its paragraph letters and it comprises, at most, 17 consecutive pages of 32 words each.

During input of orders the X1 deduces the binary representation of the address from these symbols. In this deduction the page number, multiplied by 32, is added to the line number; this sum is added to the address at which the paragraph starts, i.e. the address of line 0 of page 0 of the paragraph. Hence line 3 of a page may also be referred to as line 35 of the previous page.

The programmer may choose from 169 possible paragraph names; they all consist of two letters and he may use one of the following thirteen for both the first and the second letter:

Z, E, F, H, K, L, R, S, T, W, U, Y, N.

An address is written in three columns, from left to right:

1. the line number
2. the first paragraph letter
3. the second paragraph letter followed by the page number.

Remark. If a line number or a page number happens to be zero, one is nevertheless obliged to write this 0 down.

It is useful to regard those paragraphs with the same first paragraph letter as being the paragraphs of one chapter. If the first paragraph letter equals that of the previous address, it may be omitted. A special column is reserved for the first letter, so that this omission is clearly seen: a blank in this column acts as "dittos".

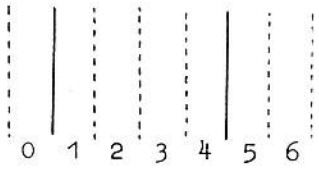
Thus it is only necessary to write down the first paragraph letter when it changes, and one saves much writing and punching when one avoids changes in this letter as much as possible. The idea of chapters now becomes clear: if those paragraphs, that refer to each other most frequently, are grouped under one chapter, we obtain a clearer and more concise notation. By contrast to pages in a paragraph, the paragraphs of a chapter need not directly follow each other in the memory.

The address at which each paragraph starts may be chosen by the programmer; this address functions as an arbitrary reference point, with respect to which the programmer can number his addresses.

There are, however, two fixed reference points, each of which is indicated by a single letter (viz. X and D respectively). The processing of these letters is independent of the chapter in which they occur; their use does not give rise to a change in the first paragraph letter in the sense described above. For this reason the symbols X and D are written in the last address column, together with the page number, that may have any value from 0 to 31 inclusive after these letters. The X refers to address 0, the beginning of the (living) memory; the D refers to a fixed point in the dead memory section.

2.5.2 The order

Orders are written in seven columns found on the standard program sheets; for the moment we number the columns from 0 to 6 as shown below. (At the extreme left of the columns we find the numbers 0 to 31 for the lines, at the right a fair amount of space is left for notes and explanations.)



From left to right they are reserved for:

column 0: Condition reaction (When it occurs, U, Y or N is entered in this column.)
column 1: Function, i.e. function digit followed by function letter(s)
column 2: Line number
column 3: First paragraph letter (usually omitted)
column 4: Second paragraph letter followed by page number
} address
column 5: Address modification (When it occurs, A, B or C is entered in this column. In the orders 4T and 6T, where these letters cannot be used, the index m $\leq$ 7 or $\leq$ 15 respectively is written in this column.)
column 6: Condition-setting (When it occurs, P, Z or E is entered in this column.)

In short: function and address are written between the thick lines, outside of these space is provided for the variants.

In many of the orders in the absolute version, the programmer will prefer to write the numerical part in decimals. For example, when one wishes to multiply (S) by 1000 the notation

2X 1000 X0 A

is clearer than

2X 8 X31 A

(although the last form is permissible). As a result A is often preceeded by "X0", as nothing is to be added to the decimal number. The combination "X0 A" occurs so often that a shorter notation has been introduced: in this particular case the X0 in column 4 may be omitted, and the usual notation for the said instruction is

2X 1000 A

2.6 Shift and communication orders

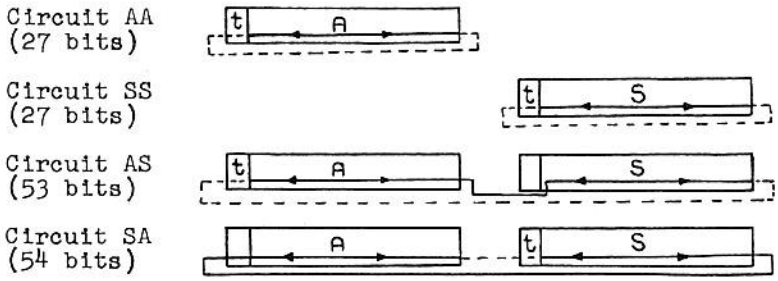
The order code includes a special group of instructions known as "shift and communication orders". The address digits of these orders never refer to a storage location, but hold additional information about the nature of the operation (according to a specific code).

They can all be denoted by the function letters Y and Z. For many of these operations, however, this would lead to a rather cumbersome notation: the programmer would have to be familiar with the coding of the address portion of the shift and communication orders. To avoid this, a special function letter P is introduced.

The function letter P is the indication that the tape reading program has to process the symbols that then follow in a special way, as they have an entirely different meaning from ordinary address symbols. The subdivision into pages does not apply here, these "addresses" never bring about a change in the first paragraph letter as described in 2.5.1. As a reminder the column in question is always left open for these orders.

2.6.1 Shift orders

Shifts can take place in four so-called "circuits". They are



The sign digit of the first register named acts as "the sign digit of the circuit", in the diagrams it is indicated by t. We note that the sign digit of S is not included in the circuit AS.

Digits of a circuit may be shifted in either direction, as shown by the arrows in the diagrams.

There are two different ways of shifting, "round shifting" and "clear shifting". Shifting around needs little further explanation: the digits of the circuit are cyclically shifted over a given number of places, either to the right, or to the left. In the clear shift the circuit is, as it were, cut open at the sign digit t of the circuit.

(The part of the circuit indicated by dotted lines in the diagrams then no longer applies.) The sign digit t does not change during the clear shift; the digits leaving the circuit at the cut are lost, at the other side of the cut copies of t are sent into the circuit.

The kind of shift is indicated in column 1 of the program sheet, i.e.

0P	Round shift to the left
1P	Round shift to the right
2P	Clear shift to the left
3P	Clear shift to the right

In column 2 we write n, the number of places over which the digits are to be shifted ($0 \leqslant n \leqslant 31$).

Column 3 is left open, in column 4 we write the two letters denoting the circuit.

The U-version does not exist for shift orders; neither does the A-version, because the interpretation of the address digits is already fixed. The B- or C- correction may be applied, but with due care: if the resulting n should for example become greater than 31, the order would become a different shift or communication order! The P-, Z- and E-versions all exist, but we must specify that the term "the result" refers to the final contents of the A-register for the circuits AA and AS, and those of the S-register for the circuits SS and SA.

In illustration, we give a rounded-off version of multiplication and division. The program below rounds the product {0 X1}.{1 X1} off and places the result in {A}:

	2S	0	X1		
	2X	1	X1		P
	0P	1	AS		
Y	0A	1		A	
N	1A	1		A	
	1P	1	AS		

The following program rounds the quotient (0 X1)/(1 X1) off and places it in {S}:

	2A	0	X1		P
	2S	1	X1		E
N	5P		SS		
	3P	1	SS		
	0D	1	X1		

$-(S) \Rightarrow (S)$, see 2.6.2.

The above examples show that rounding-off is a rather complicated matter when nothing is known about the signs of the numbers involved.

2.6.2 Register transports

The contents of any one register may be copied, with or without inversion, into another or the same register by means of a "register transport order". Columns 2 and 3 are left blank. The general form of these orders is

4P	RR^*	$(R) \Rightarrow (R^*)$
5P	RR^*	$-(R) \Rightarrow (R^*)$

where R and R^* represent A, S or B.

The A-version does not exist and B- and C-correction are difficult to apply. The variants of condition reaction and setting may be used without restriction.

2.6.3 Normalize orders

In normalize instructions columns 2 and 3 are also left blank. What has been said about the variants in 2.6.1 applies here as well, but B- and C-correction are of less importance.

Normalize orders are, in fact, shift orders to the left. The number of places shifted, however, is not given in the order; the digits in the circuit are shifted to the left until the most significant digit of the number differs from the sign bit t (in other words, over the maximum number of places for which a clear or round shift makes no difference). The normalize order places this number in the B-register.

Normalize orders exist only for the circuits AA, AS and SS (see 2.6.1). The maximum number of places shifted in normalizing is 26 for the circuits AA and SS, and 52 for the circuit AS. These numbers are only reached if all digits of the circuit are the same.

The normalize orders are written as follows:

6P	AA	normalize (A)
6P	SS	normalize (S)
6P	AS	normalize (AS)

(The above description of the order 6P AS is not completely correct, as the order may be given when sgn(A) ≠ sgn(S).)

2.6.4 The stop order

	7P					stop

The order 7P stops the X1; it exists in the Y-version and in the N-version.

2.6.5 The fast multiplications by 10

Two special multiplications are included to speed up the conversion from decimal to binary representation and vice versa.

	6Z	32	XX		10 [S] ⇒ [AS] (or 10 {S} ⇒ [AS])
	6Z	33	XX		10 [S] ⇒ [S]

In the first order the original contents of A are lost, in the second they remain intact. The first order is of special significance in the output, the second in the input, of data in decimal form. The two orders are technically the same but for the fact that transport of the "carry" to A is suppressed in the second order; the latter only gives a correct result if the product does not exceed the capacity of one word! In both orders the final contents of S act as the "result" for the purpose of condition-setting.

The time taken by both multiplications is 64 μs (normal multiplications require 500 μs).

The "actual" communication orders are of only small interest to most programmers, because in most cases the standard communication programs will meet their needs.

Since the communication programs for tape reader, tape punch and typewriter will be discussed in detail in this thesis, we now describe the communication orders that control these mechanisms.

2.6.6 The communication orders for the tape reader

The tape reader handles five-hole punched tape: the two most significant positions are on one side of the sprocket hole, the three least significant on the other side.

Tape read instructions regard the output of the tape reader as a positive number of 5 digits (a hole = 1 and no hole = 0); the number is supplemented by 22 zeros at the most significant side. This number is denoted below by (BL); it may be added to or subtracted from (A) or (S), or transported to one of these registers with or without change of sign. As soon as a symbol has been read by one of the following tape reading instructions, the tape is stepped up by one position, so that the next symbol can be read.

0Y 1 XP	(A) + (BL) ⇒ (A) and step up
1Y 1 XP	(A) - (BL) ⇒ (A) " " "
2Y 1 XP	+ (BL) ⇒ (A) " " "
3Y 1 XP	- (BL) ⇒ (A) " " "
0Z 1 XP	(S) + (BL) ⇒ (S) " " "
1Z 1 XP	(S) - (BL) ⇒ (S) " " "
2Z 1 XP	+ (BL) ⇒ (S) " " "
3Z 1 XP	- (BL) ⇒ (S) " " "

The variants of condition-setting and reaction may be used without restriction.

2.6.7 The communication orders for the tape punch

The five least significant digits of (A) or (S) can be punched with or without inversion. They are sent to the so-called type-punch-relays TP. As soon as a symbol has been punched, the tape is moved up over one position.

6Y 1 XP	+(A)⇒(TP), punch and step up
7Y 1 XP	-(A)⇒(TP), " " " "
6Z 1 XP	+(S)⇒(TP), " " " "
7Z 1 XP	-(S)⇒(TP), " " " "

In condition-setting the complete word ±(A) or ±(S) functions as the "result"; the variant of condition reaction may be used in the Y- and N-version, but not in the U-version.

2.6.8 The communication orders for the typewriter

The type order The operation of the type order is analogous to that of the punch order; in this case, however, the six least significant digits of ±(A) or ±(S) determine the symbol to be typed. The type-punch-relays TP are used here too.

6Y	2	XP	+(A) ⇒ (TP) and type
7Y	2	XP	-(A) ⇒ (TP) " "
6Z	2	XP	+(S) ⇒ (TP) " "
7Z	2	XP	-(S) ⇒ (TP) " "

The use of six bits makes it possible to operate all the keys of the typewriter. The "Capital Letter key" can also be used; the typewriter will remain in this position until a signal is sent to the "Small Letter key".

The correspondence between (TP) and the typed symbol is given in Appendix 1.

The "echo" orders The purpose of these orders is checking the output by means of the typewriter as fully as possible. The typed symbol can be sent back in binary form from the type relays TR to A or S. The number (TR) is supplemented by 21 zeros at the most significant side.

0Y	2	XP	(A) + (TR) ⇒ (A)
1Y	2	XP	(A) - (TR) ⇒ (A)
2Y	2	XP	+ (TR) ⇒ (A)
3Y	2	XP	- (TR) ⇒ (A)
0Z	2	XP	(S) + (TR) ⇒ (S)
1Z	2	XP	(S) - (TR) ⇒ (S)
2Z	2	XP	+ (TR) ⇒ (S)
3Z	2	XP	- (TR) ⇒ (S)

2.6.9 Timing of communication orders

As mentioned before the tape reader can handle 150 symbols per second, the tape punch 25 and the typewriter 10 (see 2.2.3). When the control encounters a tape reading, punching or typing order, one of two situations may arise.

It is possible that so much time has elapsed since the execution of the last similar order, that the apparatus

in question can deal with a new order immediately; the communication order is then executed in 36 μs and the X1 continues its program without delay.

On the other hand the apparatus in question may still be engaged in the completion of the previous similar order; in this case the X1 automatically waits until the new communication order can be executed and only continues the program after the necessary delay.

It is therefore possible to make a communication program for the X1, without paying any attention to the relative timing of the external apparatus on the one hand and the X1 on the other hand.

The X1 can continue with the program immediately after obeying a communication order, although the actual completion of this operation may require a considerable time (viz. 7, 40 or 100 ms); this is possible thanks to the fact that the external apparatus has a separate control.

The tape punch and typewriter make use of the same control apparatus, while the tape reader has its own. As a result, the blocking of tape read orders on the one hand is entirely independent of the blocking of punch, type and echo orders on the other hand.

When a tape read order has been obeyed, no new tape read order is accepted for 7 ms.

When a tape punch order has been obeyed, no new punch, type or echo order is accepted for 40 ms.

When a type order has been obeyed, no new punch, type or echo order is accepted for 100 ms. (If the type order sends a Tab-signal to the typewriter, the blocking lasts longer, after a NLCR-signal it lasts much longer.)

The execution of an echo order does not give rise to any blocking.

When the X1 encounters a communication order that cannot be accepted immediately, it automatically waits, as mentioned earlier, until the restriction is removed.

Since the X1 would be nonactive during this period, a system has been devised whereby it is possible to avoid waiting time; during the period of blocking the X1 continues with some other program but as soon as the restriction is removed, the X1 is "warned" that the following communication order can be executed without delay. This "warning" is called "an interruption signal".

2.7 The synchronization with external apparatus

2.7.1 Introduction

The communication orders for tape reader, punch and typewriter have been described in 2.6.6 to 2.6.9.

Besides these, the order code includes instructions for other input and output apparatus which may be coupled to the X1, e.g. fast tape punch, reproducers and fast sorters for punched cards, tabulators, fast printers, magnetic tape mechanisms. They will not be considered here; we prefer to direct our attention to a general problem that arises in connection with the coupling of the X1 to external apparatus, viz. the problem of synchronization.

The fact is that these orders (which will occur in a communication program) enable the X1 to make contact with such apparatus, but the construction of the latter will, as a rule, impose the restriction that the time of execution of such a communication program cannot be arbitrarily chosen.

For example, we have already seen that an interval of at least 100 ms must have passed after the execution of a type order, before the next one can be obeyed. For a punched card machine, which can deal with cards at fixed intervals, the "consciousness of time" is even more pronounced. In consequence, it is desirable (if not necessary) that the moment at which the X1 starts on the communication program, be determined by the external apparatus rather than by the X1.

For example, in the case of the typewriter, this is the moment at which another type order can be accepted, in the case of a reproducer this is the moment, at which a card has just passed the brushes, etc.

The so-called "interruption" has been built into the machine to enable the X1 to execute a specific communication program within a period of time, that is independent of the main program. Broadly speaking, the interruption involves the following: at the beginning of such an interval the program under execution is interrupted as soon as possible, a so-called "interruption program" attends to the external apparatus concerned, and afterwards the program which was interrupted is continued as if nothing had happened.

We can picture the interruption as follows: on completion of an order, the following order is not selected,

but a special subroutine jump is placed in the order register instead. As the inserted instruction is a subroutine jump, the current contents of the order counter and the condition etc. are preserved in the corresponding link (s_m) and control is transferred to the beginning of the interruption program. As a rule this will begin by storing the current contents of the registers A, S and B in locations specially reserved for the purpose. Thereafter the interruption program proper attends to the apparatus in question, and finally the status quo of the interrupted program is restored with the aid of the preserved data.

In other words: as soon as more urgent work has to be done, the X1 can be automatically "lent" to an interruption program so that it can be speedily attended to. By comparison to solutions without an interruption facility, the efficiency of the X1 is considerably increased in this way.

If the X1 has to attend to several external mechanisms, one can understand that it is desirable that different mechanisms produce their own interruption. These interruptions should have an order of priority. For example, a small interruption program, which must be executed within a short and specified range of time may not be interrupted by a long interruption program, which can probably be postponed longer than the short one; the reverse case, however, should be permitted.

For any interruption, the so-called "interruption permit" indicates whether the interruption is permitted or not. It can be modified by the program: interruptions therefore take place automatically, but only inasmuch as allowed by the program!

If one prevents an interruption by means of the interruption permit the actual interruption is postponed until the permit is suitably modified. To make this postponement possible mechanisms (or parts of these) which require attention do not bring about the interruption directly, but they only send a so-called "interruption signal" to the X1. (This signal remains until it is removed by the appropriate order in the corresponding interruption program.) The permit then determines when the signal will bring about its corresponding interruption.

Provision is made for seven different interruptions at most. It is possible for different signals to call for the same interruption. Those interruption signals that cause the same interruption form a so-called class. The interruption permit can specify per class, whether its signals are to have an immediate effect or not; in other

words the interruption permit can postpone certain interruptions. The classes are numbered from 1 to 7; interruptions from class 7 cannot be suppressed by the permit (see 2.7.4) and the interruption permit therefore consists of six bits.

As far as interruption signals are concerned the presence of one from its class is sufficient to produce an interruption. The interruption program must then be able to decide which of the signals of its class are present. Special orders have been built into the machine for this purpose: they read the so-called "class word" and transfer it to A or S. Every interruption signal of a class corresponds to a specific bit of the class word that keeps a record of the presence or absence of the interruption signals of the class in question. As soon as the presence of an interruption signal has been recorded in the arithmetic unit, i.e. when the class word has been read, the interruption signal ceases to exist.

Finally there is a method of preventing all interruptions without altering the permit. The X1 may be in one of two states, viz. "susceptible" or "non-susceptible". The above description holds for "X1 susceptible"; for"X1 non-susceptible" no interruption takes place, regardless of the permit. Transition from susceptible to non-susceptible and vice versa can be brought about by special orders. Furthermore, the X1 becomes non-susceptible, as soon as an interruption has taken place.

The hierarchy of interruptions can be controlled by the interruption permit. The state of susceptibility of the X1 serves another purpose. With the interruption facility an entirely new element is introduced: the actual order in which the X1 performs its different tasks is no longer determined by the program only. To enable the program to fulfil its total function properly, it is desirable that uncertainty about the order of execution be excluded at certain critical stages of the program. (This applies particularly to those situations where the different tasks are connected to each other.) This has been realized - without altering the interruption permit! - by the introduction of the non-susceptible state. As a rule non-susceptibility only lasts for a few orders.

2.7.2 The preserving function of subroutine jumps

A detailed description of the preserving function of

the subroutine jump 6T can now be given. The word (s_m) records, from left to right:

d_{26} stop bit = 1 stop at end of subroutine
= 0 do not stop " " " "

d_{25} ⋮ d_{20} interruption permit bit, class 6 ... class 1 = 1, permit interruption
= 0, prevent interruption

d_{19} susceptibility bit = 1 when susceptible
= 0 when non-susceptible

d_{18} condition bit = 1 when negative
= 0 when affirmative

d_{17} last sign bit = 1 when negative
= 0 when positive

d_{16} overflow bit = 1 when overflow
= 0 when no overflow

$d_{15} = 0$ (unused, see 3.5)

d_{14} ⋮ d_0 instruction counter bits .

The reverse process takes place in the restoring jump; the restoration does not only cover condition etc. (see 2.4.3), but also the interruption permit and the susceptibility.

2.7.3 Orders related to interruptions

The setting of the interruption permit can be done by means of a restoring jump, for instance in the additive version 0T when a non-absolute address n is used. One makes the 15 least significant digits of (n) equal to nought, control therefore does not jump. Note that the order destroys condition etc!

(If we wish to make the X1 susceptible with all classes allowed by the interruption permit but we do not wish to change condition, last sign and overflow indication then we could make use of the following piece of program. It is assumed to start at address 0 Z E0.)

		DA	0	Z	EO	DI
0		6T	1		EO	0
1		2A	8		XO	
2		OP	11		AA	
3		2LA	7			A
4		OLA	24		X31	A
5		1P	11		AA	
6		6A	8		XO	
7		OT	8		XO	P
8						

(see 4.1.3 and 4.1.4)

Another way of modifying the interruption permit is to use the following order:

	OY	n		XS			$(0 \leqslant n \leqslant 126)$.

This order removes those classes from the interruption permit that are indicated by a 1 in the binary representation of the number n (i.e. a type of collation with the inverse of n); as there is no class 0, n is always even.

Two orders from this group have a special function, viz. for n = 0 and n = 126.

	OY	0		XS			make the X1 susceptible	] without altering the interruption permit!
	OY	126		XS			make the X1 non-susceptible	

The following orders enable the interruption program to detect which interruption signals from a class are present, their presence being indicated by a 1 and their absence by a 0. They are $(1 \leqslant k \leqslant 7)$:

	kY	4		XP			read k-th class word into A	
	kZ	4		XP			" " " " "	S .

The interruption signals cease to exist as soon as their class word has been read out by one of the above orders.

The interruption of class k is effected by means of an inserted subroutine jump which uses s_{8+k}. Of the (s_m)'s with $9 \leqslant m \leqslant 15$ the programmer can only use those that correspond to unused classes.

Remark. The seven "interruption subroutine jumps" which can be inserted are stored in consecutive addresses in the dead memory, viz. 1-7 D16 (see Appendix 4).

2.7.4 The interruption by means of the keyboard

Class 7 is somewhat different from the others. Its interruption signals are not obtained from an independent mechanism, but from the keyboard on the console of the X1 (see 2.8). Pressing the keys while the X1 is in action has no effect at all; pressing the key when the X1 has been stopped will start the X1 and produce an immediate interruption of class 7, irrespective of the contents of the interruption permit and of the susceptibility of the X1. The fact that the X1 was started by the interruption is recorded by $d_{26} = 1$ in s_{15}. At the end of the keyboard interruption program the status quo is restored, amongst others, by means of a restoring jump via s_{15}, which then stops the machine. The keyboard makes use of an interruption in order to be able to restore the status quo and does not use it for the purpose of synchronization!

In class 7 the reading of the class word is replaced by a reading of the number of the key that has been pressed. For this purpose the keys 0 to 9, ., +, -, F, G and H are numbered from 0 to 15, in this order. The keyboard program can thus detect which key has been pressed. This may have considerable influence on the action of the keyboard program, as will be clearly shown when we discuss the so-called autostarts (see 4.4.2).

2.8 The console of the X1

A number of switches and indication lights are to be found on the nearly vertical panel behind the section with nineteen keys and two switches on the surface of the desk. The latter are arranged as follows:

	BNA	+	1	2	3	F
SNA	BCA	-	4	5	6	G
SCA	DO	.	7	8	9	H
				0		

The two switches at the extreme left of this section control the stopping of the machine.

SNA "Stop Next Address" When this switch is switched on while the X1 is working, the machine completes the order under execution and then stops. The order counter T then contains the address of the next order to be obeyed. Amongst others this switch is used to stop the machine instantaneously. When one starts the machine while SNA

is on, it stops after completing the first order. When a subroutine jump is executed while SNA is on, the stop bit of the link concerned is made = 1. The switch SNA therefore enables us to execute a program order by order for the purposes of inspection. If this program calls in a (standard) subroutine, the intermediate results of which do not interest us, we can put the switch SNA off and let the X1 execute the subroutine at full speed. As the position of SNA made the sign bit of the link = 1, the X1 stops automatically after completing the subroutine (see 2.3).

SCA "Stop Chosen Address" This switch enables one to stop the machine at any desired point in the program, the point being fixed by means of the 15 switches of the stop address. When SCA is on the X1 stops as soon as the order in the stop address has been executed; the order counter again contains the address of the next order. The 15 switches of the stop address are at the top of the vertical panel on the right hand side; any address can be specified by these switches (down = 0, up = 1).

The machine is started by means of push buttons.

BNA "Begin Next Address" On pressing (and releasing) this key the machine starts working, beginning at the order to which the order counter refers. When SNA is off the machine continues working, when SNA is on the machine stops immediately after the execution of the first order. The key BNA is therefore used to allow a stopped program to continue its operation, it is also used to carry out (part of) a program order by order.

BCA "Begin Chosen Address" By using this key it is possible to start the machine at any specified point. The address of the first order to be executed must be given in the 15 switches of the so-called start address. The start key BCA copies the start address into the order counter; from then onwards its action is the same as that of the key BNA.

The switches of the start address can be found at the top of the vertical panel on the left hand side; as in the case of the stop address the address is given in its binary form.

DO The key DO is used primarily in testing. Once this key has been pressed the X1 executes the order specified (in binary form) in the 27 so-called word switches; these can be found at the base of the vertical panel.

When SNA is on, the order in the word switches is obeyed only once; when SNA is off the order is executed repeatedly, until SNA is thrown. Provided this order is not a non-skipped jump order, the contents of the order counter will not be altered by pressing DO.

The term "keys of the keyboard" applies particularly to the remaining 16 keys on the horizontal section (see 2.7.4). They are used to cause the X1 to do all kinds of standard service operations. These include typing information stored in the memory or in the registers, introducing orders or numbers by hand, starting the tape read program, etc.

We have already mentioned the switches of the start and stop addresses and the word switches on the vertical panel. Each of these three rows of switches corresponds to a particular word in the dead memory:

(1 DO) ≡ start address] supplemented by 12 zeros at
(2 DO) ≡ stop address] the most significant side
(3 DO) ≡ console word (i.e. the contents of the word switches).

Although these words can be varied, they belong in the dead memory: they can be read by the program, but not filled in! The facility to read these words is of the utmost importance: it enables the programmer to construct programs, the course of which can be "externally" influenced.

The remainder of the vertical panel is used for purposes of display. There are two rows of 28 and 27 lamps on which the contents of a number of registers can be seen: the upper row can show the contents of OR, U and M, the lower one those of A, S, B and T. For each row the selection of the register to be shown is made with the aid of a rotary switch.

The registers U and M require further explanation.

The U-register is the central register of the X1; it was not necessary to discuss it before as the programmer has no access to it. In general one can say that the arithmetic result of every operation is left in the U-register. If one makes the contents of this register visible on the indicator lights, while the program is run through order by order, the intermediate arithmetic results can be seen in U, no matter where they are formed, in A, or S, or B, or in the memory. The results of transports also pass through the U-register. (What is

left in the U-register can be deduced from the fact that condition-setting orders analyse the final contents of U.)

The M-register acts as "gate" to and from the memory, and is the only place where one can see the parity bit.

In this context it should be pointed out that it is possible to look at a number in the memory without affecting the contents of any of the "active" registers. To do so one may place a "U 2A-order" - with the appropriate address - in the word switches and push the button DO. The word in question then appears in U and M.

When the contents of the order counter T are shown, the interruption permit and the susceptibility are displayed in the more significant positions. More precisely, they show the link which would have been formed if the order had been a subroutine jump, except for the fact that the lights in positions d_{18}-d_{16} are not used. They need not be used as the condition, the last sign and the overflow indication are permanently displayed by three separate indicator lights at the top of the vertical panel. Finally there are two lights which indicate "wrong order" or "wrong number" respectively when the machine has stopped on account of a discrepancy of the parity check.

The reader will understand that indication lights are not used as long as program and machine are perfect. While the X1 is running there is not much to be seen on the lamps either: the contents of the registers change far too frequently. An experienced observer will perhaps be able to follow the course of the program to some extent by looking at (the most significant side of) the order counter. The advantage of the indicator lights will be fully realized when the X1 is stopped: in the debugging of programs they are invaluable.

2.9 The speed of the X1

The terms "additive" and "clear", used below, serve to distinguish those orders that do involve an "actual" addition (i.e. carry propagation) from those that do not; both transports and logical operations fall under the heading "clear".

Register transports are included in the category "Absolute and communication" as no second memory contact is required for their execution.

We note that the variant of condition-setting never influences the time taken by an order. Condition reaction only has an effect when the order is skipped. The A,B,C-variant may affect the time except in skips, multiplications and divisions.

The times are:

1. Skip (in all cases)		32 μs
2. Multiplication and division (in all cases)		500 μs
Remark: Under all circumstances the fast multiplications by 10 require		64 μs
3. Shift and normalize orders (shift over n places)		40+8n μs
4. Absolute and communication,	clear	36 μs
	additive	44 μs
5. Normal:	clear	64 μs
	additive in	64 μs
	additive out	76 μs
6. B-correction		no change
7. C-correction (except those under 1. and 2.)		8 μs extra

3 DISCUSSION OF SOME OF THE FEATURES OF THE X1

This chapter is devoted to some of the considerations which played a role during the design of what is described in the previous chapter. Where possible we shall mention the arguments for and against the decisions, especially when these arguments were of significance in programming.

The reader should not expect to be presented with a set of conclusive arguments. In the first place, it is not possible to mention all the various arguments considered in the design, in the second place the X1 is definitely the product of steady growth and development: many techniques and methods were copied from earlier machines without much discussion, as experience had shown us that they were perfectly satisfactory.

Some of the motives, however, cannot be appreciated without a knowledge of the requirements the machine had to meet.

3.1 The requirements

Although important from other points of view, we may ignore some of the requirements here, such as price, reliability, volume, power consumption etc. We are more concerned with the fact that the X1 had to be: "general, fast, adaptable and elegant".

The term "general purpose computer" is often appied to machines specially designed for scientific computations. When we say that the X1 must be a "general" computer, we mean that the machine must be suitable for both scientific and clerical work. This requirement had a bearing, amongst others, on the composition of the order code. We tried to avoid orders which would only be used in certain types of work as far as possible.

As is well known, the speed a machine must attain before we call it "fast", is not clearly defined and continually increases. Until now, the demands made by the scientific user in this respect appear to be without upper limit, and if the needs of the clerical user have been given more attention it was for this reason. The outcome was a machine which can cope, for example, with a number of punched card machines, one being a fast sorter (42000 cards per hour); if the required processing does not involve too many operations, it must even be possible for the X1 to use two such sorters (both working at their maximum capacity) as fast input mechanisms.

By the term "adaptable" we mean that the standard basic machine can be provided with all kinds of requisites and apparatus which may be necessary for a specific task. Once an organization has a machine the demands made on it often increase as time goes on, and the possibility of enlarging an installation which was initially a modest one, is a major aspect of the adaptability of the machine. Such an enlargement may consist of extending the memory, or adding more communication mechanisms. In this respect it is essential that the way in which a mechanism is coupled to the machine will not be affected if further apparatus is added later.

As we shall see later the factor of adaptability was also considered in more than one way in the making of the tape read program.

The adaptability - being an aspect of the required generality - is separately mentioned here, because it has had definite consequences: it must be possible to extend the basic design, but provision for this possibility should not be too expensive as the extensions may never take place. The same applies to the basic tape read program, that should not include pieces of program that are, as yet, superfluous: its structure must be such that programs for additional facilities may be incorporated.

The term "elegant" refers particularly to the structure of the order code Efficient solutions should present themselves to the most common problems encountered in programming. There is redundancy in the code, however, if a large number of equivalent solutions present themselves at any moment. Such redundancy is harmful in two respects, in the first place because the user pays for it, in the second place, because each programmer becomes accustomed to his own personal methods, so that reading another man's programs - difficult enough in itself! - becomes even harder.

On the other hand a code should not be so rigid that there is only one reasonable solution in any particular situation. One should be able to modify an "average" program in one of two ways: either by speeding it up (at the expense of more memory space), or else by making it more compact (at the expense of time of execution).

In the case of a machine with a relatively expensive memory, the latter is often not too large. It may then be important to find a satisfactory compromise between the conflicting requirements "fast" and "compact". In scientific computations it often happens that some parts

of the program have to be passed through much more often than others. It is then worthwile to make those parts of the program which are most intensively used as fast as possible, be it at the expense of some memory space; if necessary, the space can probably be regained in the less intensively used parts of the program. To arrange programs along these lines should not cost too much time and energy: an elegant code should facilitate the making of more speedy or more compact programs.

3.2 Arithmetical facilities

Historical factors undoubtedly played a great role in the choice of the representation of numbers in the machine and the facilities the arithmetic unit should provide. The registers A and S, for example, can be found under the same names in three older machines designed by the same group; in all three they already function as independent accumulators. We will nevertheless try to give some justifications.

Some consideration makes evident what experience has taught us, viz. that programs for a machine equipped with two accumulators run much more efficiently than those for a machine with only one accumulator: in the latter case programs are considerably lengthened by numerous transports to and from the only accumulator.

As we add more registers to the arithmetic unit the relative gain in efficiency continually decreases, unless ... the new register has a special function and thereby provides us with entirely new facilities! The B-register definitely satisfies this condition, due to the facility of B- and C-correction: the presence of the B-register is justified by the usefulness of these facilities, although its inclusion would <u>not</u> have been worth while if it had merely been a third accumulator.

Amongst others, technical considerations were responsible for the development of B into a register with all accumulator facilities: the difference in price between an accumulating and a non-accumulating B-register was relatively small, and the former was sufficiently attractive to justify the extra cost.

The term "all accumulator facilities" used above includes the additive out-order, by which the contents of a storage location are increased or decreased by the contents of a register. (The concept of the additive out-instruction originated through technical considerations.

As reading from the ferrite core memory is destructive, each word must be rewritten after it has been read; but then, another possibility is to write the sum there!) An important application of the additive out-order is making a sum (in the memory) when each new term is formed in one of the registers. If the additive out-order were not included in the code the addition of a new term would cost two orders, which would imply a repetition of the address of the partial sum, i.e. "low information density" (see 3.3). The addition of a new order to the code tends to make the latter more complex. In this case, however, I am of the opinion that the additive out-order makes the code easier to handle: adding a new term to a partial sum is felt to be one operation to such an extent that spending two orders on it seems unnatural. If many numbers in the memory are to be increased or decreased by the same quantity the additive out-order enables one to program this in a very compact way.

The contents of the registers A and S must be considered as one double length number after multiplications (0X to 3X) and before divisions (0D and 1D). In additive multiplications the product is increased by a single length number at the least significant side; a carry, if present, is covered by a suitable adjustment in the more significant half, irrespective of the combination of signs involved. Nevertheless the X1 is <u>not</u> equipped with a double length accumulator as such. To use A and S together as a double length accumulator and yet retain the possibility of using each independently, it would be necessary to have a kind of "coupling" between A and S. This did not really fit into the general idea of the design, and the double length accumulator was eventually rejected as its field of application is virtually restricted to double length arithmetic: multilength arithmetic has little use for it.

(In the course of our investigations we were able to formulate two special forms of addition. By using them, both double length and multi-length additions could be performed. They are, however, not included in the code.)

The three previous machines were binary machines too, which also made use of the inverse system to represent negative numbers. A strong argument in favour of the inverse system was the existence of the logical orders, cf [4] . The advantages and disadvantages of the binary number system have been discussed too often and too extensively to go into the subject any further here, cf [3],[6]. The disadvantages are confined to communication with the outside world - used to decimals! - and we therefore mention the measures taken to meet this difficulty.

Fast multiplications by 10 (see 2.6.5) are included in the code as the time factor can play a role in automatic communication, e.g. when punched cards are used.

To enable the operator to introduce decimal numbers into the machine manually, the console holds, amongst others, the decimal keys of the keyboard (see 2.7.4, 2.8, 4.4). Furthermore, it is possible to type out the contents of an arbitrary storage location by using a single key autostart (see 4.3). *)

Operations for floating point arithmetic have deliberately been excluded from the order code. Their inclusion seemed to be inconsistent with the required "generality" in two respects: in the first place because the clerical user is seldom interested in floating point arithmetic, in the second place because it was not clear, to us at any rate (cf [1]), how built in operations for single length floating point arithmetic could profitably be used in programming arithmetic operations on floating point numbers of greater length.

However, normalize orders have been included, primarily to facilitate the programming of floating point arithmetic.

3.3 The word length

In order to make the X1 sufficiently fast it has been provided with a ferrite core memory, and all transports and additions are performed in parallel. In consequence, the costs per arithmetic unit and per storage location are nearly proportional to the word length. Thus there are numerous arguments of an economic nature which do not encourage the choice of a long word length. On the other hand, there are strong objections to too short a word length: one then has to resort to multi-length techniques too often.

It is difficult, however, to draw a definite conclusion about the ideal word length of a general machine from considerations of the size or required accuracy of the numbers the machine will have to handle. For example, one

---

*) In my opinion the most important of the two forms of "incidental" communication just mentioned is the input of numbers and not the output: I have seen many programs debugged at the machine by inspection of the binary words, without a single intermediate result being typed out.

can obviously not support the statement that a word of 26 bits is too short for most calculations, while a length of 27 bits would be sufficient!

(The only positive conclusion I can draw from these observations is the following: the order code of a scientific machine should at any rate be suitable for efficient programming of multi-length operations, because a word length which would diminish the need for them sufficiently is absurd.)

Apart from numbers, the memory has to store instructions. When one tries to find a word length ideal for this purpose, one can come to a more definite conclusion. In the early stages it had already been decided that the memory should hold 2^{15} words at most, each of them being at the immediate disposal of all orders: 15 bits are therefore reserved for the address part of the order. The function part gradually grew to 12 bits; the worth length then grew to be 27 bits, i.e. the minimum number of bits necessary to accomodate both function and address.

Longer addresses bring with them arguments in favour of longer function parts. A shorter function - i.e. a less powerful code - results in longer programs, which contain more orders with addresses that are, in consequence, less interesting (viz. repetitions of addresses, addresses of jumps, more references to working space by means of addresses that are, in themselves, unimportant, etc.). The longer the address part, the larger the number of bits spoilt by an excess of addresses.

A function part that is too long, however, is harmful in two respects. The code then provides so many facilities that many of them will often remain unused; the redundancy brings with it an unnecessarily large number of bits for the function. Also, excessive flexibility makes a code too difficult to handle. There are then so many possible functions that the programmer no longer knows them all individually, but views them as consisting of different constituent parts. The number of "constituent parts", i.e. the various aspects of a whole operation, should not be so large as to be confusing. What we understand by "confusingly large" does not only depend on the expected intelligence of the future programmers, it also depends on the way in which whole operations can be subdivided. The role of each constituent part should be well defined and independent of the others as far as possible; furthermore the order notation should be such that the various aspects of each order can be clearly shown on the program sheets.

Obviously the above arguments are all purely qualitative and they did not compel us to decide upon a function part of 12 bits; there are 6 for the main function and 6 for the variants. The latter do not specify six independent variants, instead they specify only three variants, each using two bits. This was done intentionally, to avoid confusion. We were able to define the main functions and variants in such a way that the number of meaningless combinations remained small enough to be acceptable. The X1 order code is not an easy one to use to full advantage, due to its high degree of flexibility. For this reason the notation for the main function, and particularly that for the variants, was chosen with the utmost care.

In one respect the above considerations about the word length are more or less obsolete: they date from a time when it was not sure that the dead memory could be constructed at relatively low cost. As soon as a considerable portion of the orders required for a program are recorded in the dead memory, i.e. in a less expensive medium, the motives for choosing the word length exactly equal to the order length, and not one or two bits longer, become considerably weaker. The arguments are further weakened because another method of reducing the number of order bits has not been investigated extensively: less address bits would siffice if one is willing to sacrifice the facility that every possible storage location is at the immediate disposal of every order.

Finally, one should realize that the need for a long word length - often only a supposed one! - is felt almost exclusively by the scientific user; in his dreams he is inclined to be rather demanding in this respect. Nor will his conscience deter him, as this is a form of redundancy that will never inconvenience him. On the contrary, it can only make things easier for him and considerations of price seldom enter into day-dreams!

3.4 The condition

In the code of EDSAC 1 (cf [7]) - quoted as a classical example - the conditional jump orders react to the sign of the (only) accumulator; in one of the conditional orders the jump is executed if the number in the accumulator is positive, otherwise it is skipped, in the other the inverse reactions take place. An extension of this system to a two- or three-register machine would imply an increase in the number of jump orders in the code, viz. a pair for each register.

The same number of jump orders, however, will suffice, if a special so-called condition register of one bit is introduced. The reaction to a sign digit then takes place in two stages: in the first stage the sign digit is copied into the condition register (condition-setting), in the second stage conditional orders are executed or skipped depending on the current content of the condition register (condition reaction).

One of the advantages of this arrangement is that condition-setting need not be restricted to the copying of a sign digit, but may take place according to other criteria. The X1 computer - being a three-register machine - is equipped with a condition which can be set according to one of three criteria, the sign test, the zero test, and the test for equality of signs (see 2.4.3).

Once the whole process of conditional execution is split into two stages, one must decide where the possibility of inversion is to be included. An inversion possibility in both stages - although logically permissible - would be superfluous, no inversion possibility gives rise to less efficient programs.

One method is to have the possibility of inversion in the condition-setting (there are then two different sign tests in which the sign digit, or its inverse, is copied into the condition register) while conditional orders react uniquely to the condition. Alternatively, no possibility of inversion is incorporated in the condition-setting (in the case of the sign test the sign digit, and not its inverse, may always be copied into the condition register), and two kinds of conditional orders are included in the code.

As long as all conditional orders of a code are jumps, it does not matter very much to the programmer in which of the two stages one incorporates the possibility of inversion. As soon as <u>every</u> order can be conditional, however, it is definitely preferable to include the possibility of inversion in the condition <u>reaction</u>.

Indeed, if every order can be made conditional, the machine can, dependent on the condition, skip a number of orders without a conditional jump being used. However, one often wishes to execute either "operation A" or "operation B", as indicated by the condition: this can also be done without using jump orders if the inversion possibility is included in the reaction to the condition. Condition-setting is then followed by the orders for operations A and B and, depending on the condition, the orders

of either operation A or operation B are executed, while the other set is skipped.

This decrease in jump orders considerably shortens many programs. One should bear in mind that the gain in space may be accompanied by a loss in speed: if a large number of orders is to be skipped considerations of time may make it worth while to use a conditional jump; this is definitely not the case if the orders concerned are only to be skipped in exceptional cases!

The U-version is grouped under the variant of the condition reaction. This variant is described by two bits in the function part, and the U-version gives a worthy meaning to the fourth possible combination. It is pointless tc use an order with "undisturbed destination" unless the result is used to set the condition. The supposition that conditional condition-setting is such a complicated operation, that the need for it will be negligible, reconciled us to the fact that the U-version on the one hand, is incompatible with the Y- or N-version on the other. To be honest we should mention that the ingenuity of some programmers, at least, was underestimated by this supposition.

In the U-version jump orders are conditional to the overflow indication (see 2.4.2). Sometimes the absence of an inversion possibility "costs" an extra order,it is true, but the considerable gain that results from the facility of overflow detection at all is fully attained. At the moment of writing no programmer has, to my knowledge, felt the incompatibility of the U-version and the normal reaction to the condition as a restriction in the case of jump orders.

It is possible to execute a number of ordinary orders between the condition-setting and the subsequent condition reaction, thanks to the separate condition register (and the possibility for orders to leave the condition unaffected): in other words, the condition register is really a memory element that leaves the arithmetic registers available for other information. Conditional reaction of an order to the sign of a register may actually amount to using a whole arithmetic register to store one bit!

Two bits in the function part describe the variant of condition-setting; the four possible combinations indicate "leave condition unaltered", and "set condition according to P, Z or E", The problem of incompatibility does not arise here. Once the inversion possibility is excluded from the condition-setting, it must be decided

in which form the questions will be asked, e.g. in the zero test, shall we ask "Result = 0?" or "Result ≠ 0?".

When only one criterium is applied for condition-setting, it does not matter how this question is formulated: inversion of the question then only amounts to interchanging Yes- and No-conditionality (two symmetrical possibilities of reaction!). Since the X1 has three criteria one must ask oneself - for the application of the second and third criterium - whether there are reasons for preferring one question to the other (its inverse).

As soon as the programmer thinks of condition-setting as recording the answer to a question, it is preferable to formulate those questions in their most natural form, i.e. without negation. Hence, in the Z-version, one asks whether there is equality to zero, in the E-version, whether there is equality of signs. A second argument in favour of these choices is related to the preference of the X1 for -0.

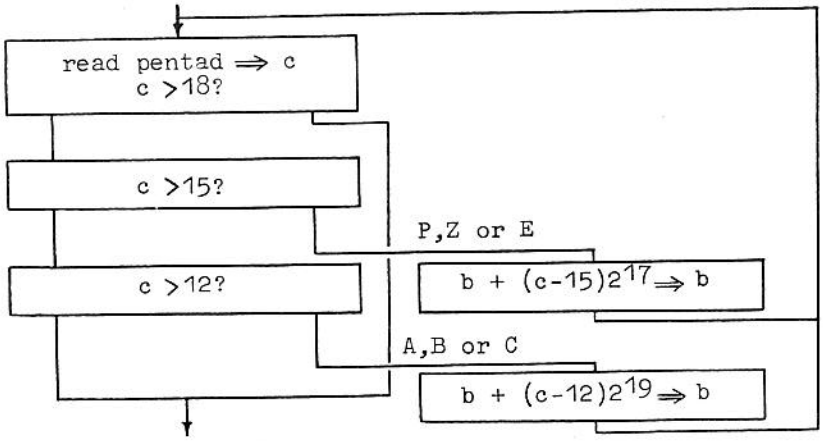
A test which is often used is the sign test as applied to a result that can only be > 0 or = -0. Here the P-version sets the condition inversely to the Z-version! This possibility may be of importance when the setting of the condition for a certain condition reaction in the program can take place at more than one point.

The use of the E-version is not restricted to comparison of the (unknown) signs of two numbers. Thus one can make use of the fact that with "last sign negative" the E-version sets the condition inversely to the P-version. One can even use the E-version to let an order ask, either whether the result is positive, or whether the result is negative, by letting the order be executed (intentionally!) with last sign positive or negative respectively.

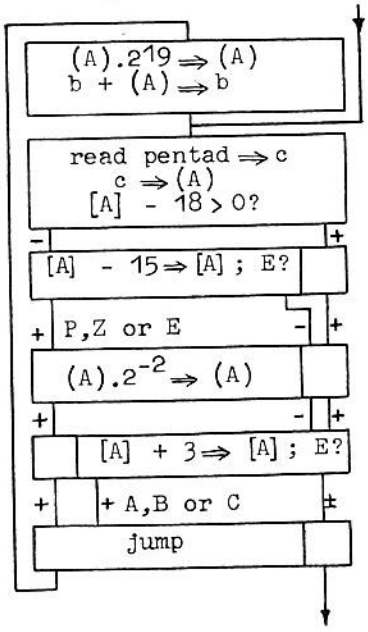
In illustration we discuss a section of the tape read program. In reading the symbols that specify an order, the tape read program must detect whether the address is followed by "A, B or C" and/or "P, Z or E"; if so, the function part under construction must be suitably modified and a new pentad must be read.

A detailed description of the operations to take place is given in the flow diagram below. We use the convention that a block is entered at the top and is left at the bottom. If a question occurs in a block, we leave it on the right hand side if the answer is "Yes", on the left hand side if the answer is "No".

In explanation: the letters P, Z and E correspond to c = 16, 17 and 18 respectively; the letters A, B and C correspond to c = 13, 14 and 15 respectively. The function part is built up in the word denoted by b; in this



word the pair of bits describing A, B or C are to be stored on the immediate left of the pair of bits describing P, Z or E.



For the purposes of clarity we make use of a flow diagram to describe the procedure in the X1 program. Where relevant the content of the last sign register is shown by + or -. The question about equality of signs is indicated by "E?". Conditional execution is explicitly shown by dividing the block in two by a vertical line.

(N.B. To appreciate the compactness of this piece of program, one should realize that a single order corresponds to every line in the diagram, in contrast to the previous diagram; furthermore no jumps were shown there.)

The section described here is to be found in the tape read program at the addresses 14 D12 to 22 D12 (see Appendix 4).

It is not the ordinary use of the E-version that is shown by the above example! On the contrary, it illustrates its hidden possibilities; they become apparent when it is used in an unconvential way.

Neither condition-setting, nor condition reaction cost extra time. This sometimes makes it possible to speed up a program without including extra orders, for instance by skipping the addition of a new term to a partial sum, when the new term happens to be = 0 (see 3.9.4).

Finally: with the exception of the counting jump, no order reacts conditionally to its own result. It is for example not possible to subtract, by means of one order, a certain quantity from the contents of a register only if the result is positive. Similarly, orders that operate on the absolute value of numbers are not included in the code.

Operations on the absolute values of numbers must be executed with the aid of the condition; for example -assuming that y = (2 X1)-:

	2A	2	X1	P	$y \Rightarrow (A) \geqslant +0$?	total function:
N	5P		AA		if not: $-(A) \Rightarrow (A)$	$\lvert y \rvert \Rightarrow (A)$

and

U	2A	2	X1	P	$y \geqslant +0$?	total function:
Y	0A	2	X1		if so: $(A) + y \Rightarrow (A)$	$(A) + \lvert y \rvert \Rightarrow (A)$
N	1A	2	X1		if not: $(A) - y \Rightarrow (A)$	

3.5 The subroutine jump

The index m that follows the address of a subroutine jump can have a value from 0 to 15 inclusive (see 2.3). The programmer can use the subroutine jumps with $m \leqslant 8$ freely; those with $m \geqslant 9$ are reserved for the effectuation of the interruptions (see 2.7.3).

This makes it possible for the main program to call in a subroutine, which in its turn calls in a subroutine, which in its turn calls in a subroutine, etc. until the ninth subroutine has been called in.

It is convention to choose m = 0 for those subroutines that do not call in another subroutine; for subroutines that call in a sub-subroutine with at most m = 0, the m

is chosen = 1, etc. All this applies to subroutines that call in other known subroutines. However, as soon as a subroutine calls in an "arbitrary" subroutine (as in the case of a standard integration subroutine which calculates the integrand with the aid of a special subroutine) this is no longer possible. In such cases it is always safe to call in the outer subroutine with m = 0; the latter starts off by transferring (s_0) to a location in its own working space. The restriction that the index m can "only" take on nine different values can be circumvented by the same technique. (If need be a single value of m, e.g. m = 0, would suffice; transferring (s_0) would then be rule and not exception. We will encounter something of this nature in the communication subroutines to be dealt with later; they restrict themselves to the use of s_{14}. The reason for this is the same as that for which they do not use counting jumps (see 3.6).)

All this is only possible thanks to the fact that subroutine jumps write their link in the ordinarily accessible address 8 + m. A further possible application is the technique of "program parameters", as developed for EDSAC 1, cf [7] . Here one or more parameters, giving further specifications of the function of the subroutine, are placed in the storage locations immediately following the subroutine jump; on completion the control (usually) returns to the address that follows the last parameter. As these parameters will be read by B-corrected orders, the subroutine first places (s_m) in the B-register; it is for this reason that the bit d_{15} is = 0 in s_m (see 2.7.2). When (s_m) is transferred to the B-register, d_{15} is copied into the sign digit of B (see 2.4.2).

The subroutine jump to the subroutine and the jump at the end of the subroutine (with the non-absolute address 8 + m) to return control to the main program are. as a rule, the only two extra orders to be executed in comparison to an open subroutine. They each require 64 μs. The fact that calling in a subroutine needs only one order in the main program (leaving the contents of the arithmetic registers intact) is also attractive as far as program space is concerned; as a result it is worth considering programming even rather simple operations as subroutines. This is one of the ways in which the speed of the machine, and even more the efficiency of the code, compensates for the price of the memory.

The fact that orders to "plant the link" are not necessary on entering the subroutine, facilitates the

making of compact, fast and flexible subroutines even further. (The same advantage can be found in machines with less refined subroutine jumps but more B-registers.) It is then possible to make subroutines with different points of entry elegantly.

Much space can often be saved by combining subroutines that perform practically the same operation - in slightly different versions - into one single subroutine in such a way that the different versions are distinguished from each other by different points of entry. In the simplest case one version can be derived from another by omitting a number of orders at the beginning: one then jumps to the first order to be executed by means of a subroutine jump (with the same m).

A simple example of a subroutine with more than one point of entry is the computation of the sine or the cosine. The cosine subroutine begins by increasing the argument by $\frac{1}{2}\pi$, after which the sine of the sum thus formed is calculated. The entrance for the calculation of the sine is such that the addition of $\frac{1}{2}\pi$ is omitted, so that the sine of the given argument is computed.

3.6 The counting jump

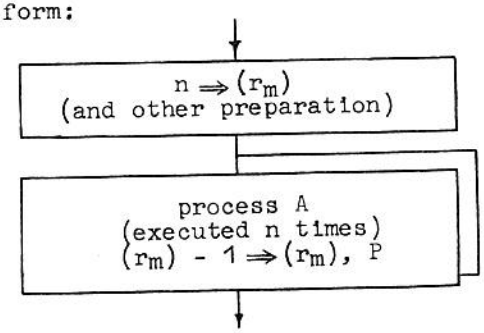
Counting jumps are designed especially for loops that must be passed through a certain number of times. As they are not used in the communication programs to be dealt with later, we illustrate the way to use them by a few examples.

The communication programs to be dealt with later are executed at moments that are not exactly known in the main program (see e.g. 4.5). As a result the communication programs and the main program may not have working space in common: the information that the main program would like to store in such an address is then under permanent danger of being destroyed by the communication program. In order that all counting jumps may be at the disposal of the main program, they are not used in the communication program; the communication programs in question naturally make use of (s_{14}) - due to the interruption of class 6 - but further restrict themselves to subroutine jumps with $m = 14$, so that the programmer of the main program is not denied the use of more subroutine jumps.

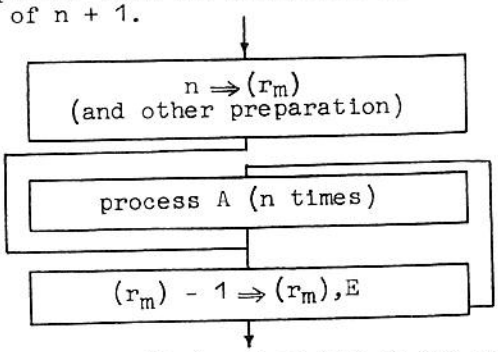
In the counting jump the execution of the jump may depend on the new value of (r_m). In order to avoid confusion

with the ordinary condition in flow diagrams this dependence is indicated by one of the letters P, Z or E without the question mark, and the control leaves the block at the side in case of the execution of the jump, otherwise it leaves it at the bottom.

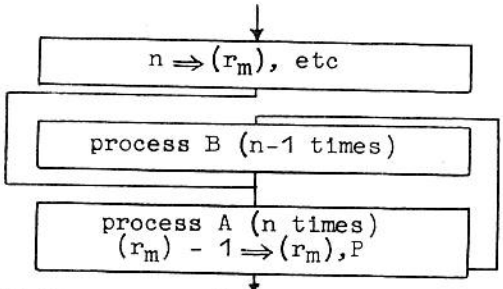
For instance, if the process A has to be executed n times, the flow diagram would roughly take the following form:



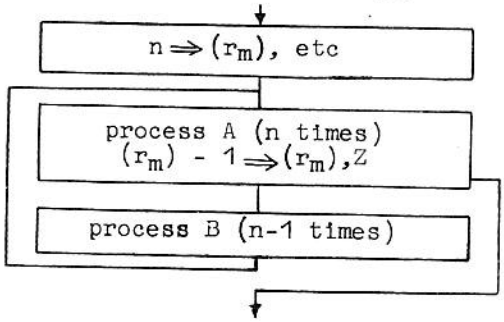
This diagram only applies to the case $n \geq 1$. Particularly in the case where the number n is a previously computed quantity, it often happens that the minimum value of n is = 0. Here we can use the last criterium E ("Extra Jump") for the counting jump and save a separate test for detection of n = 0 or the formation of n + 1.



It often happens that a loop has to be passed through "n - $\frac{1}{2}$" times, i.e. one part (process A) must be done n times, the other part (process B) only n - 1 times. This is shown in the following flow diagram.

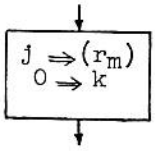


The same process can be accomplished by making use of a counting jump in the Z-version:

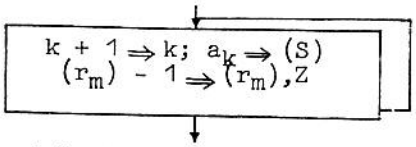


In the previous diagram the counting jump functions as the inverse of the counting jump in the latter. The first diagram is to be preferred, because there the inconditional jump is not included in the loop itself, this in contrast to the second arrangement. Of course this consideration of time is only important for small loops that must be passed through frequently: nevertheless one should make it a habit to test - even in the case of longer loops - "whether the loop must be passed through again". In this way programs become more homogeneous and the probability of errors thus decreases.

A "pure" application of the counting jump in the Z-version is given in the next example. We are asked to place a sequence of numbers a_k (k = 1,2,3,...) in the S-register, omitting, however, a_j.



Preparation



Selection of the next a_k

Being additive out-instructions, counting jumps always require 76 μs, in other words they belong to the group of rather time consuming operations; as they do not use the registers A, S or B, however, they can be particularly useful.

3.7 Address modification

The field of application of the A-version (see 2.4.1) is restricted to integers less than 2^{15} in absolute value. The A-version is applied mainly in "administrative" sections of programs, for instance in the manipulation of addresses, address increments and decrements; it can also be used to great advantage in more general "red tape operations" (counting etc), because, as a rule, only small integers are involved here.

The gain in space and time due to the A-version has already been mentioned (see 2.4.1). Finally, it makes things more convenient for the programmer, for there is no need to reserve a storage location for this constant, or to refer to it with the aid of an - arbitrary - address. Instead he can write the constant in the order straight away.

The A-version is incompatible with B- or C-correction; this is not due to a logical necessity but to considerations of economy (for the similar case U versus Y and N, see 3.4). If the absolute version were compatible with B-correction, the significance of the B-register as arithmetic register would have increased considerably (so much so that it would have been worth considering extending the length of the B-register to full word capacity). For then it would be possible to let [B], increased by a constant that is given by the address bits, function as the operand in each order: one could, for example, multiply (S) by (B)!

The function of the B-correction is well known: an order to be executed with a variable address remains in the memory without the address being altered, while the contents of B are added before its execution.

The C-correction was suggested by technical considerations similar to those leading to the additive out-order. (The fact that C-correction cannot be applied to orders in the dead memory is an analogous restriction.)

When a program loop operates on a sequence of numbers stored in the memory, both B- and C-correction can be applied. If B-correction is used, the contents of B are modified every time the control passes through the loop. If a test applied to the contents of B can then answer the question whether one must remain in the loop, B-correction often results in a slightly faster program than C-correction.

For program loops operating on more than one sequence of numbers, C-correction may give rise to a faster program, particularly in cases where every sequence has its own spacing, or where, for instance, the program in passing through the loop, does not necessarily use the next number from each sequence. (To program this with B-correction one would like to have as many B-registers as there are sequences to be processed; in C-correction the addresses, which vary independently, are recorded in the C-corrected orders.)

The C-correction modifies orders stored in the memory; the programmer is therefore obliged to see to it that the cycle is preceded by suitable preparation, in which the orders in question are set to their initial values; for B-correction the preparation is usually considerably simpler.

3.8 Shift orders

In the early stages of the design it was decided not to include shift orders in the code. There were two reasons for this: in the first place there is less need for shift orders in a code that already includes collation than in one that does not - as in the case of our earlier machines -, in the second place shifting in a parallel machine is not an attractive operation, and certainly not a fast one. Where logical operations did not give the solution, multiplication and/or division would have to do so.

Further investigation, however, showed that this argument did not hold. Besides the fact that shifts had to be simulated by multiplication or division more often than was expected, such simulation usually lead to slow and cumbersome programs. This had various causes.

Firstly, multiplication and division use both the registers A and S, and they use them differently; this resulted in numerous transports: to preserve contents of registers in the memory, and to transfer them to the register where multiplication or division would need them.

Secondly - in contrast to addition, subtraction and logical operations - multiplication and division are the very operations in which sign digits play a special role. Thirdly, it turned out that two multiplications or divisions were often needed per simulation, viz. where numbers of double length had to be shifted.

Fortunately, renewed investigations of the technical possibilities resulted in a comparitively inexpensive method of effecting shifts twice as fast as originally conceived.

The shift orders included in the code are basically those which already occur in the code of the ARMAC (the most recent of the three earlier machines, cf [2]); the facility of shifting to the left is added. They are of such a varied nature that I cannot give an account of all the possible applications. Some remarks will have to suffice.

The circuit AS (the sign digit of S being excluded) is specially designed to handle the (sign consistent) double length number (AS) as it occurs in multiplication and division. As the clear shift has a more precise arithmetic meaning than the round shift, the latter will probably be applied less frequently to the circuit AS than the former; it is not improbable that the reverse holds for the circuit SA, which does not have a precise arithmetic meaning.

Despite the many possibilities of the shift orders we are justified in retaining logical orders in the code. They are considerably faster and they can furthermore be executed with "Undisturbed Destination".

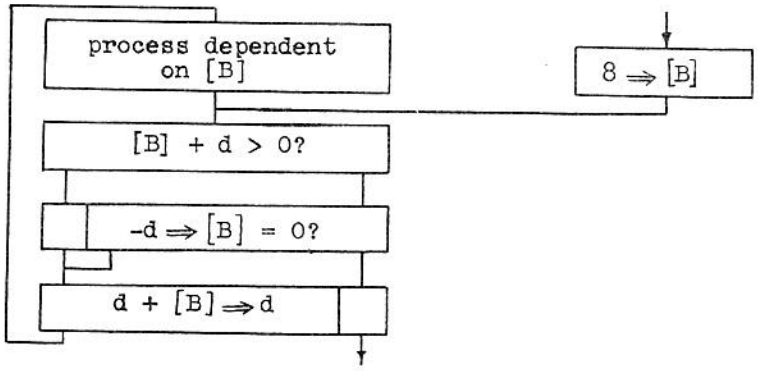
3.9 Some examples

3.9.1 A "step by step reduction to zero"

An integer $d \leqslant -0$ is stored in the memory. This must be reduced to zero in steps, more precisely, it must be reduced to zero by addition of integers in as few steps as possible; in our example the maximum value of the constant c to be added is $c = 8$. A certain process must be carried out at every step; the process is dependent on the length of the step, and it is desirable that this length be stored in the B-register. The process itself will not disturb the contents of B. To start off with B contains the size of the maximum step.

More precisely:
as long as d + [B] $\leqslant$ 0, the addition d + [B] $\Rightarrow$ d must be executed and [B] remains unaltered;
when d + [B] > 0, the substitution -d $\Rightarrow$ [B] must first be made, as this is the distance that separates d from zero; when d = -0, there must be some indication that the process is complete.

The problem is encountered in the subroutine for the conversion from decimal floating to binary floating representation. The solution consists of only three orders, which can be found in the communication program at addresses 28 to 30 D4 (see Appendix 4). We describe it by means of a flow diagram.



3.9.2 Determination of the smallest factor

As next example we give a program that determines the smallest factor (> 1) of a given odd number [0 X1] $\geqslant$ 3, and places that factor in the S-register. The program starts at address 0 FR0 and uses the address 1 X1 as working space.

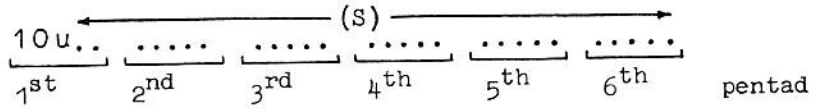
0		2B	2		A	
1		2A	3		A	
2		6A	1	X1		
8→ 3		3A	0		A	
4		3S	0	X1		
5		0D	1	X1		Z
6	N	4B	1	X1		
7	N	0S	1	X1		P

8	N	2T	3	F	R0	A	→
9		2S	1		X1	E	
10	Y	2S	0		X1		
11							

In the above program the number [0 X1] is divided by 3,5,7,9,... etc, each test taking 776 μs. The process can end in one of two ways: viz. when a factor has been found, or when the number turns out to be prime: when the cycle is left these two cases are distinguished from each other by a negative and positive last sign respectively.

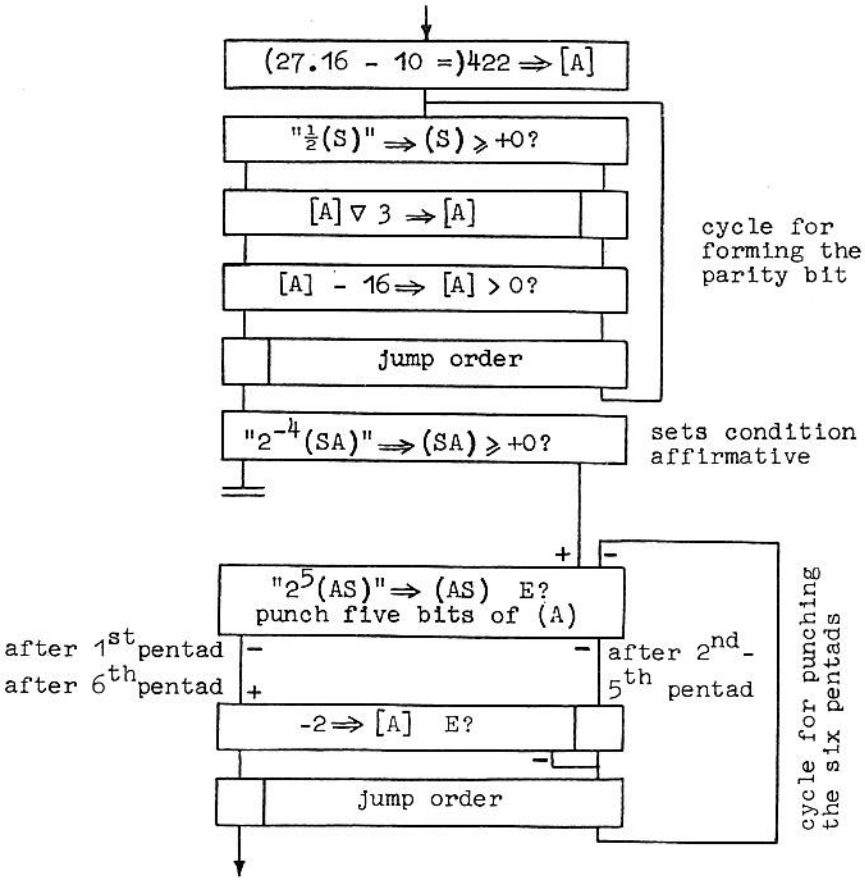
3.9.3 Punching a binary word

It is required to punch out the contents of S in binary form. By convention a word of 27 bits is supplemented by three bits on the most significant side; from left to right they are a 1, a 0 and a parity bit u. The parity bit u is = 1 when (S) contains an even number of ones, otherwise u is = 0. The 30 bits thus defined must be punched in six pentads from left to right as shown below:



The program is shown here in the form of a flow diagram; the program itself can be found at addresses 4 to 14 D14, (see Appendix 4) (The subroutine jump at address 11 D14 sees to the synchronization of the punching mechanism and the X1; as it is of no arithmetic importance whatsoever, it is left out of the flow diagram.)

It was desirable that the program should not make use of counting jumps and should not use the B-register. Besides illustrating the use of the condition, this program shows the use of shift orders: round shifts are indicated by " ").

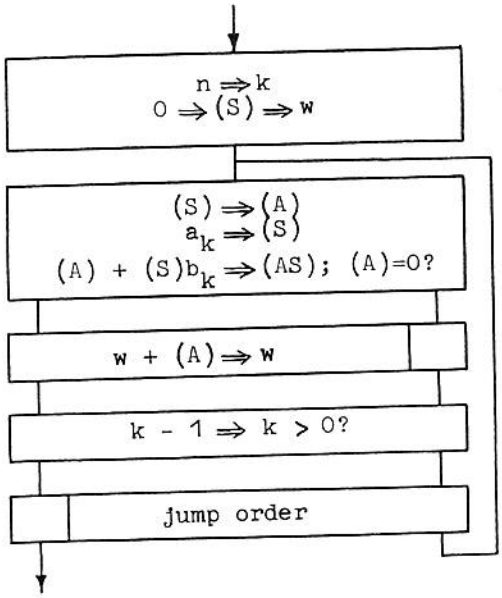


3.9.4 Forming a scalar product

With the proviso, that no overflow occurs, the scalar product

$$\sum_{k=1}^{n} a_k . b_k$$

is calculated in the following flow diagram.



The scalar product is delivered in two words: the most significant half in the working space w, the least significant half in the S-register. The halves need not have the same sign!

The fact that the least significant half of the partial sum is placed in the A-register by means of a register transport, and is then added during the additive multiplication, is not only of importance when a_k and b_k are integers; if they are (rounded) fractions, some reliable digits still appear in S after every multiplication; the gain in precision thus achieved may be of great significance, particularly in cases where n is large.

The addition of the most significant half to w (by means of an additive out order) is skipped when (A) = 0. The execution of the addition requires 76 μs, skipping it 32 μs. There can therefore only be a small gain in time, but it costs no time at all!

If the elements of both vectors are stored in consecutive locations, reference may be made to the numbers a_k and b_k with B-correction, provided the index k is stored in the B-register. In that case a passage through the cycle takes 756 μs (or 712 μs if the additive outorder

is skipped, i.e. a possible gain of 6%).

From this example it is clear that parallel processing of two vectors is faster if the locations of the vector elements are equally spaced. If we are not free to choose a storage allocation of numbers suitable for this purpose, the process can sometimes (and with some luck!) be rearranged so that the sequences to be processed are still equally spaced. The following example is an illustration.

The elements a_{ij} $(0 \leqslant i,j < n)$ of an $n \times n$ matrix are stored rowwise in the memory with fixed spacing in both rows and columns, e.g.

$$a_{ij} = (c + ni + j) \quad .$$

If one has to transpose this matrix, one can construct this process by interchanging the elements of equally spaced sequences, viz. sequences of elements a_{ij} with a constant difference in indices i-j. The spacing in these sequences is the sum of the row and column spacing of the original matrix. (This problem only has a point if the matrix is not symmetrical; if it has to be symmetrical, one can apply the same trick to check its symmetry.)

4 COMMUNICATION PROGRAMS

4.1 The tape read program as independent program

4.1.1 General survey of the functions of the tape read program

Information to be introduced into the machine can be punched on paper tape in a suitable code. For orders this code is the order notation described previously (see 2.5 and 2.6).

All symbols used in the code can be found on the keyboard of the X1 tape punch. The symbols that appear on the keys are tabulated below: there are two symbols on nearly every key. In the table the secondary symbol is written to the right of the main symbol. The latter runs from 0 to 31 and is equal to the numerical value of the pentad punched on the tape.

0	0	8		16	P	24	S
1	I	9		17	Z	25	T
2	J	10	.	18	E	26	W
3	G	11	+	19	F	27	U
4	M	12	-	20	H	28	Y
5	Q	13	A	21	K	29	N
6	V	14	B	22	L	30	D
7		15	C	23	R	31	X

When the punched tape is inserted into the photo-electric tape reader it can be read by the tape read program. The latter has an important "translating" function. For: the programmer has written the orders in a code that is convenient to him, next the orders are punched directly from his program sheets and thereafter it is left to the tape read program to construct the internal binary representation of the orders from the symbols on the tape i.e. from the pentads. Furthermore, when the tape is used for the input of numbers (punched in decimals!) the tape read program must perform the conversion from decimal to binary number system. We call all such operations "assembling", by which term is meant all computation, in the widest sense of the word, that the X1 must perform to construct binary words from groups of symbols that specify the words according to some code.

Apart from "assembling" pentads into words, the tape read program is able to process these words. There are two forms of processing, viz. "writing" and "checking". When the processing cycle "writes", all constructed words are written in the proper locations in the memory; at this

stage the actual input of information into the memory takes place. When the processing cycle "checks", each word delivered by the assemblage is compared with the contents of the corresponding storage location; in the case of a discrepancy the machine stops. This gives one of the methods of checking the input of data; other applications of tape reading in the checking mode will be mentioned later.

Other pentad combinations besides groups from which binary words are assembled, occur on the tape, viz. the so-called directives. They give further specifications about one of the operations (assembling or processing) of the tape read program; as a result there are two main groups of directives.

The first group gives further specifications about the assemblage. In this group we have the so-called type-indications, which specify according to which rules assemblage must take place. Two different "types" are, for example, orders and numbers. The type-indication therefore actually indicates the language (code) in which - until further notice - input data are punched on the tape.

The second group of directives has a bearing on the processing cycle of the tape read program. The most important directives belonging to this group are the address-indication and the change of mode.

The address-indication shows in which storage location the next assembled word is to be stored. (This applies to the writing mode, in the checking mode the address-indication specifies the storage location of which the contents are to be compared with the next assembled word.) Unless otherwise indicated - e.g. by a new address-indication - consecutive words from the tape refer to consecutive locations in the memory. Thus, if one wishes to fill a number of consecutive addresses with information - as is usually the case - one only needs to specify the destination of the first word of the sequence explicitly.

In the description of the address notation given previously (see 2.5.1), we mentioned how the addresses are numbered with respect to the beginning of the paragraphs. In the building up of addresses these reference points (i.e. the beginning of the paragraphs) are used as additive parameters; before this can take place, however, the value of these parameters must have been given to the machine. This is done with the aid of special directives, the so-called "paragraph definitions"; together

they form the so-called "initials", which can always be found at the beginning of the tape.

One can deduce from the previous sections that an active tape read program thus far keeps record of five forms of information:

1) the (current) type indication
2) the value of the (specified) parameters
3) the transport address, i.e. the address of the storage location for the next word to be assembled from the tape. (As successive words from the tape generally refer to consecutive storage locations, the transport address is increased by one after a word has been processed.)
4) whether the processing cycle is engaged in writing or in checking
5) the last initial paragraph letter that was explicitly mentioned (this first paragraph letter is the letter that may be omitted as long as it does not change (see 2.5.1): the X1 must then know which first paragraph letter has been omitted).

Finally we mention that (possible) extra pentads X at the beginning of units of information (directives, orders, numbers, etc) are skipped by the tape read program. The X (numerical value = 31) corresponds to the pentad in which all the holes are used, and can function as "Erase". This makes it possible for a person who is punching to correct mistakes noticed while punching immediately; beginning at the first pentad of the incorrect information unit he punches X's over all the punched pentads of the unit and punches it over again.

4.1.2 Initials

We can sum up what has been said previously (see 2.5.1) as follows. With a view to the address notation the memory is divided into paragraphs (each with a maximum of 17 consecutive pages of 32 words). Each paragraph is denoted by two paragraph letters; paragraphs with the same initial letter together form a chapter.

The programmer is left free to choose the points at which new paragraphs begin. It is wise to choose multiples of 32 for the first lines of paragraphs; by doing so one makes the five least significant binary digits of the address equal to the line number. For longer paragraphs the choice of a multiple of a higher power of two as start address is recommended where possible; this choice makes checking at the machine easier.

The start address of a paragraph is given to the machine by means of a special directive, the so-called paragraph definition. This consists of:

1) the letters DP, followed by
2) the paragraph letters which are to be given a new value (both letters must mentioned), followed by
3) the new value, in address form (i.e. line number, paragraph letter(s) and page number). Here one may only make use of those paragraph letters for which a paragraph definition has previously been given to the machine. In the address of the first paragraph definition on the tape the letter X (or D) will therefore be used.

The division of the memory into paragraphs has been introduced in order to meet the needs of the programmer. In practice only very simple programs are conceived as a whole and written down order by order from beginning to end. Very soon it is found more convenient to split up a computation into sections, each having a separate function which can be isolated to a greater or lesser extent from that of the other sections. It is very important to choose these sections with care: the more clearly isolated the function of these sections the clearer the arrangement of the program.

Normally one allocates a separate paragraph to each section and supplies each paragraph with its own letters. These different letters make the program easier to read, furthermore they make it possible for one to start programming one paragraph while one or more of the others is incomplete. One does not know how much memory space an incomplete one will require; however, as the starting points of paragraphs may be chosen at will one can delay the choice until they are all complete. One then knows their length and can subdivide the memory as seems best.

Which paragraphs one groups under one chapter is suggested only by the logical connection between the paragraphs; it is quite independent of their final position in the memory. In contrast to the pages of one paragraph, paragraphs of one chapter need not be consecutively placed in the memory.

The paragraph definitions which determine the subdivision of the memory for each program are referred to as the general initials. When the tape bearing this information has been read by the machine, the starting points of the paragraphs have been fixed and the rest of the tape can be read.

A second application of the facility to vary the arrangement of paragraphs is the way in which standard subroutines can be incorporated into a program. Although the programmer makes use of standard tapes here - in any case of mechanically reproduced copies of the tapes - he has complete freedom in deciding in which memory locations he wishes to store the standard subroutines.

A standard subroutine usually consists of one paragraph and the programmer using it is left to choose the starting point of this paragraph in the memory. The paragraph itself is referred to by a certain paragraph name on the standard tape of the subroutine. All the programmer has to add to the beginning of the standard tape is the so-called specific initials, by which the parameter in question is given the desired value.

As a rule all subroutines are punched in terms of the same paragraph letters (viz. those with a Z as the first letter); such a paragraph name can therefore occur several times on the tape of a program that makes use of a number of subroutines, but each time with a different meaning. In consequence letter combinations that begin with a Z are not suitable for reference to an arbitrary address in the memory and they may not be defined under the general initials.

In practice it is customary to give the paragraphs of the standard subroutines two names: a paragraph name beginning with a Z and a further name with a meaning as defined in the general initials. The name first mentioned serves only as a means to be able to make use of the standard tape. The programmer only has to consider exactly which paragraph name of the Z-group is used for this purpose when he compiles the specific initials of these subroutines; this new value of the parameter in question, and also every reference to this subroutine in the main program, is made in terms of a paragraph name that has been determined by the general initials. At the same time one ensures that in this way the final location of the program as a whole can be altered by altering the general initials only.

4.1.3 Type indications

The group of directives that shows the most variation is that of the type indications. They specify according

to which code - until further notice (i.e. until the next type indication) - the units of information that now follow are to be punched.

In the following we deal with five type indications. We regard the type indications for double length and floating point numbers as part of the associated arithmetic programs; these fall outside the scope of this description.

DO "Skip blank tape"

The directive DO indicates that as yet it is not known according to which code the assemblage program will interpret the symbols on the tape. Thus the lack of a "real" type indication is also regarded as a type indication.

When the directive DO has been read the machine skips zero's and pentads X until it has found a D. If the tape read program encounters a pentad $\neq$ D, the machine stops as an indication that there is a mistake, viz. there is a tape indication missing. (Like all directives, this begins with the letter D.)

If the new D is the initial letter of a non-type-indicating directive - such as a paragraph definition or an address indication (see 4.1.4) - the tape read program goes on skipping blank tape (and X's) when it has dealt with the directive, until it finally meets a (real) type indication. The skipping of blank tape then comes to an end for the time being.

DI "Read instructions"

After this type indication and until further notice (i.e. until it meets a new type indication) the tape read program expects orders and directives exclusively.

Orders are punched as they are written, from left to right; nothing is punched for blank columns.

No additional symbols are needed to separate the orders from each other (such as, for instance, Line Feed or Carriage Return; these symbols do not occur on the keyboard of the tape punch). At the beginning of each order any possible extra pentads X are skipped by the tape read program.

The page number, also the index m of special jump orders, must be punched as one pentad. The line number, if ≤ 31, may be punched as one pentad.

Further: if one inserts a minus sign between line number and page name, the line number is interpreted as a negative number in the calculation of the address (it is therefore a minus sign with retroactive effect). The final address may not become negative in this way; this restriction is not essential in the special application for which this facility was included. In orders with function letter P (0P to 3P inclusive) this minus sign must not be used.

DN "Read Numbers"

After the type indication DN the tape read program expects only directives and single length numbers until further notice. Pentads X are skipped at the beginning of each number (i.e. before the sign); in punching the numbers one uses only the digit keys 0 to 9 and the keys for the plus and minus signs and the decimal point.

After DN the assemblage program is able to handle conversions from decimal to binary system for integers and proper fractions.

All numbers start with a sign (+ or -). The sign must always be specified and it may therefore not be omitted, for example, in the case of positive numbers; on the tape the signs actually serve as points of separation between the decimal digits of consecutive numbers.

Integers consist of a sign followed by the decimal digits; non-significant zero's at the beginning may be omitted. The restriction that the absolute value of the number must be less than $2^{26} = 6710\ 8864$ is implied by the word length.

In fractions the sign is followed by the point and then the decimal digits; non-significant noughts at the end may be omitted. In view of the word length of the X1 it is rather futile (although it is allowed) to intruduce the fractions into the machine with more than eight decimals after the point. (The integer formed by the decimals after omitting sign and point must be less than $2^{52} = 4\ 50359\ 96273\ 70496$; if this condition is not satisfied the X1 stops.) An arbitrary number of zero's may be inserted between sign and point.

We point out that after DN integers and fractions may alternate without further directives. Here two different interpretations of the binary word therefore fall under one type-indication; the choice between the two possibilities is determined each time by the presence or absence of the decimal point.

DB "Read Binary words"

After the type indication DB the tape read program only expects directives and binary words as these are punched out by the X1 (see 3.9.3). As described there, each word on the tape is supplied with a parity bit; after the type indication DB the assemblage program checks the parity of the groups of symbols read from the tape.

DT "Read Type codes"

After this type indication the tape read program only expects directives and type codes (see 4.7.1).

4.1.4 Directives for the processing cycle

Five directives control the operations of the processing cycle of the tape read program.

They are:

DA "Address indication"

This directive serves to specify the storage location of the first word that follows. Hence, the letters DA should be followed by the address concerned, i.e. by the line number, the paragraph letter(s) and the page number.

DX "Skip a number of addresses"

After the directive DA the words coming from the assemblage program refer to successive addresses in the memory in the order in which they are delivered. It often happens that one wishes to leave a few locations blank - because they are to be filled in by the program; the directive "Skip a number of addresses" has been introduced especially for this purpose. The letters DX must then be followed by the number of places that are to be skipped by the processing cycle. The maximum value this number may have = 31 and it must be punched as one pentad.

DC Change of mode

A programmed switch determines whether the processing cycle reads or checks. A change of mode can be effected by means of the directive DC: if the processing cycle was busy writing it then starts checking and vice versa.

Normally programs are punched as follows. After leaving a piece of blank tape one punches a page of program. This starts with at least two directives (address and type indications) and is finished off with DCDO and a small piece of blank tape. Immediately after this one punches a complete duplicate. One checks whether no mistakes have been made visually, viz. by placing one half over the other and holding the two together against the light. When such a so-called "self-checking" tape is read into the machine, the information is introduced by the first half and, once in the machine, it is checked by the second half. Thanks to the fact that the directive DC is repeated, the processing cycle is left in the writing mode, ready to read the next self-checking tape.

The effect of the directive DC depends on the way in which the tape read program was started (see 4.1.6).

DS Stop

The directive DS indicates the end of the tape; when the tape read program meets it, it ends its activity. If the tape read program was working as an independent program (i.e. if it was started by the operator, see 4.1.6) then the X1 stops after the directive DS has been read. If the tape read program was incorporated as a subroutine (see 4.5) the directive DS ends reading from the tape but the X1 goes on working in the main program.

DE Exit

The directive Exit consists of the letters DE followed by an address. The control jumps to the address indicated. This directive can be used to let the X1 perform a non-standard operation during reading of the tape. Furthermore, it can also be used to start a program directly from the tape, i.e. without using the switches of the start address (see 2.8 and 4.9.6).

4.1.5 Remarks about the use of directives

The above directives all bring about a certain modification

in the "state" of the tape read program, and only the particular modification mentioned. In all other respects they leave the state of the tape read program exactly as it was. Thus type indications do not alter the state of the processing cycle in any way, the address indication leaves the type indication and the condition of the mode switch unchanged, etc. As a result type and address indications at the beginning of a tape may be punched in an arbitrary order.

Finally, the following points about self-checking tapes should be noted. Apart from the essential general initials, it is advisable to provide each self-checking tape with the necessary directives to make it a complete unit on its own. It is true that this costs some extra symbols, but one evades a possible source of errors in this way. This also implies that one must make it a habit to write the first paragraph letter explicitly in the address indication at the top of each page. This again, is in accordance with the normal way of using ditto's. Things like these make it easier to handle the tape and improve the general arrangement of the program in no small measure.

4.1.6 Starting the tape read program

When a standard program is started by pressing one (or more) keys of the keyboard (see 2.7.4 and 2.8), we use the term autostart (see 4.4.2). The autostarts bear the names of the corresponding keys: autostart 1 starts the tape read program as an independent program, viz.

Autostart 1: with console word positive: start tape read program in the writing mode with directive DC effective;

with console word negative: start tape read program in the checking mode with directive DC ineffective.

Autostart 1 starts the tape read program after having set the type indication to "Skip blank tape". As a result it may only be used if a piece of blank tape actually lies under the reader and the necessary directives are still to come.

When the console word is positive, the tape read program starts working in the writing mode and each time it meets the directive DC on the tape the processing cycle changes its mode.

When the console word is negative, however, the tape read program starts working in the checking mode and it goes on checking no matter how often it encounters the directive DC on the tape.

In both cases the directive DS brings the reading of the tape to an end: the machine stops after having restored the status quo of the moment the autostart was used.

If one fears, during the testing of a program, that an erroneous out-order has destroyed the initial information in one or more storage locations, one can investigate this by letting the tape be read again, but this time with a negative console word. With this application in view this autostart was so constructed that the directive DC is ineffective when the tape read program is started in the checking mode.

When the contents of a part of the memory have been punched out on a binary tape (see 4.2), one uses autostart 1 with negative console word to check whether the tape produced is correct. If the X1 does not stop during the checking of this tape, one knows that the contents of the memory have been punched out correctly; at the same time one has verified that the correct parity bit was added to each binary word when it was punched out.

4.2 The tape punch program as independent program

The standard tape punch program is able to punch out binary words. As a parity digit is punched out with every word (and this parity is checked every time the tape is read) binary tapes are not punched in duplicate; the directive DC does not appear on them.

One can let the X1 punch out the contents of a number of consecutive storage locations in binary form, using

Autostart 9: "Punch Binary Tape"

For this purpose one specifies before hand:

1) the address of the first word to be punched out in the switches of the start address (1 DO);
2) the number (> 0) of words to be punched out in the switches of the stop address (2 DO);
3) the so-called punch code in the word switches (3 DO); the punch code must be positive.

The tape punch program consists of six parts, the punch code is specified in the six least significant bits of the console word: they indicate, from right to left, whether the corresponding part must be executed (if so = 1, if not = 0). In order:

$d_0 = 1$ Punch piece of blank tape (about 19 cm)

$d_1 = 1$ Punch "DB"

$d_2 = 1$ Punch first address in the form "DA.....XO"

$d_3 = 1$ Punch the binary words

$d_4 = 1$ Punch "DO" followed by a piece of blank tape (about 6 cm)

$d_5 = 1$ Punch "DS" followed by a piece of blank tape (about 6 cm).

Normally these six switches will all be up (= 1). If the information to be punched out is, however, stored (for example) in two separate parts of the memory, one punches (for example) the first part with punch code = 15 (i.e. 001111) and then the other part with punch code = 60 (i.e. 111100). Another possibility is to use punch codes = 31 and = 62 respectively; the two parts are then separated by a piece of blank tape.

The switches used to give further specifications of autostart 9 must always be set before the key 9 is pressed; once the key has been released the X1 starts punching almost immediately. The tape punch program has then already copied the data from the switches and while the X1 punches the first part of the tape one can already set the switches for the following autostart 9.

When the whole tape has been punched the X1 stops after having restored the status quo of the moment that autostart 9 was given. It is therefore possible to stop a computation at a certain point, to punch out some relevant intermediate results by means of autostart 9, to check the tape with autostart 1 under negative console word, and then to let the X1 continue its computation by pressing the button BNA ("Begin Next Address", see 2.8) as if nothing had happened.

Remark. Naturally reading in the checking mode does not check, whether one has placed the correct first address and length in the switches. This should therefore be done with the greatest possible care.

4.3 The type program as independent program

Two autostarts enable us to let the X1 type out the contents of storage locations or of registers at any desired point of a program.

In order to type out the contents of a storage location one puts its address in the switches of the start address (1 DO). Three addresses have a special function: during the operation of the keyboard program there is a one to one correspondence between these addresses and the registers, viz.

(0 X2) ≡ (A)
(1 X2) ≡ (S)
(2 X2) ≡ (B)

If one puts one of these three addresses in the switches of the start address the contents of the corresponding register are typed out.

The autostarts in question are:

Autostart 7: "Type integer"

Autostart 8: "Type fraction" .

In autostart 7 the sign is followed by eight digits of which zero's at the most significant side are replaced by spaces. In autostart 8 the sign is followed by a unit digit, a point and then seven decimals; naturally the unit digit is nearly always = 0: however, as the type routine rounds the fraction off correct to seven decimal places, it may become = 1.

When the typing has been completed the X1 stops again after having restored the status quo; if one does not wish to type out any more at this stage, one can let the X1 continue its computation by pressing the key BNA (Begin Next Address, see 2.8). See also 4.8.4.

4.4 The keyboard program

Provided the stop switch SNA is off one can let the X1 execute different standard operations by pressing different keys of the keyboard (see 2.7.4). At the end of such a standard operation the X1 will stop again after having restored the status quo of the moment the key was pressed. (A number of special storage locations that function as working space for the keyboard program are naturally excepted).

4.4.1 Manual input of numbers

The keyboard can be used for the input of numbers by hand. This application of the keyboard, like most of the others, is not meant to be used frequently but it is nevertheless extremely useful. Using punched tape for a single number that is not known before hand is rather cumbersome!

We restrict ourselves to the input of double length numbers: for this purpose one presses the sign followed by the successive decimal digits, when necessary the point, and finally the key G once. The number introduced in this way can then be found in addresses 30 X0 and 31 X0 in double length representation: 30 X0 contains the most significant half of the number, 31 X0 contains a copy of the sign digit and the 26 least significant binary digits.

We must now distinguish between three different cases.

Integer If the point is not pressed the decimal digits are regarded as forming a whole number, which (analogous to the notation $[AS] = [A].2^{26} + [S]$) is stored in the representation [30 X0, 31 X0]. If the introduced number is less than $2^{26} = 6710\ 8864$ in absolute value, [30 X0] therefore becomes = ±0, while [31 X0] is equal to the number.

Mixed number If the point is pressed, but not immediately after the sign, the decimals are regarded as forming a mixed number, which (analogous to the notation $\{AS\} = [A] + \{S\}$) is stored in the representation {30 X0, 31 X0}. In this case [30 X0] is equal to the whole number formed by the digits before the point, and {31 X0} is equal to the fraction formed by the digits after the point. An exception to this rule is the case where the fraction lies so close to 1 that in rounding it off we get an overflow.

Proper fraction If the point is pressed immediately after the sign the decimal digits are regarded as forming a double length fraction, which (analogous to the notation $\{AS\} = \{A\} + \{S\}.2^{-26}$) is stored in the representation {30 X0, 31 X0}.

One should therefore, note that pressing "+0.137 G" and "+.137 G" gives rise to different results! In the input of single length fractions it is advisable to make it a habit to insert a nought between sign and decimal point. In the first place the fraction is then rounded off correctly, in the second place the result is then delivered to the same address (i.e. 31 X0) as single length integers.

Remark. In all cases the keyboard program demands that (ignoring the sign, and the point when present) the integer formed by the digits must be less than 2^{52} = 4 50359 96273 70496.

As soon as one is afraid of having made a mistake in pressing the keys, one presses the key H and starts again. Pressing the key H brings the keyboard program into its "neutral state".

The keyboard program, which can be in different states, is normally in what is called its neutral state. It must be in the neutral state when one begins with the manual input of numbers. If one of the signs is then pressed in, its state is then no longer neutral; the keyboard program then only returns to this state when one presses the key G or H. (The key F, which is not considered here, will subsequently also be able to restore the neutral state.)

The way the X1 reacts to the pressing of a key depends on the state in which the keyboard program finds itself. However, this does not apply to the key H: at all times its function is to bring the keyboard program to its neutral state.

4.4.2 The autostarts

One effectuates a so-called autostart by pressing a non-sign key without having pressed a sign key before hand (i.e. when the keyboard program is still in its neutral state). The autostarts bear the names of the corresponding key(s). At the moment of writing definite functions have been assigned to ten of them; the keys ., F and G are still reserved for multiple autostarts.

Autostart 0: Start the program that begins at 0 Z Z0

Autostart 1: Start tape read program (see 4.1.6)

Autostart 2: Preparation for interruption program class 6 (see 4.9.6)

Autostart 3: Transport (31 X0) ┐
Autostart 4: Transport (30 X0, 31 X0) ├ see below
Autostart 5: Transport console word ┘

Autostart 6: Complete active type-punch program (see 4.8.3)

Autostart 7: Type integer (see 4.3)

Autostart 8: Type fraction (see 4.3)

Autostart 9: Punch binary tape (see 4.2)

Autostart H: Restore neutral state keyboard program (see 4.1)

Autostarts 3 and 4 make it possible for one to fill in the number that has just been introduced manually, in an arbitrary place in the memory; autostart 5 transports the contents of the 27 word switches at the bottom of the vertical panel (see 2.8). In other words, one uses the autostarts 3 and 4 in bringing in decimal data, autostart 5 in introducing binary data.

In all three cases the destination is given by the fifteen switches of the start address at the top left of the vertical panel (see 2.8). Autostarts 3 and 5 fill the storage location indicated by these switches; in autostart 4 the most significant part is written there, while the least significant part is written in the next storage location.

As the start address and the console word correspond to (1 D0) and (3 D0) respectively, we can represent the function of the autostarts that have been mentioned as follows:

Autostart 3: $(31\ X0) \Rightarrow ([1\ D0])$

Autostart 4: $(30\ X0,\ 31\ X0) \Rightarrow ([1\ D0],\ [1\ D0] + 1)$

Autostart 5: $(3\ D0) \Rightarrow ([1\ D0])$.

The fact that, during the operation of the keyboard program, there is a one to one correspondence between 0, 1 and 2 X2 and the registers A, S and B (see 4,3), enables us to transport information to the registers. As a result (A) is modified by autostarts 3 and 5 if the destination is 0 X2, while autostart 4 then modifies (AS).

Remark. The operator who wishes to make use of this facility must know the one to one correspondence, by contrast to the special role of addresses 30 X0 and 31 X0. As the reader will have noticed, manual input of numbers could also have been described along the following lines: "To fill in a single length number at a desired address one puts this address in the switches of the start address and successively presses the two keys G 3 after the last decimal" etc.

4.5 The tape read program as subroutine

We have already seen that, if the X1 has stopped, the tape read program can be started by autostart 1 (see 4.1.6). The function of autostart 1 with a positive console word can be effectuated by a program with the aid of a special subroutine call:

6T 24 D7 0 ⇒

This subroutine has the function: "Start the tape read program in the writing mode with directive DC effective".

When the control returns to the main program below the call, (A), (S), (B) and (s_0) have been modified. However, the control does not wait for the completion of the tape reading before returning to the main program, but returns to it as soon as the X1 would have to wait for the next pentad on the tape! As soon as the following pentad can be read, the program following on the call is interrupted again and the tape read program proceeds for the time being, etc. The actual executions of the tape read program and the main program following the call are thus intertwined; during this time the tape read program restricts itself to the working space reserved for tape reading (see 4.8.5). Naturally, after every intermediate interruption the control returns to the main program with complete restoration of the status quo.

As in the case of autostart 1 the tape read program reads from blank tape up to and including DS. If autostart 1 was used to start the tape read program, the latter reacts to the directive DS by stopping the X1 after having restored the status quo (see 4.1.6). Now, however, the tape reader stops after DS, but the X1 goes on working and the reaction to the directive DS is confined to noting the fact that the tape read program has ended its activity.

The main program can inquire after this with the aid of the following condition setting subroutine call:

6T 5 D0 0 ⇒ "Tape read program Active?"

The subroutine modifies (A) and (s_0) and records the answer to the question whether the tape read program is still in action, in the condition. Before the main program uses the data just read from the tape, it can therefore verify whether the desired information has actually all been introduced with the aid of the subroutine just mentioned.

If it transpires that the tape read program is still active the main program can find out how far it has got: [28 X0] = "the so-called transport address", i.e. the address of the storage location to which the next word from the tape will refer. If the tape is a binary one and will therefore only be read in the writing mode, (28 X0) makes it possible for the impatient programmer to use information from the beginning of the tape, before the end has been read.

The most common application of the transport address is the following. If the tape contains an unknown number of words which are to be consecutively stored in the memory, starting at a known address, the number can then be deduced from the final value of the transport address after the directive DS has been read. The required number is equal to the final value of (28 X0) decreased by the address of the first word of the sequence. Of course (28 X0) must then be inspected before a new piece of tape is read in!

The second application of the transport address is reading a binary tape which lacks an address indication: before the subroutine call that starts the tape read program is given, one fills in the destination of the first word in address 28 X0. Of course this may not be done while the tape read program is still busy reading the previous section on the tape.

The subroutine call "Start tape read program in the writing mode" is equipped with so-called "semi-automatic synchronization", i.e. one is only allowed to call in the subroutine if there is no tape read program active anymore; if necessary a wait should be programmed:

```
→      6T  5  D0  0  ⇒ Tape read program active?
   Y   1T  2      A  → If so, jump back.
```

(As a rule the main program will already have checked whether the previous activity of the tape read program has ended, because it is necessary that this condition should be satisfied before the information can be used.)

If the X1 stops in the tape read program e.g. due to a failure of the tape reader, one puts the tape back with the piece of blank tape preceding the rejected section under the reader. One can usually start the tape read program anew with autostart 1. However, this is not allowed in the following two cases.

1) If the tape read program was started as independent program by autostart 1 (with the result that the X1 would stop at the directive DS) with the intention of making use of the restoration of the status quo.

2) If the tape read program was started by the subroutine call. (In this case a part of the main program following on the subroutine call has also been executed already!)

In these two cases one starts the X1 at

address 0 D9: "Restart address tape reading in the writing mode"

after the tape has been put back.

(As we have already seen, it is possible that the address indication may not appear on the tape in the second case. If so, the operator must first give the transport address at 28 X0 its initial value.)

4.6 The tape punch program as subroutine

Autostart 9 (see 4.2) is not the only means of starting the tape punch program: it can also be started by a program and it then has exactly the same facilities. The program then uses the subroutine call

6T 24 D21 1 ⇒ "Punch binary tape"

The three parameters required by the tape punch program are supplied in the registers, viz.

[A] = punch code
[S] = first address
[B] = length

When the control returns to the main program under the subroutine call, (A), (S) and (B) are modified, also (s_0), (s_1) and (0 X1). From then onwards these three addresses are at the disposal of the programmer again, although the punching has not yet been completed. As a matter of fact the actual execution of the tape punch program is intertwined with that of the main program following on the call (in the same way as that of the tape read program (see 4.5) and the type programs (see 4.7)). From then onwards the punch program restricts itself entirely to the working spaces reserved for this purpose (see 4.8.5); in autostart 9 the tape punch program only uses these working spaces and (s_0) (s_1) and (0 X1) are therefore <u>not</u> used.

In order to determine, some time after calling in the subroutine "Punch binary tape", whether its execution has been completed in the mean time, one may make use of the condition setting subroutine call:

6T 5 D1 0 ⇒ "Type-punch program active?"

The subroutine modifies (A) and (s_0) and records the answer to the question whether the type-punch program is still in action, in the condition. The term "type-punch program" is used here to refer to a program that operates either the typewriter, or the tape punch, or both (In the last case neither of the two mechanisms work at top speed continuously; a program of this kind is not included in the set of programs described here.) Tape punch and typewriter have some of their controlling apparatus in common (see 2.6.9); as a result the synchronization between the X1 and these mechanisms is governed by the same signals and the same piece of administrative program. Tape-punch and typewriter on the one hand and tape reader on the other are completely independent. As a result it is permissible to start the tape punch program as subroutine while the tape read program is still active, and vice versa.

The subroutine call "Punch binary tape" is equipped with automatic synchronization", i.e. it is permissible to give the subroutine call "Punch binary tape" while some type-punch program is still active. In that case the control automatically waits until this program has ended its activity and only then returns to the main program.

N.B. Of course the contents of the addresses that are to be punched out must be left intact until the punching out has actually taken place. The program that follows on the call "Punch binary tape" must therefore not write in these addresses, before

1) the subroutine "Type-punch program active?" has given a negative answer, or
2) a new call of a type-punch subroutine with automatic synchronization has been executed (see 4.7).

If it transpires that the binary tape is not yet finished, it is possible to let the main program determine how far it has got. The impatient programmer can use the fact that [27 X2] = the number of words still to be punched.

Finally: if one only uses the subroutine "Punch binary tape" in order to punch a piece of blank tape and/or some

directives, we need not specify the "superfluous" parameters. In particular: if $d_2 = d_3 = 0$ in the punch code, then [S] (= first address) is of no consequence; if $d_3 = 0$, the length does not have to be specified in B.

One could ask why the punch subroutine is equipped with fully automatic synchronization, whereas the tape read subroutine only has semi-automatic synchronization. The reason is that the punch subroutine may have to wait for the completion of a type subroutine (which, see 4.7, also has fully automatic synchronization), while the tape read subroutine may only need to wait for the end of previous tape reading. The question, whether the tape reading has been completed, is of great significance to the main program that wishes to use this information. When the main program wishes to start the tape read program, it will therefore in general already have verified that the previous activity of the tape read program has ended. The situation is quite different for the type subroutine: if this subroutine is presented with a number to be typed out, it is, logically speaking, immaterial to the main program at which moment the typing out of the number has been completed.

Finally we wish to draw the attention to the punch code: it controls six separate functions of the punch program. An alternative solution would have been the construction of six separate subroutines. However, at every call of a synchronizing communication subroutine, the main program and the process of communication are, for a moment, synchronized with respect to each other. The less often this occurs, the more efficient the combination of the two processes tends to be. For this reason the six functions of the punch program are combined into one subroutine, that is controlled by a punch code.

4.7 The type program as subroutine

The standard type program types single length numbers. Which decimal digits the type program derives from the binary word, depends on the interpretation of this word: the standard type program can choose between two interpretations, viz. integer and fraction. This classification only has a bearing on the position of the binary point and is independent of what the result looks like on paper: apart from this classification the programmer is free, for example, to shift the decimal point in a

fraction over a number of places during typing, or even to omit it altogether, or, on the other hand, to insert one or more points in the typing of an integer.

The way in which the type routine must type out a number is recorded in the memory with the aid of a so-called type code. The binary representation of a type code is of no significance at the moment (however, see 4.7.1); for the present it is sufficient to know that a type code always occupies one address. We will now describe the way in which the programmer specifies the type code, and at the same time explain the facilities of the type program.

4.7.1 The type code

1) Absolute value or signed number When the type code begins with an A, the absolute value is typed out, i.e. the typing of the sign is suppressed. If one omits this A at the beginning of the type code, the type program begins by printing the sign of the number.

2) Integer or fraction Next one specifies in the type code whether the binary word is to be interpreted as whole number or as fraction. There are three possibilities:

Gn ($0 \leqslant n \leqslant 8$) "Integer with n digits"

Bm ($0 \leqslant m \leqslant 7$) "Fraction with m digits"

Em ($0 \leqslant m \leqslant 7$) "Fraction with 1+m digits"

In the case Gn the word is regarded as a whole number, that is typed out in n decimal digits; the whole number must be less than 10^n in absolute value (for $n = 8$, this condition is satisfied automatically).

In the cases Bm and Em the number is regarded as a fraction and is rounded off exactly to the mth decimal place. The difference between the two forms is that in the case Em the digit before the decimal point is included, in the case Bm this is not so. In the case Em it is permissible that in rounding off the fraction becomes = 1; by contrast, in the case Bm the fraction must be less than $1-\frac{1}{2}10^{-m}$ in absolute value.

At this point, the digits which are to be typed out, have been specified in all three cases.

3) Number layout After this one specifies how the decimals just defined are to be typed out. For this purpose the digits are subdivided into, at most, three groups of consecutive digits. The number layout is described from left to right for each separate group. Per group the specification consists of three parts in the following order:

a) indication for the (possible) insertion of a symbol

b) indication for the (possible) suppression of zero's

c) number of digits in the group (≥ 1)

A symbol can only be inserted at the beginning of a group; the description of a group can begin in four ways:

. : insert point

- : insert minus sign (dash)

S : insert space

N : insert nothing .

One uses the N when one starts a new group for some reason other than the insertion of a symbol. (As every group contains at least one digit it is therefore not possible to insert, for example, a point and a space next to each other.)

The second symbol of the group description controls whether or not spaces are to be substituted for noughts. There are three possibilities:

I : imperative

F : facultative

L : last digit imperative .

By "imperative typing" we mean that all digits, including the noughts, are typed out; by facultative typing we mean that those zero's that occur at the beginning are replaced by spaces. As soon as a digit $\neq 0$ is encountered, the "transition to imperative typing takes place": from then onwards any zero's, which may occur, are typed out until further notice. If such a digit $\neq 0$ is not present, only spaces are given. The possibility L was included, because it is often desirable that this last nought should be typed out normally: the typing starts as for F, but the last digit of the group is always typed out, irrespective of the value of the previous digits.

N.B. By F and L zero's at the beginning of a group are replaced by spaces, unless the previous group was typed according to F and the transition to imperative typing had already taken place there.

The group description ends with the number of digits in the group.

4) Closing symbols When the groups have been dealt with, the closing symbols are written down. There are three possibilities:

XT : Tabulator signal

XS : Space signal

XN : Nothing .

The first set of closing symbols causes the carriage of the typewriter to move on to the next tabulator stop when the last digit has been typed. The second set of symbols only gives rise to a space after the last digit. If, however, we wish to go on typing immediately after the last digit of the number, the type code is ended by the letters XN.

In illustration we give a number of examples. We begin by pointing out that the version L could always be constructed with the aid of the two other versions, e.g.

"SL4" ≡ "SF3 NI1"

Here, however, the four digits must then be divided into two groups: without the L version the maximum of 3 groups is more apt to be an inconvenient restriction.

To type the contents of a register including its sign as a whole number facultatively, followed by a space, one uses

G8 NL8 XS .

If, however, we know that the number is positive, so that we are not interested in the sign, if furthermore we wish the eight digits to be split into two groups of four, separated by a space, and if, in addition, the carriage must move on to the next tabulator stop, then the type code takes the form

A G8 NF4 SL4 XT .

(According to this type code a million would be typed out as:

100 0000 , and <u>not</u> as

100 0 .

Here we see the reason why, after a group under F, the type program does not begin by unconditionally substituting spaces for noughts in the following group, when the latter is typed under F or L.)

With the aid of the type code

A G8 SL2 - L2 - I4 XS

one types the number 25091931 as the date

25- 9-1931 .

Insertion of horizontal dashes will as a rule be used in the typing of code numbers.

One uses the following type code in order to type out a fraction in four decimals:

B4 .I4 XT .

As fractions are rounded off by the type subroutine, both 0.3478503 and 0.3479332 will be given as +.3479 on paper.

If one must allow for the case that the absolute value of the fraction may exceed 0.99995, one uses the type code

E4 NF1 .I4 XT .

Then possible results on paper are

+ .7382
- .0031
+1.0000
+ .0000 .

The version E can thus also be used to insert a space between sign and point; if the first group had been specified by "NI1", a nought would then have been inserted between sign and point.

One often wishes to suppress or shift the point. In order to type a fraction as a percentage with one decimal, one uses, for example,

A B3 NL2 .I1 XS .

As mentioned before each type code occupies one word in the memory: they are built up by the assem lage routine under contro' of the type indication DT.

After the type indication DT the tape read program expects only type codes and directives until further notice. At the beginning of every type code extra pentads X are skipped. The type code is written down according to the above rules and is punched from left to right.

The binary representation of a type code can be derived from the following:

1) d_{26} = 0 if A
   = 1 if non A

2) $d_{25}d_{24}$ = 00 if Gn
   = 11 if Bm
   = 10 if Em

To the right of these there follow:

for Gn : 8-n zeros
for Bm : 8-m zeros
for Em : 7-m zeros

} followed by a 1.

(The number of noughts is thus equal to 8 - the number of typed decimals.)

3) Immediately to the right again follow the descriptions of the groups from left to right. Each group description consists of

3a) Two bits describing the insertion of a symbol, viz.

00 for "-"
01 for "N"
10 for "."
11 for "S"

3b) The next two bits describe the zero suppression, viz.

01 for "I"
10 for "L"
11 for "F"

3c) Thereafter, the number of digits $k \geq 1$ of the group is recorded by a string of k-1 zeros followed by a one.

4) The closing symbols are recorded in the three least significant digits of the type code, viz.

$d_2d_1d_0$ = 0 1 0 for "XT"
= 0 0 1 for "XS"
= 1 1 0 for "XN" .

The unused digits, if any, between 3) and 4) are = 0.

4.7.2 Calling in the type subroutine

The type subroutine, which is equipped with automatic synchronization, is called in with the number to be typed out in the S-register. The type code needed is stored in the location following on the call, the control returns to the main program at the address that then follows. The general form of such a call is

6T 0 D22 0	⇒	Type (S)
DT .... DI		Address for type code
⇒		Return address

A consequence of this method of giving the type code is that the subroutine jump to the type routine may, as a rule, not be skipped conditionally: for then the X1 would interpret the type code as order.

Analogous to the subroutines for tape reading and type punching, the type subroutine returns control to the main program as soon as the X1 would have to wait for the external apparatus (the typewriter in this case). Thereafter, the actual execution of the rest of the type program automatically takes place, in parts, in between that of the main program that follows the call.

When the control returns to the address following on that of the type code, this happens with complete restoration of the status quo of the moment the type routine was called in: (A), (S), (B), the condition etc. and the interruption permit are all unchanged; (s_0), which was used for calling in the type routine, is then again at the disposal of the main program.

Like the subroutine call "Punch binary tape" (see 4.6), the subroutine call "Type (S)" is equipped with automatic synchronization, i.e. one is allowed to call in the type subroutine while a type-punch program is still active. If so, the control automatically waits until the previous type-punch program has ended, and only then returns to the main program.

A secondary function of the type routine is counting the number of times it is called in per line: after the typing of every n-th number the signal NLCR is automatically given.

For this purpose the number n = "the number of numbers per line" must previously have been filled in at address 31 X2. In typing the last number on the line, the closing symbols have no effect: if, for example, the type code ends with XT, the tabulating signal is not given after the last number on the line for reasons of speed; instead the signal NLCR is given immediately.

The process of typing is completely checked. The decimals sent to the typewriter are read back out of the type relays by means of the echo orders. From these data the number is converted back to the binary number system again and the result is compared to the original number. When a discrepancy occurs the type routine automatically types the incorrect number again in the same column, but one line further down.

The type check also checks whether the given number satisfies the conditions demanded by the type code, viz. that for Gn the absolute value of the number must be less then 10^n and that for Bm unity may not be reached in rounding off.

For the purposes of checking too it is necessary that the type routine keeps count of the numbers, that have already been typed on the line. In the case of an error, the signal NLCR is given. The counter shows how many tabulator signals must then be sent to the typewriter, in order to return the carriage to the position it had at the moment the typing of the incorrect number started. With a view to this repositioning of the carriage one sets a tabulator stop at the beginning of every number, with the exception of the first, which starts at the left margin (see 4.7.4).

It is possible to call in the type routine in such a way that the paper remains blank. One can use this possibility if a column must be left open in a line. In the program one writes, for example,

```
      2S  0        A
      6T  0  D22   0     ⇒
   DT  A  GO  XT      DI
⇒
```

As a result a tabulator signal only is sent to the typewriter: a column is skipped and all precautions for synchronization are automatically taken. Of course, the column is counted!

The type check demands that (S) = 0. If we happen to know

that the two most significant digits in S are equal to each other, there is another possibility:

```
    6T   0  D22  0      ⇒ provided |{S}| < ½
DT   A  B0  XT      DI
```

⇒

4.7.3 Extra line blank

Another member of the set of type subroutines is called in by:

```
6T  10  D28  0    ⇒  Extra line blank
```

If the counter shows that the carriage is at the beginning of a new line only one signal NLCR is given, otherwise two are given and the counter is cleared. When control returns to the main program, (A), (S) and (B) have changed, for further details see the calling in of the type subroutine "Type (S)" (see 4.7.2). The subroutine "Extra line blank" is also equipped with automatic synchronization.

By deliberately changing the value of the counter one can modify the function of this subroutine. Normally [30 X2] = "the number of typed numbers on the line"; hence, when the carriage is at the beginning of a new line, this situation is characterized by [30 X2] = 0. Reversely, one can force the subroutine to act <u>as if</u> the carriage were at the beginning, by filling in a zero at this address. This enables us to break off a line before the end, and to go on typing on the next line, for example by means of the orders

```
2A   0      A
6A  30  X2              0 ⇒ [30 X2]
6T  10  D28 0    ⇒      only 1 NLCR
```

We advise the programmer to leave (30 X2) intact as long as type program is active!

4.7.4 "Tab-tape"

One should provide a so-called "tab-tape" for every program that uses the typewriter; this small piece of tape is used to fix the positions of the tabulator stops on the carriage of the typewriter. If, considering the tabulator stops from left to right, the distances between a stop and the preceding one (for the first stop: "the left hand margin") are equal to a,b,...,m, the text of the tab-tape must be as follows:

```
DN
DE  30  D27
    +a
    +b
    :
    +m
D...
```

It is essential that the directive DN should precede the directive DE. After the directive DE the program gives a signal NLCR, reads a number from the tape, gives that number of spaces and waits 1.6 seconds before reading the next number, to allow the operator time to set a tabulator stop. This process is repeated until some new directive is encountered. (Before the tab-tape is read by the tape read program, the operator must have removed all stops from the carriage. The tab-tape may not be read while a type program is active.)

4.8 Some remarks about synchronizing routines

4.8.1 The interruption permit

For both the semi-automatic synchronizing call of the tape read program (see 4.5) and the automatic synchronizing calls of the tape-punch program (see 4.6) and the type programs (see 4.7), the following holds as regards the interruption permit and the susceptibility of the X1.

In these respects the status quo at the moment of the call is fully restored when control returns to the main program. As interruptions of class 6 must be possible from then onwards, this state should imply that the X1 is susceptible and that interruptions of class 6 should be allowed by the interruption permit. Normally this is always the case.

During the execution of the communication program - both after the call and during the interruptions to follow - the state is, in these respects, as at the moment of the call, except that class 6 has been removed from the interruption permit.

As a result, interruptions from other classes that were allowed at the moment of the call, can only be prevented afterwards with due precaution as long as the communication program just started is still active. The easiest way is to make the X1 non-susceptible for a short while.

It can also be achieved by means of the interruption permit, provided that class 6 is removed from it as well. This last method will be used in interruption programs for other classes. As a matter of fact interruptions of class 6 never have priority because they can always be postponed: for class 6 the interruption signal indicates that the mechanism concerned may be used again, and not, that it must be used again within a specific period of time.

4.8.2 The end of a program

If a program starts the tape read program as subroutine, it does so because it wants to use the information on the tape: the main program will therefore detect that the tape read program is no longer active at a certain moment.

The situation is different for the tape punch and type routines. If the main program has started an output program the latter will be completed during the part of the main program that follows, unless the X1 stops prematurely! The programmer must be aware of this at the end of his program: the stop instruction should only come after the program has ascertained that it is no longer necessary for the X1 to remain active because a type-punch program must still be completed. The program should be ended, for example, as follows:

```
→   6T  5  D1  0    ⇉ Type-punch program active?
Y   1T  2      A    → if so, jump back
    7P              otherwise stop .
```

If the programmer has forgotten this, autostart 6 can save the situation (see 4.8.3).

Another way of ending the program is by means of a dynamic stop (e.g. 1T 1 A). One may prefer this method of ending the program when the X1 has to remain active with a view to other interruptions. If this is not the case, I prefer the method described above, where the 7P-order is finally obeyed, for then the control panel gives a clear indication of the state of affairs: a green light goes off and a red one goes on as soon as the machine stops, and it can further be arranged that a buzzer then starts buzzing.

4.8.3 Autostart 6

With the aid of

Autostart 6: Completion active type-punch program

the X1 is kept working as long as a type-punch program has not been completed; thereafter the X1 stops with the usual restoration of the status quo. (If no type-punch program is active at the moment autostart 6 is given, the X1 immediately stops again.)

This autostart was included with a view to the testing of programs. When the X1 is stopped at a certain point in the main program in order to type out a number for the purposes of inspection, it may happen that a type-punch program is still active. If one then uses one of the autostarts 7,8 or 9, (typing and punching respectively) then the half completed type-punch program is cut and will never be completed. If one prefers the program to perform the entire output process first, one uses autostart 6.

4.8.4 Compatibility of the keyboard program and active communication programs

If a communication program of class 6 is active, one should use the keyboard program with due care.

It is forbidden to use the keyboard for the input of decimal numbers as long as the tape read program is active. The reason for this restriction is that the keyboard program makes use of the same conversion subroutines (and the related working spaces!) as the tape read program, so that in using the first, information belonging to the second may be destroyed.

The probability that this situation will arise is fairly small, amongst others because the tape reader has a relatively high speed. Should this situation nevertheless arise, a small trick can be used to let the tape read program fulfil its task. For this purpose one presses autostart 6 "Completion active type-punch program"; as a result type-punch program that happens to be active is completed and during that time the tape is also read! Should the X1 stop before the tape reading is finished, one then makes use of the fact that an active tape read program is is continued during the punching of a binary tape: by repeated use of autostart 9 one punches small pieces of tape, until the tape reader definitely stops. If necessary the tape punch mechanism is switched off, before autostart 9 is given!

A second remark concerns the restoration of the status quo after autostarts 1,6,7,8 and 9. It is incorrect to use these autostarts when the X1 has stopped at an order of either a type-punch program or a tape read program. Should one nevertheless do so, the means of continuing the main program afterwards have been irrevocably lost. One therefore uses these autostarts exclusively if the X1 has stopped at a certain point in the main program (e.g. due to the contents of the stop address switches (see 2.8)).

The reason for this restriction is as follows. The orders of read and type-punch programs form part of the interruption program of class 6. In order to execute them the main program was interrupted at an unknown point; the interruption program, however, has recorded in the memory how and where the main program is to be continued at the end of the interruption. This record, which holds information that is vital for the continuation of the main program, would be destroyed by the incorrect use of the autostarts as described in the preceding paragraph: restoration of the local status quo is necessary, but not necessarily sufficient, to be able to continue the computation. Fortunately there is little reason to use the autostart mentioned at any point other than in the main program.

4.8.5 Reserved storage locations

A number of storage locations in the living memory are reserved as working space for interruption programs. The moment at which an interruption program is executed, is, to a large extent, undefined with respect to the main program. Hence these storage locations are unsuitable for the storing of information occuring in the main program.

The communication program under discussion here uses the last ten addresses of pag. X0 and the whole of pag. X2 (see also Appendix 3). Pag. X1 is reserved for ordinary subroutines; because it is known when these subroutines uses these addresses, the main program can make use of them too. (Address 0 X1, it is true, is used in the call of the automatic synchronizing subroutine "Punch binary tape", but it is free again as soon as control returns to the main program!)

During tape reading the memory must hold a table of the first addresses of the pargraphs. In this table each

chapter occupies thirteen consecutive addresses; per chapter the paragraph names, and the chapters themselves, occur in the order

Z E F H K L R S T W U Y N .

The beginning of the paragraph table is fixed for every installation; as soon as one knows which chapters are used in a program, one therefore knows from which point onwards the memory is free.

Pag. X3 et seq. are reserved for interruption programs of other classes. The paragraph table follows on these reserved storage locations. The beginning of the table of paragraph names is recorded in addresses 0 D16 and 27 D16. The address 0 X5 can thus be found here when X3 and X4 are reserved for additional interruption programs. By changing (0 D16) and (27 D16) one shifts the paragraph table to another position in the memory.

The communication programs dealt with here occupy pages D0 up to D27 and part of D28 in the dead memory. Address 0 D0 ≡ 24576 (= 3.2^{13}); in binary form 0 D0 ≡ 11000 00000 00000.

The interruption jumps of classes 1 to 7 can be found at addresses 1 D16 to 7 D16.

4.9 Description of synchronizing subroutines

4.9.1 Introduction

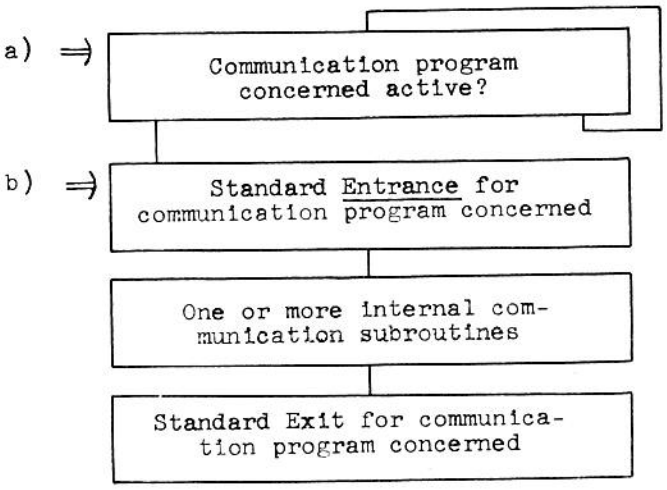
The use of automatic synchronizing calls of the type-punch programs can imply a loss of potentiel computing time if two such calls succeed each other too quickly: when the second call is encountered the control waits, if necessary, until the previous type-punch program has been completed before actually starting on the new type-punch program. Only when the latter has been started does the control return to the main program. (In the discussion that follows we restrict ourselves to the case where the X1 does not have to attend to other mechanisms simultaneously: if, for example, a tape read program were active at the same time, a large part of the waiting time caused by the typewriter could be used by the X1 to continue with the tape read program.)

If the time required by the typing is so much in excess of the total time necessary for the computation that the typewriter nevertheless operates at full speed, this loss

in computation time is fictitious. Real loss in total time only occurs when the X1 is unproductive during a "concentrated" number of calls of the type-punch programs, while the typewriter is not active at a later stage, because the X1 is still busy forming the next result. In the following we will describe how a computation of this type can be speeded up by means of a somewhat different arrangement. We should point out, however, that under the most favourable circumstances the total speed can at most be doubled, viz. when the times required for typing and computing are equal. This, then, is the only case where the X1 and the typewriter can both be productive continuously.

4.9.2 General structure of synchronizing communication programs

In the following the term communication program refers to either a tape read program or a type-punch program. The diagram given below shows an automatic synchronizing communication subroutine: without the waiting cycle at the beginning it would be equipped with semi-automatic synchronization.



a) Call with automatic synchronization
b) Call with semi-automatic synchronization

When the subroutine is called in, the X1 is susceptible and class 6 is allowed by the interruption permit; in call a) the interruption can therefore take place and a previous communication program of the same kind, which may still be active, is continued and therefore completed sooner or later. We now restrict ourselves to call b) as shown in the flow diagram.

The synchronizing communication subroutine starts with a standard entrance and ends with a standard exit. The first consists of three orders, the last of one.

The standard entrance begins by removing class 6 from the interruption permit. This is done because the program soon changes into an interruption program of class 6 and in every interruption program interruptions from its own class are prohibited.

The next operation concerns the link, for this must not be used at the end of the communication subroutine but as soon as the X1 would have to wait for the apparatus. The corresponding (s_m) which refers back to the main program is placed in the A-register for this purpose, and is thus given to a preparation subroutine as a parameter.

The call for this preparation is the third and last order of the standard entrance; the function of the preparation subroutine is twofold. In the first place the link given in the A-register is recorded amongst the administrative data belonging to the interruption program of class 6; in the second place a note is made of the fact that the communication program concerned is (by definition!) active from now onwards. On account of this note the preparation subroutine exists in duplicate.

The standard entrance of a tape read subroutine takes the form

```
0Y  64  XS          Remove class 6
2A  8+m XO          link ⇒ (A)
6T   7  D0  14  ⇒   Preparation tape read program .
```

The standard entrance of a type-punch subroutine takes the form

```
0Y  64  XS          Remove class 6
2A  8+m XO          link ⇒ (A)
6T   8  D1  14  ⇒   Preparation type-punch program .
```

The preparation subroutines change (S).

Remark As soon as class 6 has been removed from the interruption permit the 6-th interruption can no longer take place. For this reason (s14) is temporarily at the disposal of subroutine jumps and will be used exclusively between the standard entrance and exit of communication programs (see 3.6).

The standard exit of a communication subroutine consists of one jump order; as a result a note is made that the activity of the program in question has been ended again. Accordingly the exit also has two forms:

```
    2T  12  D0  A   ⇒ Exit tape read program
and
    2T  13  D1  A   ⇒ Exit type-punch program .
```

In the first instance control goes to the central administration program of class 6; as the communication program has now been completed, the control (finally) goes back to the main program.

The communication subroutines that may be called in between the entrance and exit, must be specially adapted to this situation. A subroutine having the structure required is called an internal communication subroutine.

A clear illustration of an automatic synchronizing subroutine is "Punch binary tape", which can be found at address 24 D21 et seq. During the waiting cycle the contents of the A-register (the punch code) is stored at address 0 X1 (see 4.9.3).

4.9.3 The calls of the internal communication subroutines

The internal tape read subroutine (the so-called assemblage subroutine) may only be called in between the standard entrance and exit of a tape read program (see 4.9.2). It is therefore only called in when class 6 is not allowed to interrupt and is accordingly called in by using s14.

The subroutine has no synchronizing analogue. We only mention the ordinary way of calling it in:

```
→   6T   0  D10  14  ⇒ Assemblage subroutine
⇒   2T  15  X2       ⇒ Point of return after det. of direct.
⇒   1T   3        A  ⇒ Point of return after proc. directive
⇒                    Point of return with (S) = new word
```

The assemblage subroutine is a fairly complicated piece of program; amongst others this is shown by the fact that control returns to one of the three addresses that follow on the call, depending on the kind of information that was encountered on the tape.

As soon as a directive is detected on the tape, the control comes back to the first point of return, this directive not having been effectuated yet. (The symbol that follows D can be found in A, the next symbol has also been read and can be found in S.) If control finds the order 2T 15 X2 here - as shown above - then the assemblage routine goes on to process the directive (and more symbols are read if necessary). Finally the link of the assemblage routine is increased by 1 and used again so that the control now comes back at the second point of return; as a rule it will find a jump order here which directs it back to the call of the assemblage routine. This arrangement of different points of return was deliberately chosen to enable the user of the assemblage subroutine to detect the occurrence of directives on the tape and to take special measures before or after their processing, should he wish to do so. A simple application can be found in the program processing the tab-tape, that can be found at address 30 D27 et seq. The number of spaces to be given is read from the tape by the assemblage subroutine, but when a directive occurs, control is redirected to the processing cycle (see 4.7.4). Further applications fall outside the scope of this description.

However, when the assemblage subroutine is to deliver a new word, its link is increased by 2 before it is used: control comes back at the third point of return with the new word in the S-register. When the assemblage routine is called in repeatedly, the words are delivered in the order in which they are normally stored in the memory.

In the above scheme of calling in the assemblage subroutine, all directives pass "unnoticed". The consequences of the directives DA, DX and DC are restricted to the processing cycle of the tape read program and have no effect on the assemblage proper. The type indications have their full effect and the paragraph definitions DP are, in the first instance, only effective in as far as the paragraph names specified are actually used by the assemblage. The directive DS ends the activity of the tape read program in the ordinary way, i.e. it effectuates the standard exit 2T 12 DO A (see 4.9.2) of the tape read program. As a result, the standard exit of the tape read program need not be programmed explicitly in a special tape read program

if the form of the latter is such that its activity is to be ended by the occurrence of DS on the tape. One programs the standard exit explicitly if one wants the activity of the tape read program to come to an end for reasons other than the occurrence of DS, e.g. when a fixed amount of data have been read in.

The use of this subroutine demands two fold preparation. For, when the subroutine reads the tape, this is done in the way that is specified by the last type indication. Another way of saying that the assemblage subroutine "remembers" the last type indication, is that the assemblage subroutine can find itself in different "states", in this case in as many states as there are type indications. This, however, implies that the user is obliged to bring it into a definite state before it is used for the first time. We will assume that initially there is a piece of blank tape under the reader, and that therefore the "current" type indication must be set to "Skip blank tape". (The current type indication is specified by a characteristic address stored in 18 X2; for "Skip blank tape", for example, the value of this address is = 7 D7.)

Furthermore, in the assemblage of many units of information it is essential that the assemblage routine has already noted the first pentad of the next information unit. (Thus the end of a number, for example, is indicated on the tape by the first symbol that does not belong to it, i.e. as a rule the sign of the next number.) As this necessity is the rule rather than the exception, the assemblage routine consistently assumes that each time a new unit must be assembled, its first symbol has already been read (and can be found at address 24 X0). We are therefore obliged to supply the tape read subroutine with a "suitable" first pentad before it is first used. We can use 31 (= X, Erase) for this purpose. The entire preparation could therefore be performed by

```
2S   7  D7  A ]  set current type indication
6S  18  X2    ]  to "skip blank tape"
2S  31      A ]  simulate
6S  24  X0    ]  first pentad = X .
```

The subroutine "Punch binary tape" also exists in the form of an internal subroutine, be it with somewhat less elegant conventions for specifying the parameters, than in the case of automatic synchronization, viz.:

6T 25 D19 14 ⇒ Internal subroutine "Punch binary tape"

with [26 X2] = punch code
[28 X2] = first address
[B] = length .

The addresses 26 X2 and 28 X2 are working spaces for nearly every type-punch program: therefore the parameters concerned may only be filled in "just before hand" (see addresses 24 D2; et seq.) For further details we refer the reader to the description of the synchronizing subroutine (see 4.6 and 4.2); however, the words (s_0), (s_1) and (0 X1) are not modified by the internal routine.

Next we mention one other internal punch subroutine (but without synchronizing analogue):

6T 2 D14 14 ⇒ Internal subr."Punch (S) in binary form" .

When control returns under the call (A) and (S) have been modified, while (B), condition, etc, remain unchanged. The subroutine punches (S) as binary word (parity bit included) in six pentads (see 3.9.3).

We now give two internal subroutines for typing.

6T 18 D22 14 ⇒ Internal subroutine "Type (S)"
DT . . . . . DI Address for type code
⇒ Point of Return .

The internal type subroutine also types the contents (S) according to a type code; this is supplied as program parameter in the same way as to the synchronizing subroutine. The control returns at the address below the type code with (S) and (B) as at the moment of calling in. The A-register contains the type code actually used, the condition gives the answer to the question "Was this the last number on the line?". (If this was the case, the signal NLCR has already been given.)

By contrast to the automatic synchronizing call, the X1 has in this case completed the typing when it returns under the type code. If desired, the contents of the A- and S-register make a final check on the typing possible. As a matter of fact the type routine checks the typing of (S). As, however, the S-register is not supplied with a parity bit, we have - strictly speaking - no guarantee that the number supplied to the type routine was equal to the number we wished to type out, unless we

make a check afterwards. For similar reasons, the final contents of the A-register are the type code actually used.

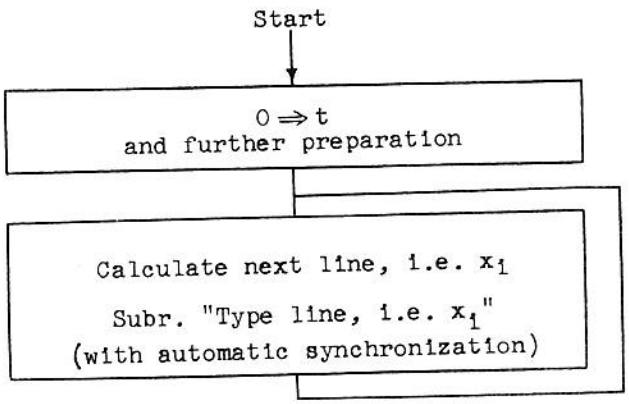
The second internal type subroutine is

6T 4 D19 14 ⇒ Internal subroutine "Extra line blank"

The function of this subroutine is the same as that of the automatic synchronizing analogue (see 4.7.3). When the control returns, only the contents of the B-register are unchanged. The condition gives the answer to the question "Has only one signal NLCR been given?" In this case too, the function of the subroutine can be modified by changing the count = [30 X2].

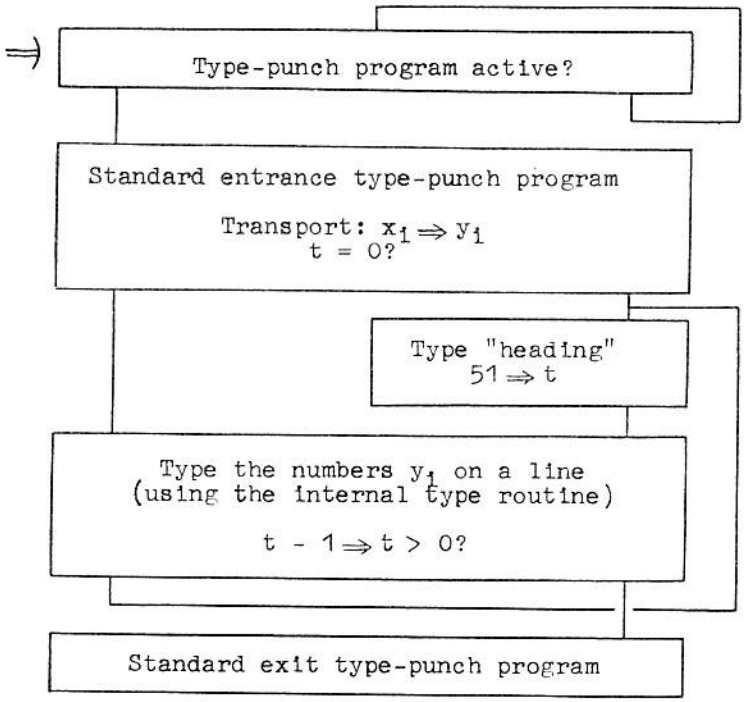
In illustration, we give an example of the use of the internal type routines, in which the entire layout administration is dealt with by the synchronizing program. Let the computation form its results one line at a time and offer them to the type program. This type program must see to it that fifty lines are typed per page and also a copy of the first line of the following page. A "heading" must be typed at the top of each page.

In the following flow diagram of the computation we assume that the table starts at the top of the page.



The test for the end of the table has been omitted in the above diagram. The results of the computation are stored per line at a number of addresses, reserved for the variables x_1; these addresses are to be regarded as

working spaces for the computing main program and as "parameter addresses" for the subroutine "Type line", which is now given.



The test "t = 0?" was introduced in order to type the very first line only once. The transport $x_1 \Rightarrow y_1$ is necessary because the one line still has to be typed while the main program is already engaged in the calculation of the new values x_1 for the following line. The addresses for the numbers y_1 are to be regarded as working space belonging to the type-program; it is strictly forbidden territory for the main program!

4.9.4 Structure of the internal communication routines

In internal tape read programs the tape read order (see 2.6.6) need not occur explicitly. In order to obtain the value of the new pentad we make use of:

6T 15 D0 14 ⟹ Tape read program "Wait switch" (LWW)

Control returns under the call with complete restoration of the status quo. The subroutine investigates whether the tape read order can be executed at this moment without delay. If so, control returns immediately under the call; if not, the tape read program is temporarily stopped and a main program (viz. that following on the synchronizing call) is continued, until the tape read order can now be executed again without delay. The main program is then interrupted and the tape read subroutine is continued under the call LWW. When the control returns under the call of LWW the following pentad has, however, already been read and stored in address 24 X0. The tape read program is therefore organized as though the X1 were equipped with a tape reader that has to be stepped up explicitly (by the subroutine call LWW), after which the pentad can be read repeatedly (being the contents of 24 X0).

Internal type-punch programs are organized somewhat differently. Here the communication orders concerned (see 2.6.7 and 2.6.8) are actually used. In using these orders, which place the flexibility of the elementary code of the X1 entirely at the disposal of the programmer again, one rule must be strictly obeyed: the X1 must have executed the subroutine

6T 15 D1 14 ⟹ Type-punch program "Wait switch" (TPWW)

once and only once before every type or punch order. As above, control returns under the call with complete restoration of the status quo, but it only returns as soon as the next type-punch order can be executed without delay. It is not necessary that the type-punch order now allowed follows the call TPWW immediately: a number of orders may be executed first but <u>not</u> the call for one of the internal type or punch routines mentioned (see 4.9.3), for these, in their turn, contain a call TPWW before the first type-punch order. In that case the necessary alternation would be disturbed.

The echo order, as used in the checking type subroutines, gives rise to a minor complication, for the echo order may only be given when the next type-punch order could be executed without delay. To obtain the echo of the last digit of a number an additional call TPWW must therefore be given. If no further layout signal is sent to the typewriter now, we can restore the disturbed alternation by the call

6T 24 D1 14 ⟹ "Obliterate TPWW" .

This subroutine call, which destroys (S), brings the central program for class 6 back into the state it would have had if a type-punch order had been executed last. Now a TPWW can come again without ill effect (or, as the case may be, the standard exit of the type-punch program).

4.9.5 The interruption of class 6

When the interruption of class 6 takes place, the subroutine jump (stored at address 6 D16)

6T 26 D1 14 $\frac{1}{=}$)

is executed. (The only difference in the execution of the interruption subroutine jump is that the increase in the order counter T is suppressed and that the X1 automatically becomes unsusceptible.)

The interruption only takes place if class 6 is allowed by the interruption permit, if the X1 is susceptible, and if at least one of the two interruption signals of class 6 is present. Which interruption signals are present, becomes apparent when the 6-th class word is read (into A by the order 6Y 4 XP):

$d_0 = 1$ interruption signal from tape reader present,

$d_1 = 1$ interruption from typewriter or tape punch present.

When these signals are absent, the corresponding bits are = 0; the more significant bits of the class word are always = 0.

As soon as the class word has been read out the interruption signals detected cease to exist (see 2.7.3). For class 6, however, this is not the only way of ending the interruption signals. Should a tape read order be given while the corresponding interruption signal is still present, the latter disappears; a new interruption signal only appears when, following on the last tape read order, a new one can be executed again without delay. The same applies to the other interruption signal with respect to the tape punch and the typewriter. (In the communication program discussed here, we do not make use of this property.)

Furthermore the communication orders are equipped with an automatic blocking, should the control encounter two

such (similar) orders too quickly after each other, i.e. the second order is blocked as long as the interruption signal is still to come. Only when this moment has arrived, the execution of the order is completed, and the previous communication order does not engender an interruption signal.

The communication program makes use of this automatic blocking in its reaction to the tab-tape (see 4.7.4), where a number of space signals are sent to the typewriter in the middle of the tape read program (see address 30 D27 et seq.). It is, however, not our intention that extensive use should be made of this blocking; it is primarily an electronic safety measure to protect the apparatus in the case of program errors. Apart from the fact that this use of the automatic blocking may imply a loss of useful computing time, there is another complication. When any interruption signal has arrived, the actual interruption is (at least) postponed until the current order has been completed. Once orders are introduced that may take 100 ms, there is a risk that an urgent interruption must be postponed for the same period of time. This could be fatal. During the reading of the tab-tape, however, these difficulties do not arise.

The situation that one of the three mechanisms would accept a next communication order only with some delay, is characterized under all circumstances, by the fact that the corresponding interruption signal is still to come. The central program for class 6 keeps a record of this situation in the four least significant bits of (26 X0) = "state record class 6". Numbering the digits from right to left, we specify:

d_0 = 0 a tape read order can be executed immediately

= 1 the program will postpone the execution of the next tape read order until the X1 has received an interruption signal from the tape reader

d_1 = 0 a type-punch order can be executed immediately

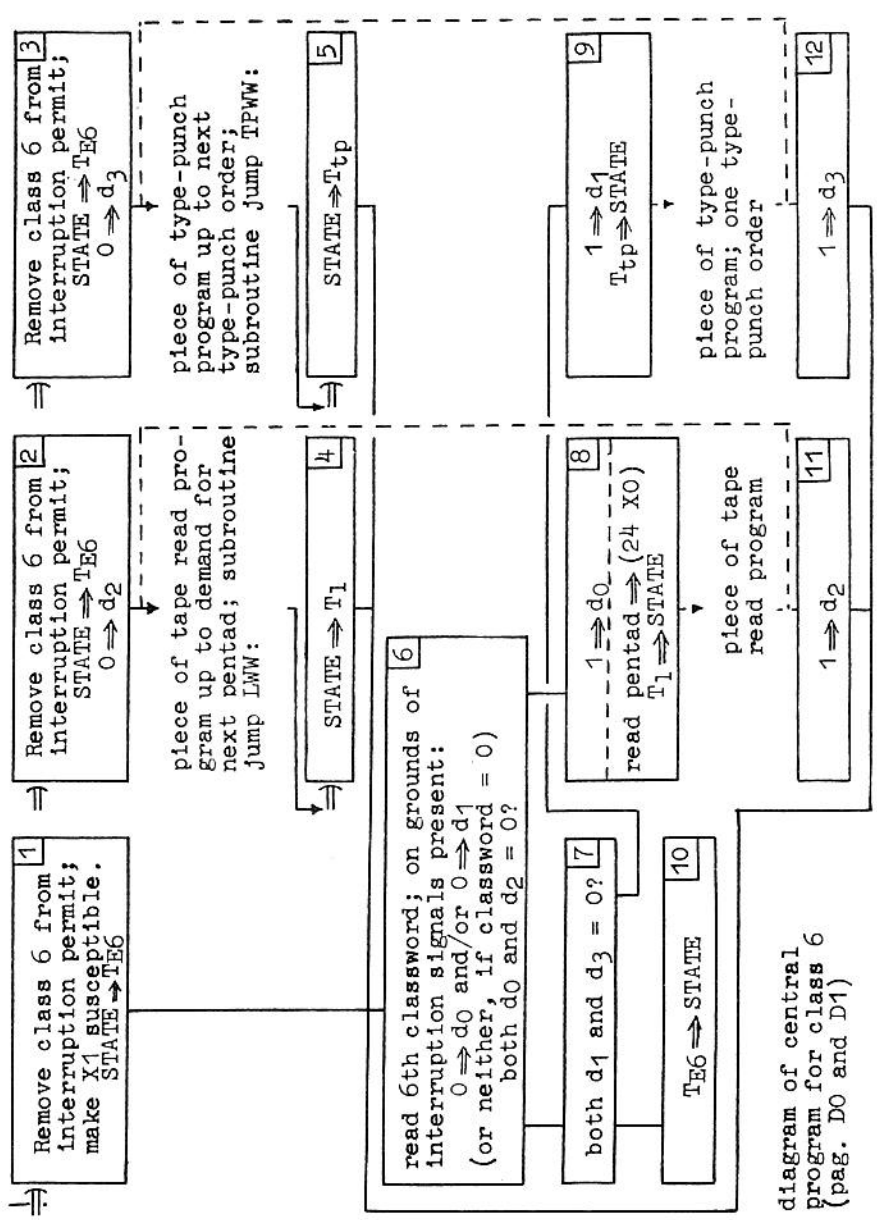
= 1 the program will postpone the execution of the next type-punch order until the X1 has received an interruption signal from the typewriter or the tape punch

d_2 = 0 there is a tape read program active

= 1 no tape read program is active

d_3 = 0 there is a type-punch program active

= 1 no type-punch program is active .


1
Remove class 6 from interruption permit; make X1 susceptible. STATE ⇒ TE6
2
Remove class 6 from interruption permit; STATE ⇒ TE6 0 ⇒ d2
piece of tape read program up to demand for next pentad; subroutine jump LWW:
3
Remove class 6 from interruption permit; STATE ⇒ TE6 0 ⇒ d3
piece of type-punch program up to next type-punch order; subroutine jump TPWW:
4
STATE ⇒ T1
5
STATE ⇒ Tp
6
read 6th classword; on grounds of interruption signals present: 0 ⇒ d0 and/or 0 ⇒ d1 (or neither, if classword = 0) both d0 and d2 = 0?
7
both d1 and d3 = 0?
8
1 ⇒ d0
read pentad ⇒ (24 X0)
T1 ⇒ STATE
9
1 ⇒ d1
Tp ⇒ STATE
10
TE6 ⇒ STATE
11
1 ⇒ d2
piece of tape read program
12
1 ⇒ d3
piece of type-punch program; one type-punch order
diagram of central program for class 6 (pag. D0 and D1)

Besides these four bits, the central program for class 6 handles three "states" of the machine.

T_{E6} = the state of the machine in the external program at the moment that it was interrupted by a signal from class 6.

T_1 = the state of the machine in the tape read program where it was broken off by LWW.

T_{tp} = the state of the type-punch program when it was broken off by TPWW.

These "states" each refer to the contents of the registers A,S and B together with all that is recorded in the link by a subroutine jump, i.e. the contents of the order counter, the interruption permit, the condition, etc. (see 2.7.2). For the recording of each of these three states, four specific addresses are reserved in the memory.

The operation that is denoted in the diagram by "STATE $\Rightarrow T_1$" only takes place in the central program after entry via a subroutine jump: the current contents of the registers A,S and B, and also the link just formed, are filled in in the four reserved storage locations. The inverse operation is indicated by "$T_1 \Rightarrow$ STATE"; here the status quo is restored in four orders, viz. a 2A, a 2S and a 2B order and finally a restoring jump (see 2.4.3 and 2.7.2).

Block 1 describes the beginning of the interruption program of class 6; here the fourth word of the "state" is derived from the link formed by the (inserted) interruption subroutine jump. Next class 6 is removed from the interruption permit. (This is a particular case of "the class itself and the classes of lesser priority": every other class has priority above class 6.) After this the X1, which became non-susceptible due to the execution of the interruption jump, is made susceptible again: from this point onwards interruptions from all other classes may take place again.

Block 2 represents the standard entrance of a synchronizing read subroutine, block 3 that of a synchronizing type-punch subroutine. The link to be filled in in the fourth word of T_{E6} is the one formed in the execution of the subroutine jump from the main program to the synchronizing subroutine: it is the link that is given in A as a parameter to the corresponding preparation subroutine (see 4.9.2). (N.B. In these blocks, as in the standard program, the specification of T_{E6} does not automatically

include the registers A,S and B; as a result, control will in general return to the main program under the call with the contents of the registers A,S and B changed.) As the program is on the point of becoming interruption program of class 6, class 6 must be removed from the interruption permit and this must be done before T_{E6} is filled in! Finally d_2 or d_3 is made = 0. From now onwards a read or type-punch program is, by definition, active.

In the diagram only the central program for class 6 is enclosed by lines. Some piece of communication program is represented immediately below blocks 2 and 3. The only necessary requirement is that it is executed with class 6 removed from the interruption permit, and that the subroutine call LWW or TPWW respectively is executed once before the first contact with the apparatus concerned.

In blocks 4 and 5 a record is made of the state of the corresponding communication program. For, we have arrived at the point where the machine will decide whether the communication program can proceed without delay or not.

In block 6 the inverted class word is collated with the "state record class 6" = (26 X0). When an interruption signal has been read out, there is a 1 in the class word, i.e. a 0 in its inverse, and the digit (d_0 or d_1) of the state record therefore becomes = 0.

Reading of the tape must be postponed if $d_0 = 1$, for in that case an interruption signal from the tape reader is still to come. Once it has arrived, it will be read out sooner or later in block 6, and d_0 will become = 0 as described above. At the end of block 6 the question is asked whether d_0 and d_2 are both = 0. (This is done by collating the state record with the number +5 and asking whether the result equals zero.) As a matter of fact the X1 will only continue with tape read program if a double condition is satisfied: apart from the fact that it must be possible to read the next pentad immediately, it must be of interest! If these conditions are both satisfied the control proceeds to block 8.

In block 8 a pentad is read and copied into 24 X0. In addition, the digit d_0 is made = 1 in order to prevent an affirmative answer to the question at the end of block 6, before a new interruption signal from the tape

reader has arrived. Finally, the substitution "$T_1 \Rightarrow$ State" causes the tape read program to be continued below the last LWW executed.

If the question at the end of block 6 received a negative answer - which can happen, for instance, when two calls LWW succeed each other sufficiently quickly - then control goes to block 7. Here the analogous question is asked with respect to the type-punch program (by collating the state record with +10). In the case of an affirmative answer the control goes to block 9.

The function of block 9 is analogous to that of block 8, only the actual execution of the communication order is left to the type-punch program that is now to be continued. In the central program of class 6 this order is anticipated by the substitution $1 \Rightarrow d_1$. If this anticipation should be incorrect, then the type-punch program must perform the call "Obliterate TPWW"; its function is to substitue $0 \Rightarrow d_1$ (see 4.9.4).

If the question in block 7 also has a negative answer, then block 10 follows. The substitution "$T_{E6} \Rightarrow$ State" results in the continuation of the external program. At this moment class 6 re-enters the interruption permit. While a communication program is active, control can only arrive in block 10 if the corresponding interruption signal was still to come.

After block 10 the period of time that elapses before that interruption signal arrives, is utilized by the external program; as soon as the interruption signal has come, control jumps to block 1 and the communication program is continued.

Blocks 11 and 12 are merely the standard exits for tape read and type-punch programs respectively.

In this scheme we have made use of the fundamental possibility of reading out interruption signals that did not result in actual interruptions (because the interruption permit did not allow them).

4.9.6 Autostart 2 and the directive DE

From the preceding paragraph it is apparent that the interruption program requires some preparation. For,

if one of the bits d_0 or d_1 of the state record of class 6 happens to be = 1 without the corresponding interruption signal actually being due to come, then the corresponding communication program never starts working! When the X1 is switched on, no interruption signals are present: the state record of class 6 must be set accordingly. This is done with the aid of

Autostart 2: Preparation class 6

The state record of class 6 = (26 X0) is given the value +12, i.e. no communication program is active and the X1 is not to wait for interruption signals from class 6. Furthermore +12 is filled in (for the time being) at addresses 30 X2 and 31 X2 (= current number count per line and = number of numbers per line, respectively); this also happens at the three addresses that are reserved for the contents of the registers in T_{E6}.

It is immaterial what is filled in in these last addresses, as long as something is filled in there; switching the machine off or on may have disturbed the parity of the words in the memory and this should be restored. Finally, a constant is filled in at address 25 X0. By modification of this constant it is possible to insert a translation program between the reading of a (physical) pentad and the delivery of a symbol in 24 X0 (see 4.9.7). Autostart 2 sets the tape read program to "no translating".

Remark Once the machine is switched on, but before autostart 2 is used, the keyboard program should be brought to its neutral state by means of autostart H.

In this connection we should like to point out that the directive DE leaves the tape read program active. The directive DE can therefore be used for an extension of the facilities of the tape read program, after which the tape read program can be continued. If one wishes to use the directive DE at the end of a program tape in order to start the calculation immediately, then the program should start with

```
2A  12      A       ]set interruption program class
6A  26  X0          ]6 to neutral state
OT  21  D21     P     X1 susceptible and all classes
                                          allowed
```

The constant in address 21 D21 has zero's in the address portion, as a result the OT-order does not jump. The other bits ensure that all classes are included in the interruption permit and that the X1 is made susceptible. (This is the normal state of the X1.) At the same time the condition becomes affirmative, the last sign positive and the overflow indication is cleared.

4.9.7 The possibility of translation

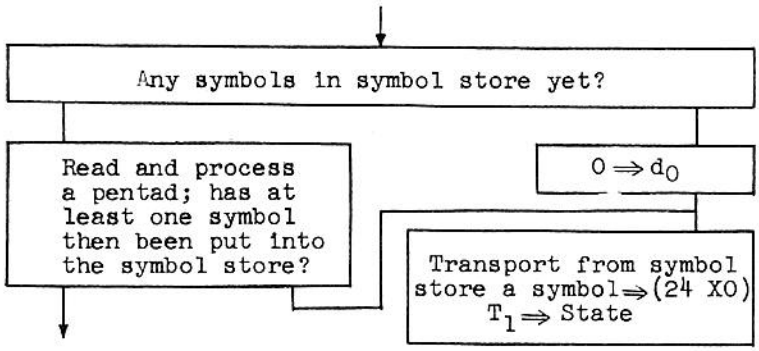
The central program for class 6 (see 4.9.5) is stored in the dead memory and can therefore not be altered by the programmer. There is, however, a possibility of "escape" in block 8; after the operation $1 \Rightarrow d_0$ the X1 performs the jump order 2T 25 X0. Normally address 25 X0 contains a constant, viz. the address 18 D1, which leads to block 8 being continued as described above. By filling in a different address at 25 X0 one can, amongst others, replace the operation "read pentad $\Rightarrow$ (24 X0)" by a more complicated process, in other words provision has been made for pentad translation.

In the following the configurations read from the tape will be denoted by "pentads"; for the purpose of distinction the numbers in address 24 X0 that are consecutively offered to the tape read program will be denoted by "symbols".

The simplest translation program derives (from a table for example) a definite symbol from each pentad. We will now describe the structure of a more complicated translation program which meets the requirement that it should be possible to derive more than one symbol from one pentad and, conversely, to derive only one symbol from a number of pentads. The word (25 X0) should equal the first address of the translation program.

This translation program can find itself in different states and therefore demands a suitable preparation. The various states of the translation program can sometimes be characterized very efficiently by different values of (25 X0).

The jump to (25 X0) also enables us to call in the assemblage routine (see 4.9.3) and at the same time



to have a list of the individual pentads used at our disposal when control returns under the call. For a last application see 4.10.

4.10 Mutually synchronized input and output

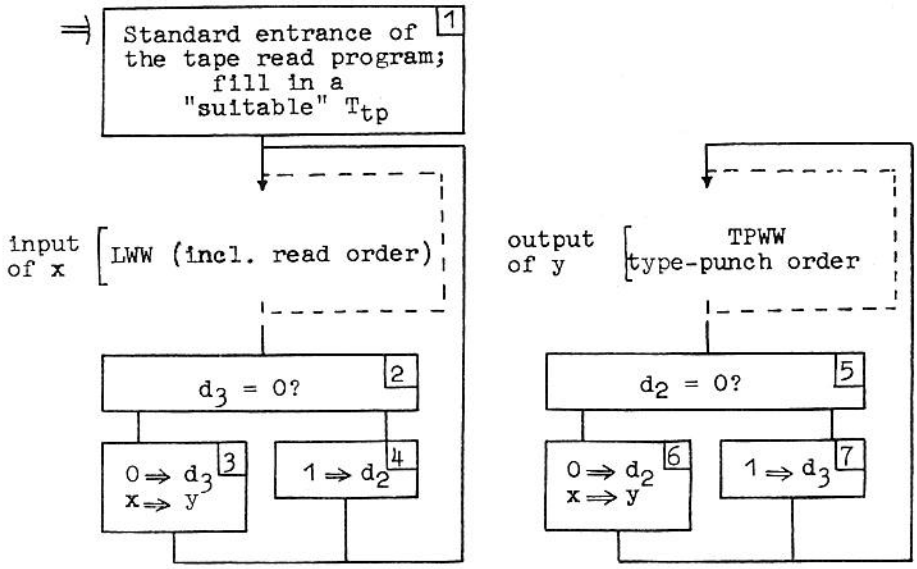
We have described a set of communication programs as acting in conjunction with a certain main program, that may want to use information provided by the input and may want to print or punch the results. The synchronization always covers the main program on the one hand, and one or both communication programs on the other hand. In this interplay the main program may have to wait for the completion of a communication process, conversely the communication programs will only be started when certain points in the main program have been reached. In the first instance input and output are asynchronous with respect to each other.

They may be coupled indirectly, for instance when a certain result to be typed or punched out requires a number of data from the input for its construction. Until now such coupling was accomplished by the main program!

Our next problem is to construct a program that reads a tape and types and /or punches out data depending on what has been read. One of the simplest examples is merely reproducing a tape. Another example is typing out the binary words from a tape in decimal form. In the latter case a number of symbols to be typed out

are derived from a group of consecutive pentads each time.

We now demand that it must be possible to execute this program simultaneously with any other program; at first we restrict ourselves to the case that the latter program does not use any of the mechanisms of class 6.



The program is constructed as a (semi-automatic) synchronizing communication subroutine. Because it must be possible to combine this program with any program, we cannot expect the subroutine jump to block 1 to occur in the latter program. In order to start the above communication program, one stops the X1 at a point in the main program, places the subroutine jump to block 1 in the word switches, and lets the X1 execute this order once by means of the key "DO" (see 2.8). Using the start key BNA one lets the computation proceed. It is necessary that one should have stopped the X1 at a point where use is not being made of the link which is given a new value by the subroutine jump in the word switches! Furthermore, we assume that the X1 is then in the normal state, i.e. susceptible and all interruptions allowed by the interruption permit. Finally the main program must leave the addresses of the communication program (and its working spaces!) intact.

The diagram shown by blocks 1 to 7 omits some details; in particular it does not show how the process ends.

In block 1 the standard entrance of the tape read program takes place; in addition the four addresses reserved for T_{tp} are given such values that the first operation "$T_{tp} \Rightarrow$ State" causes the type-punch program to start its activities correctly. (It is assumed that these values are independent of information still to be read. If necessary, the type-punch program can purposely be arranged in such a way that this requirement is met.)

The information read from the tape denoted by "x"; this is handed over to the type-punch program, which types or punches "y", by the transport "$x \Rightarrow y$". Possible additional processing of this information (translation etc) may take place either when x is formed or when y is given to the output; these operations are not indicated in the diagram either. It is essential that the storage space for x and that for y does not overlap: while the type-punch program sends one unit of information from (the addresses for) y to the output, the following unit of information is being built up in (the addresses for) x by the tape read program.

After block 1 a new x is contructed from one or more pentads. Thereafter the program investigates whether $d_3 = 0$ in the state record of class 6 (see 4.9.5), i.e. in block 2 the question "Type-punch program active?" is asked. (The asking of this question, i.e. the analysis of the corresponding digit of (26 X0) must be written out in full, and may <u>not</u> be performed with the aid of the subroutine call: 6T 5 D1 0 $\Rightarrow$"Type-punch program active?" as we may not use 8 X0 without precautions!) To start off with, the question in block 2 receives a negative answer.

In block 3 the substitution $0 \Rightarrow d_3$ makes the type-punch program active; the transport $x \Rightarrow y$ offers the information for the output to the type-punch program, and the tape read program proceeds to build up the next information unit x.

If the question in block 2 received an affirmative answer, this meant that the output of the previous y was not yet complete. Accordingly the transport $x \Rightarrow y$ does not occur in block 4, where the substitution $1 \Rightarrow d_2$ temporarily stops the activity of the tape read program, which is already more than one information unit ahead. The tape read program is continued up to

the first LWW but, for the time being, control will not return to it.

The first time control enters the type-punch program will be after the execution of block 3; the point where control then enters the output program was specified at the end of block 1.

In block 5 the question is asked whether $d_2 = 0$, i.e. whether the tape read program is still active; if not, control enters block 6.

Evidently the tape read program was made non-active in block 4 and as a result the transport $x \Rightarrow y$ did not take place. Therefore the transport takes place in block 6 and the temporarily stopped tape read program becomes active again due to the substitution $0 \Rightarrow d_2$.

Control arrives in block 7 when the tape read program is still active, i.e. when it has not yet completed the building up of the following x. The type-punch program is left no alternative but to end its activity for the time being by means of the substitution $1 \Rightarrow d_3$. As soon as the construction of x has been completed, the question in block 2 will detect that the type-punch program is waiting.

It is apparent from the diagram that after the standard entrance in block 1 at least one of the bits d_2 or d_3 is = 0, i.e. all the time from then onwards at least one of the two programs is active. As a result control can only arrive in the external main program if at least one interruption signal is still to come. Sooner or later, therefore, control will return to the communication program. There is not the slightest objection to prohibiting the interruption of class 6 in the main program for a while, e.g. for an interruption program of greater priority. In a like manner there is no difficulty at all when the communication program described here is interrupted by an interruption of greater priority.

It is not improbable that the external program will be somewhat slowed down by this additional task: in all normal applications this effect will be practically negligible. If the overall speed of the external program is limited by the capacity of (other!) communication mechanisms, so that the X1 has a surplus of computing time, then the additional task may leave the speed of the main program unaffected: one makes more efficient use of the X1.

In the scheme just described the input can, at most, be two information units ahead of the output. By providing more storage space for buffering the whole process may be speeded up in some cases (e.g. if, as a rule, the input program requires less time per unit than the output, but it occasionally has to skip a long piece of tape). The program then becomes more complicated, but from the point of view of mutual synchronization it contains nothing new. As in all probability this program will be executed in the spare time of a main program, there is in this case probably little interest in speeding it up this way.

Finally: in this example no use has been made of any knowledge of the times required by the input of x and the output of y. Even if, in certain cases, one knows these times and thinks one can make use of this knowledge by omitting some test with respect to the synchronization, we nevertheless strongly discourage one to do so. For example, the idea that after the output of an information unit the input of the next unit has already been completed (the tape reader being a faster mechanism), should never tempt anybody to omit the test in block 5 and the substitution in block 7. If the tape gets stuck, the tape reader will not send an interruption signal to the X1 until the difficulty has been removed and the tape moves on. Furthermore, one should realize that, in view of the fact that our communication programs are subject to interruptions from other classes, one only knows minimum times! In conclusion: if one does not take all possible precautions, a complete chaos seems unavoidable.

Next we turn to an analogous problem. The program required must read a tape and another tape must be punched dependent on what has been read. It must be possible, however, to execute this program simultaneously with an unknown main program, which uses neither tape punch nor tape reader, but may use the typewriter. The fact that the tape punch and the typewriter send the same interruption signal to the X1, makes this problem considerably more complicated than the previous one.

As punching and typing are mutually exclusive, we specify that the typing of the main program must have priority above the punching. Obviously it is pointless to ask our program to work together with a main program that makes continuous use of the typewriter! Again we demand that all precautions to be taken occur in the additional communication program. We demand of the main program that it leaves the working spaces of the standard tape read program untouched.

Since it must be possible to interrupt the "punch program" in favour of the type program at the first "summons" of the main program, we leave all common standard type-punch facilities at the disposal of the type program and accept the fact that no use is made of the standard punch subroutines. For: the latter make use of the same working spaces as the standard type programs, and information stored there for the purpose of punching would be in continuous danger of being destroyed by the type programs.

The working spaces previously allocated to the type-punch program are now reserved for the typing in the main program and new working spaces are introduced for the punch program. In particular, in this interplay the four addresses T_{tp} will record the state of an interrupted type program and four new addresses are introduced to record the state of an interrupted punch program; their contents will be denoted be T_{punch} .

To fill these addresses, we program a new subroutine entrance, called Punch WW, to the central program for class 6:

$$\Rightarrow \boxed{\text{State} \Rightarrow T_{punch}} \quad .$$

It is analogous to blocks 4 and 5 in the diagram in 4.9.5.

We modify the function of the bits d_2 and d_3 of the state record of class 6; furthermore we shall also make use of d_4 and d_5.

$d_2 = 0$ and permanently so, to be able to make use of the possibility of translation (see below)

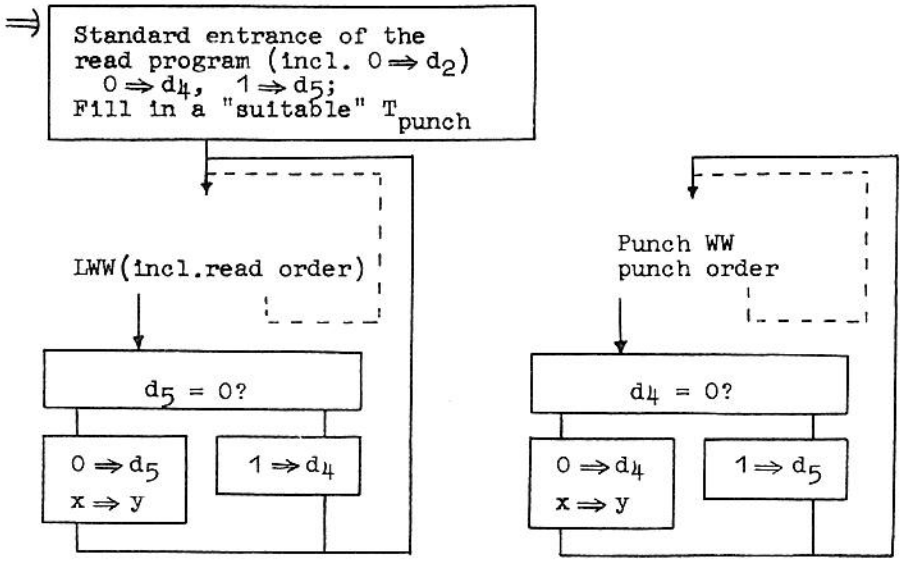
$d_3 = 0$ type program active, otherwise = 1

$d_4 = 0$ tape read program active, otherwise = 1

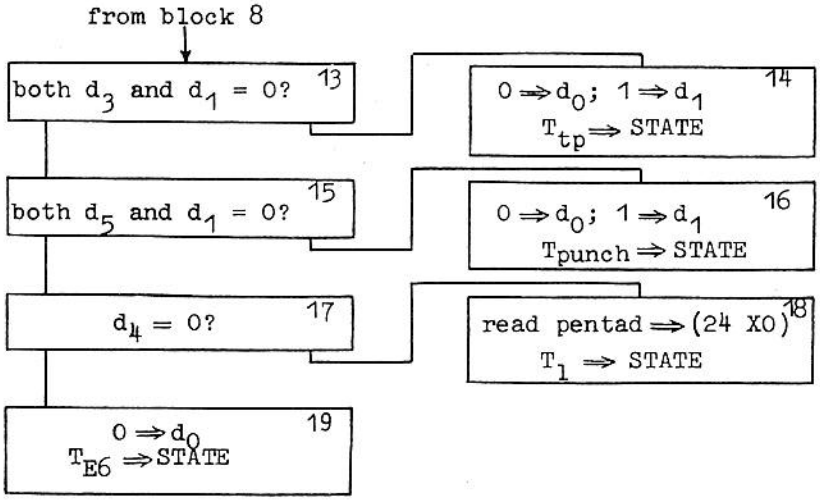
$d_5 = 0$ punch program active, otherwise = 1 .

The diagram of the program that reads and punches is analogous to the previous one.

In order to let this scheme work, in particular in order to let the new bits of the state record function properly we must interfere with the operation of the central program of class 6, as it was described in 4.9.5. We do so by changing the address in 25 X0, the translation possibility. If we now choose $d_2 = 0$ permanently, we can always



interfere as long as reading is permissible. When the jump to (25 X0) is executed the substitution $1 \Rightarrow d_0$ has already taken place in anticipation of a read order.



As the blocks in the diagram in 4.9.5 - of which this is an extension - were numbered from 1 to 12, the blocks in this diagram are numbered from 13 onwards.

Since d_2 is = 0 permanently, the answer to the question in block 6 depends only on d_0: when an interruption signal from the tape reader is still to come (i.e. $d_0 = 1$), block 7,9 and 10 will function normally, i.e. typing will then take place in the ordinary way.

The address in 25 X0 directs the control from block 8 to block 13, after substituting $1 \Rightarrow d_0$. In order to be sure that we are not going to punch when we should type instead, block 13 starts off with the question that control would otherwise have met in block 7. If the answer is affirmative we continue the type program. Block 14 is analogous to block 9.

If a negative answer is given to the question in block 13 (i.e. no typing), we investigate in block 15, whether punching must tape place, if so, control is sent, in block 16, to the punch program.

If not, the program in block 17 finally investigates whether the tape read program is to be continued. Here only d_4 is considered, for it is known that the interruption signal from the tape reader has already arrived (this was established in block 6).

Block 18 is analogous to the continuation of block 8.

When reading is not allowed either, the control returns to the main program via block 19.

The test whether the tape read program is to be continued takes place in block 17 after it has been established that there is not to be typed or punched. The reason for this is that d_0 will be = 1 for some time after an affirmative answer has been given to this question: during this period control will not enter blocks 13 to 19.

Again we can deduce (but now from the fact that at least one of the bits d_4 or d_5 will be = 0) that control will not be sent to the main program unless an interruption signal from class 6 is to come. For this purpose it is essential that control goes to block 6 on leaving block 12 and that it does not go directly to block 10! For, after block 14 no interruption signal from the tape reader is still to come: if, firstly, only an echo order is executed in the continuation of the type program, secondly,

the substitution $1 \Rightarrow d_1$ is annihilated by the call "Obliterate TPWW" (see 4.9.4) and, thirdly, the standard exit of the type program then follows, it is essential that either reading or punching takes place before the main program is continued.

5 SOME SPECIAL ASPECTS OF THE COMMUNICATION PROGRAM

In the previous section the communication program was described, emphasis being particularly laid on the fact that it is an interruption program; after an additional remark in this connection the rest of this chapter is devoted to a number of other aspects.

As already mentioned in 3.6, the communication program uses no counting jumps and only subroutine jumps with m = 14 or = 15. Without this restriction the "state" (see 4.9.5) would also include a number of (r_m)'s and (s_m)'s, which would retard the recording and the restoring of states. As per interruption of class 6, only a small number of bits of information are transported to and from the external apparatus, this would not be advantageous. In dealing with punched cards, for example, the situation is quite different: in that case there is one interruption for at least one whole card and then it will be worthwhile to streamline the interruption program itself at the cost of more elaborate recording and restoring of states.

5.1 The communication program in the dead memory

The fact that there are no C-corrected orders in the communication program is a trivial consequence of its being designed to be stored in the dead memory. This last fact, however, had a number of other consequences.

Programs in the dead memory make use of a number of fixed working spaces in the living memory: as the communication program, an interruption program, is executed at moments hardly known to the main program, we may just as well regard these working spaces as entirely reserved for the communication program. For this reason it was the more desirable to keep the number of such working locations as small as possible, be it at the cost of some program space!

Furthermore, special steps had to be taken with regard to the possibility of extension of the facilities of the communication program. For technical reasons it was desired to concentrate all words of which modification could be expected, in two consecutive pages; D16 and D17 were chosen for this purpose.

At all points at which extension was expected (amongst others more - and multiple - autostarts and more directives)

the standard program restricts itself to the detection of the situation for which no program is included yet. In most cases control then jumps to address 23 X2 or 24 X2, reserved for this purpose. (Despite the "merging" it is then possible to trace which situation was encountered.) The jumps to 23 X2 and 24 X2 enable us to actually try out and use certain extension possibilities by means of a piece of program in the living memory, before any actual modification is made in D16 or D17, i.e. before the extension is definitely included.

5.2 Extension possibilities of the keyboard program

5.2.1 Input of floating point numbers

The program that handles the input of floating point numbers by means of the keyboard is not included in the standard program; what has been included is the detection whether somebody is using the keyboard for the input of a floating point number. In that case the extension program can find all relevant data in the memory.

After the last digit of the numerical part the decimal exponent is pressed in (preceded by its own sign) and afterwards one presses the key F. If the exponent is = 0, it may be omitted. To introduce, for example, +23.4 one may press

```
+ 23.4       F
+   .234  +2 F
+  0.234  +2 F
+ 23.4    -1 F
```

Two addresses in page D16 are connected with this extension: in the final version address 24 D16 must contain the address at which the keyboard program must be continued when either the F or the sign of the exponent has been detected. Address 25 D16 provides the extension possibility for the subroutine for the conversion from floating decimal to floating binary notation, viz. for the case where the given decimal exponent is $\geqslant +0$. (For the case $\leqslant -0$ the subroutine has been included: the conversion of fixed point numbers makes extensive use of it in both the input via the keyboard and via the punched tape.)

To make this extension logically possible we were compelled to prescribe the pressing of key H (to bring the keyboard program back to its neutral state, (see 4.4)) in case of an error: otherwise the next sign would be mistaken for the sign of an exponent.

5.2.2 Multiple autostarts

No meaning has been assigned as yet to the three autostarts ., F and G. (Addresses 18, 21 and 22 D16). In order to be able to introduce more than three new autostarts, they will probably all three be used as introductory symbols of so-called "double (or possibly multiple) autostarts". Again, only the detection facility was included in the standard program. The "state record of class 7" = (27 X0) is $\geq +0$ during the input of numbers, is = -0 during the neutral state of the keyboard program; the latter now also tests, whether (27 X0) happens to be < 0. This will be the case as soon as the introductory key of a multiple autostart has been pressed. In the final version, i.e. with the extension included, address 23 D16 contains the address where the keyboard program will then have to be continued.

5.3 The extension possibilities of the tape read program

In all type indications dealt with, the assemblage routine remains sensitive to all directives, i.e. to the introductory symbol D. The lines of page D17 are in one to one correspondence to the 32 possible symbols that can follow this D. A jump order to some point in the dead memory is filled in at the corresponding line for the incorporated directives, for the time being there is a jump order to the address 24 X2 on the remaining lines. When a not yet incorporated directive is met, control therefore arrives at 24 X2 in the living memory. For detection purposes the symbol following the D can be found in the A-register, while the following pentad is recorded in S(and 24 X0). Definite inclusion of a new directive implies modification of the corresponding line of page D17.

As the reader will realize, the maximum number of directives is not restricted to 32: the number of directives can be extended at will by a technique similar to that used in the double autostarts (see 5.2.2). The most obvious extensions are additional type indications.

Due to the type indications the information on the tape could be punched in different codes and these codes are entirely independent of each other. The end of a number on the tape, for example, (under type indication DN, see 4.1.3) is indicated by the first

symbol of the next information unit, i.e. a sign, D or X. If it were permissible for a number on the tape to be followed without warning by an order, the detection of the end of the number would become considerably more complicated if not impossible: for, an order can begin with one of the digits 0 to 7.

This convention costs some extra symbols on the tape as soon as "the type" changes: the individual codes, however, being independent of each other can be made more efficient. The total effect of the introduction of the type indication is that tapes are probably somewhat shorter. The most important argument in favour of the introduction of type indications was, however, that thereby the possibility of unrestricted extension was retained.

In this connection we should like to draw attention to the possibility of constructing type indications under which the assemblage subroutine temporarily becomes "blind" to directives. It is only possible to detect a directive if no information unit starts with a D: once information units are allowed to start with the symbol D, the assemblage routine must be insensitive to normal directives. In that case one therefore chooses some kind of convention to announce a directive or to end the validity of the type indication concerned.

In the second place, we should like to mention the possibility of making type indications under which information units are constructed that occupy two (or more) words in the memory, e.g. double length numbers or floating point numbers.

When the assemblage routine is called in for an information unit of this kind, control returns for the first time with the first assembled word; at the next call no tape is read but the next word that was still in store is handed over. In other words as many states of the assemblage routine as there are words derived from one information unit correspond to such a type indication.

When the assemblage routine encounters a directive that announces a multiple-word-type, not only must the type be recorded, but the assemblage routine must also be brought into the corresponding initial state. As a matter of fact the type indication is recorded by filling in an address at location 18 X2; the assemblage routine reacts to this specification of the current type indication by jumping (in its third order, see address 2 D10) to the address indicated there (by means

of the order 2T 18 X2). The contents of address 18 X2, or only the more significant bits of this word, are particularly suitable for distinguishing between the different "sub-states" of the assemblage routine that then succeed each other cyclically.

5.4 Some closing remarks

The design of the X1 can be regarded as an effort to pass, as successfully as possible, between Scylla and Charybdis, viz. complexity and ease of handling. In the above we have described our efforts to make this narrow passage look like an open sea.

A large number of these efforts concerned the notation of the orders. Thus the position of the columns in which the programmer writes down the variants was purposely chosen so that they can be logically read from left to right. First one notes whether the order should be "partly or sometimes not" executed, next comes the kernel of the order, immediately following the address one writes down the variant of address modification and finally one specifies whether, and if so how, the order finally sets the condition.

Due to the use of two paragraph letters, the first of which can be omitted in most cases, the address notation is able to cover a large memory when necessary, but in such a way that it does not lead to an unnecessarily cumbersome notation as long as the program requires only a small memory.

The subdivision of the memory into pages of 32 lines was chosen so as to make the correlation between the address as it is written down and its binary representation as simple as possible. It was a lucky coincidence that 32 lines is a conventient number for a standard program sheet, that all kinds of five hole paper tape mechanisms are commercially available and that - as experience has shown us - most people have little difficulty in recognizing binary numbers of five digits immediately, and conversely in putting them in switches without hesitation.

In the notation of the function part (with the exception of the P-orders) we also adhered to the principle that the programmer and the operator can find the binary representation from the programmers symbols and vice versa

without calculation or the use of extensive conversion tables. Without these features the facilities of the operator's desk could never be used to full advantage.

The last aspect of the complexity of the X1 is shown in the interruption facility. The communication routines described serve a double purpose: on the one hand it is hoped that they form a well balanced set which the programmer may use, paying practically no attention to the problems of parallel programming, on the other hand it is hoped that they provide the more ambitious programmer all the facilities he may need for more refined work in this field.

Finally: the introduction of the autostarts considerably increased the ease of handling the X1 - in the literal sense of the word!

References

1. A.C.M. The Western Joint Computer Conference: "New Computers, Report from the Manufacturers", Los Angeles, March 1957;

2. Dijkstra, E.W.: "Programmering voor de ARMAC, Deel I", Rapport MR 25 van de Rekenafdeling van het Mathematisch Centrum, Amsterdam, 1956;

3. Hartree, Douglas R.: "Calculating Instruments and Machines", University Press, Cambridge, 1950;

4. Poel, W.L. van der : "The Logical Principles of some Simple Computers", Thesis, Amsterdam 1956;

5. Rutishauser, H.: "Massnahmen zur Vereinfachung des Programmierens", Nachrichtentechnische Fachberichte, Friedr. Vieweg und Sohn, Braunschweig, Band 4, 1956;

6. Scott, Dana S.: Techn. Rep. no 1, Princeton University, 10 June 1958;

7. Wilkes, M.V., D.J. Wheeler and S. Gill: "The preparation of programs for an electronic computer", Addison-Wesley Press Inc., Cambridge (Mass.), 1951.

SAMENVATTING

De X1 is een automatische rekenmachine, die ontworpen is met het doel voor ogen een machine te construeren, die zowel voor wetenschappelijk als voor administratief rekenwerk goed gebruikt kan worden.

In verband met de wijd uiteenlopende eisen moet het daarom mogelijk zijn, meer of minder uitgebreide installaties op te bouwen. We verdelen daartoe de complete installatie in drie onderdelen: de basismachine, het geheugen en de in- en uitvoerapparatuur.

Terwijl de basismachine in principe voor elke installatie dezelfde is, kan de omvang van het geheugen, evenals de hoeveelheid en hoedanigheid van de aangekoppelde communicatie-apparatuur van installatie tot installatie sterk verschillen.

De basismachine werkt intern in het tweetallig stelsel, de woordlengte is 27 bits (tekenbit, gevolgd door 26 binalen). Alle transporten en optellingen worden woordsgewijs parallel uitgevoerd, hun tijdsduur varieert van 36 tot 84 μs, vermenigvuldiging en deling vergen 500 μs.

Het rekenorgaan, dat alle operaties met vaste komma uitvoert, is uitgerust met twee registers (A resp. S) van 27 bits, welke als volledig onafhankelijke accumulatoren gebruikt kunnen worden, maar in de vermenigvuldiging en deling (evenals in speciale schuifopdrachten) gekoppeld zijn. Afgezien van deze koppeling, waarin de registers A en S een verschillende rol spelen, zijn zij gelijkwaardig. Voorts beschikt het rekenorgaan over 't zg. B-register van 16 bits (teken en 15 binalen), dat wel de volledige accumulatorfuncties heeft - zij het, dat er niet in geschoven kan worden en dat logische operaties en communicatieopdrachten in tegenstelling tot A en S de inhoud van het B-register ongemoeid laten - maar zijn belang ontleent aan het feit, dat het gebruikt kan worden voor automatische adresverandering van opdrachten.

Het geheugen kan één opdracht per woord bergen, de opdrachtencode is (in principe) een één-adres-code. In het opdrachtwoord staan 15 bits ter beschikking van het adresgedeelte, 12 voor het functiegedeelte; deze 12 zijn verdeeld in 6+2+2+2. Het zestal beschrijft de kern van de functie, de drie tweetallen beschrijven elk een zg. variant.

De eerste variant beheerst, of de opdracht soms conditioneel geskipt moet worden. Hierbij is ondergebracht de mogelijkheid de bewerking in beginsel wel uit te voeren - om het resultaat aan een criterium te toetsen - edoch zonder wijziging aan te brengen in het geheugen of de registers A, S en B.

De tweede variant beheerst de adresverandering, resp. -interpretatie Hierin ligt vast of het adres van de opdracht, zoals deze in het geheugen vastligt, vòòr de uitvoering met de inhoud van het B-register vermeerderd moet worden. (NB. De inhoud van het B-register kan negatief zijn.) Hierbij is ondergebracht de mogelijkheid, de adrescijfers niet te laten verwijzen naar een plaats in het geheugen, maar ze meteen te verwerken als operand.

De laatste variant - die der conditiezetting - maakt het mogelijk, het resultaat van een operatie aan een vraag te onderwerpen: er zijn drie criteria, nl. de tekentest, de nultest en de test op gelijkheid van teken. Het antwoord op deze vraag wordt vastgelegd in een aparte flip-flop - de zg. conditie - , de inhoud waarvan op elk gewenst moment door de eerstgenoemde variant in de afloop van het proces betrokken kan worden.

Dankzij de varianten ontstaat een machtige opdrachtencode met een grote flexibiliteit. Omdat het niet eenvoudig is, alle mogelijkheden van de code op zijn efficientst uit te buiten en pogingen in die richting aanleiding kunnen geven tot tamelijk ingewikkelde constructies, is de grootst mogelijke aandacht besteed aan de manier, waarop de programmeur de varianten noteert. In de gekozen notatie zijn voor de varianten drie kolommen gereserveerd, waarvan de plaats strookt met hun logische functie, indien we de opdracht van links naar rechts lezen. In het normale geval blijven deze kolommen oningevuld, anders wordt in zo'n kolom een speciale letter ingevuld om aan te geven welke variant op welke wijze wordt toegepast. Wanneer de uiteindelijke opdrachtencode ondanks zijn complexiteit hanteerbaar is geworden, is dat voor een niet onbelangrijk gedeelte aan de notatie der varianten en die van de kern van het functiegedeelte te danken.

De structuur van de basismachine is met zorg zo gekozen, dat een, wat geheugen of communicatie-apparatuur betreft, aanvankelijk bescheiden installatie desgewenst later kan worden uitgebreid.

Een uitbreiding van het geheugen biedt tot een maximum van $2^{15} = 32768$ woorden geen logische moeilijkheden, aan-

gezien in de opdrachtencode - een facet van de basismachine - 15 bits ter beschikking van het adresgedeelte staan.

Om de basismachine echter vruchtbaar samen te laten werken met allerhande communicatie-apparatuur moet het daarbij optredende probleem van synchronisatie van tevoren zijn opgelost Een van de manieren om dit probleem op te lossen is om voor de informatietransport tussen reken- en communicatieproces voldoend grote buffers ter beschikking te stellen en voorts de communicatie-apparatuur te laten besturen door afzonderlijke controle-apparatuur, die onafhankelijk van en simultaan met de basismachine werken kan. Het streven bij het ontwerp van de X1 is geweest, om deze extra buffers en additionele controle-apparatuur tot een nog practisch minimum te beperken en zoveel mogelijk van deze additionele taken aan de basismachine te delegeren. (De grote snelheid van geheugen en rekenorgaan, die voor het wetenschappelijk rekenwerk gewenst was, wordt hierdoor bij processen, waarbij communicatie de hoofdschotel vormt, te nutte gemaakt.) Dit nauwere contact tussen communicatie-apparaat en basismachine, dat vaak plaats zal moeten vinden op ogenblikken, die niet zozeer bepaald zijn door de preciese staat van vordering van het rekenproces, dan wel door de toestand van het externe communicatie-apparaat, dat een eigen snelheid, een eigen "tijdsbewustzijn" heeft, impliceert de noodzaak het probleem van synchronisatie in de basismachine zelf op te lossen. Daartoe is de zg. "ingreepfaciliteit" geschapen.

Deze maakt het mogelijk, dat de basismachine op grond van een uitwendig signaal (uitgezonden door een of ander communicatie-apparaat) het programma, dat op dat moment onder behandeling is, onderbreekt ten gunste van een zg. "ingreepprogramma", dat het vereiste contact tussen het geheugen en het apparaat in kwestie verzorgt en daarna de machine het onderbroken programma laat vervolgen "alsof er niets gebeurd was".

De ingreep maakt het mogelijk, dat de X1 omschakelt op een urgentere taak, zodra het tijdstip daarvoor is aangebroken. Het is begrijpelijk, dat de X1, wanneer hij met verschillende apparaten samen moet werken, daartoe voor prioriteitsregels ontvankelijk moet zijn. Behalve dat er faciliteiten zijn ingebouwd om - via het programma - de prioriteit te regelen, kunnen bovendien door het programma over bepaalde trajecten alle ingrepen tegengehouden worden. Men realisere zich, dat door de ingreep de afloop

van het hele proces niet meer bepaald is door het programma alleen, maar tevens door de onbekende momenten, waarop contact met de externe apparatuur gewenst is. Om desalniettemin een programma te construeren, dat zijn totale taak naar behoren verricht, is het noodzakelijk om over bepaalde critieke stukjes programma de onzekerheid of daarin soms ingrepen zijn opgetreden, uit te kunnen bannen.

De "parallelle programmering"- dwz. de compositie van programma's, die op in micro onbekende wijze door elkaar heen uitgevoerd worden - mag een fascinerende opgave zijn voor hem, die geïnteresseerd is in deze methode ter verhoging van de efficiëntie waarmee de machine gebruikt wordt, het is duidelijk dat dit geen taakverzwaring mag inhouden voor hem, die de machine primair gebruikt als werktuig om aan zijn resultaten te komen. De conceptie van de ingreep geeft dus de verplichting om een organisatie van subroutines op te bouwen, die enerzijds een belangrijk gedeelte van de mogelijke vruchten van de papallelle programmering plukken, anderzijds de gebruiker niet nodeloos belasten.

Voor schrijfmachine, bandlezer en bandponser is een aantal communicatie-subroutines ontwikkeld; deze apparaten veroorzaken dezelfde ingreep, zij behoren zoals wij zeggen "tot dezelfde klasse". Zij hebben uit oogpunt van synchronisatie gemeenschappelijk, dat een ingreepsignaal van een van hen niet betekent, dat de X1 binnen een bepaald tijdsbestek iets moet doen, maar dat het slechts betekent, dat de X1 van nu af aan wat kan doen: deze apparaten kunnen nl. altijd wachten en plaatsen daardoor de X1 nimmer in een zg. essentiële haastsituatie. De communicatieprogramma's voor deze drie apparaten hebben daarom na het hoofdprogramma de laagste prioriteit.

Wij beperken ons hier tot de beschrijving van die communicatie-programma's, waarbij de mate, waarin rekenproces en communicatieproces slechts in zeer beperkte mate uit de pas kunnen zijn. Staat men toe, dat dit uit de pas zijn zich over meer informatie uitstrekt, dan vereist dit bufferruimte in het geheugen. De brokstukken, waaruit de beschreven communicatie-programma's zijn opgebouwd, kunnen echter eveneens dienst doen in een organisatie, waarbij men een verder onderling uit fase zijn wil toestaan. De brokstukken zijn eveneens bruikbaar, indien men de verrichtingen van twee communicatie-programma's wil synchroniseren met betrekking tot elkaar, ongeacht een onafhankelijk hoofdprogramma, dat zelf de betrokken apparatuur helemaal niet gebruikt.

Appendix 1. The symbols on the typewriter

The correspondence between (TP) and the typed symbol is shown below.

sl = small letter
CL = capital letter
Tab = tabulator
NLCR = New Line Carriage Return.

(TP)	sl	CL	(TP)	sl	CL	(TP)	sl	CL
0	0	3/4	17	'	"	34	n	N
1	1	1/4	18	CL		35	o	O
2	2	1/2	19	sl		36	p	P
3	3	£	20			37	q	Q
4	4	$	21	a	A	38	r	R
5	5	%	22	b	B	39	s	S
6	6	*f*	23	c	C	40	t	T
7	7	&	24	d	D	41	u	U
8	8	(	25	e	E	42	v	V
9	9	)	26	f	F	43	w	W
10	Tab		27	g	G	44	x	X
11	NLCR		28	h	H	45	y	Y
12	-	_	29	i	I	46	z	Z
13	+	=	30	j	J	47	´	`
14	.	;	31	k	K	48	^	¨
15	,	?	32	l	L			
16	/	:	33	m	M	56-63	space	

Appendix 2. Binary representation of orders

The digits of the order word are numbered

$$d_{26}\ d_{25}\ \ldots\ d_1\ d_0\ .$$

The name of the order (function letter(s) and function number) determines the contents of the six most significant digits $d_{26}\ldots d_{21}$ of the order word. The number formed by these six binary digits can run from 0 to 63; its value is found by adding a number of times eight to the function number. The multiple of eight is determined by the function letter(s) and is given in the tabel below.

A:	0	LS:	$3\frac{1}{2}$
S:	1	B :	4
X:	2	T :	5
LA:	$2\frac{1}{2}$	Y :	6
D:	3	Z :	7

The value of the next six bits is determined by the variants

Address modification Condition-setting Condition reaction

	d_{20}	d_{19}		d_{18}	d_{17}		d_{16}	d_{15}
normally	0	0	normally	0	0	normally	0	0
A	0	1	P	0	1	U	0	1
B	1	0	Z	1	0	Y	1	0
C	1	1	E	1	1	N	1	1

The bits d_{14} ... d_0 give the binary representation of the address.

According to the above rules d_{21} would be = 0 in counting and subroutine jumps (4T and 6T); however, it is used for the index m, more precisely, if the binary digits of the index m are $b_2\ b_1\ b_0$ (for 4T) or $b_3\ b_2\ b_1\ b_0$ (for 6T), then the following holds.

$$d_{21} = b_2$$
$$d_{20} = b_1$$
$$d_{19} = b_0$$

and for 6T: $d_{18} = b_3$

(In the subroutine jump, the binary digit b_3 is therefore placed at the least significant side.)

The binary representation of the P-orders is given in the table below (the special paragraph letter C causes an increase of 2^{14} in the address: 0 C0 = 16384 X0).

Circuit	0P n	1P n	2P n	3P n	4P	5P	6P	7P
AA	6Y nX0	7Y nX0	6Y nX1	7Y nX1	6Y 0X8	7Y 0X8	6Y 0X5	-
AS	6Y nX2	7Y nX2	6Y nX3	7Y nX3	6Y 1X8	7Y 1X8	6Y 0X7	-
AB	-	-	-	-	6Y 2X8	7Y 2X8	-	-
SA	6Z nX2	7Z nX2	6Z nX3	7Z nX3	6Y 0X9	7Y 0X9	-	-
SS	6Z nX0	7Z nX0	6Z nX1	7Z nX1	6Y 1X9	7Y 1X9	6Z 0X5	-
SB	-	-	-	-	6Y 2X9	7Y 2X9	-	-
BA	-	-	-	-	6Y 0X10	7Y 0X10	-	-
BS	-	-	-	-	6Y 1X10	7Y 1X10	-	-
BB	-	-	-	-	6Y 2X10	7Y 2X10	-	-
-	-	-	-	-	-	-	-	7Y 0C0

Appendix 3. Storage reservations in the living memory

22 X0: Link interruption jump class 6
23 X0: Link interruption jump class 7
24 X0: Last pentad read
25 X0: Translation possibility
26 X0: State record class 6
27 X0: State record class 7
28 X0: Transport address
29 X0: Check switch
30 X0: Head] double length number
31 X0: Tail]

0 X2: A]
1 X2: S] STATE T_{E7}
2 X2: B]

3 X2: A]
4 X2: S] STATE T_{E6}
5 X2: B]
6 X2: link]

7 X2: A]
8 X2: S] STATE T_{tp}
9 X2: B]
10 X2: link]

11 X2: A]
12 X2: S] STATE T_1
13 X2: B]
14 X2: link]

15 X2: binary exponent
16 X2: decimal exponent
17 X2: sign of number of function digits
18 X2: current type indication
19 X2: current 1st paragraph letter
20 X2: link of tape read subroutines (order 0)

21 X2: link of tape read subroutines (order 1)
22 X2: link of tape read subroutines (order 2)
23 X2: extension possibility for all cases other than directives
24 X2: extension possibility for additional directives

25 X2: link of type-punch subroutine
26 X2: initial type code, punch code
27 X2: initial number, length of binary tape
28 X2: transformed number, start address of binary tape
29 X2: transformed type code, link for punch routine
30 X2: column count
31 X2: number of columns per line

Appendix 4. Standard program for class 6 and class 7

		DA	4	DO	DI		
0							unused
1							START ADDRESS
2							STOP ADDRESS
3							CONSOLE WORD
9 D2 ⇒ 4		2T	19	D18		P⇒	for autostart 6
⇒ 5		2A	4		A		Subr."Tape read program Active?"
6		2T	6	D1	A	⇒	
⇒ 7		3S	4		A		Subr."Preparation tape read pr."
8		2T	9	D1	A	⇒	
25 ⇒ 9		OLA	1		A		if "reading desired and permitted",
10		6A	26	X0			$1 \Rightarrow d_0$ and jump via possibility
11		2T	25	X0		⇒	of translation
⇒12		2A	4		A		Exit tape read program
14 D1→13		4A	26	X0			
14		2T	21	D0	A	⇒	
⇒15		6B	13	X2			LWW=Subr.Tape read pr."Wait switch"
16		2B	8		A		
30,17 D1→17		6A	3	X2	B		
18		6S	4	X2	B		STATE ⇒ T_{E6}, T_1, T_{tp}
19		2A	22	X0			
20		6A	6	X2	B		
14 →21		6Y	4	XP			read 6th class word
22		5P		AA			
23		2LA	26	X0			
24	U	2LA	5		A	Z	reading desired and permitted?
25	Y	2T	9	D0	A	→	
26	U	2LA	10		A	Z	type-punching desired and permitted?
27	Y	OLA	2		A		if so, $1 \Rightarrow d_1$
28		6A	26	X0			
29	N	2A	3	X2			
30	N	2S	4	X2			$T_{E6} \Rightarrow$ STATE
31	N	2B	5	X2			

		DA	0	D1		DI		
0	N	2T	6	X2		E	→	continue main program
1		2A	7	X2			⌈	
2		2S	8	X2			|	T_{tp} ⇒ STATE
3		2B	9	X2			|	
4		2T	10	X2		E	⇒ ⌊	continue type-punch program
⇒ 5		2A	8		A			Subr."Type-punch program Active?"
6 DO → 6	U	2LA	26	X0		Z		
7		2T	8	X0		Z	⇒	
⇒ 8		3S	8		A			Subr."Preparation type-punch pr."
8 DO → 9		6A	6	X2				
25 → 10		2LS	26	X0			⌈	$0 \Rightarrow d_1, d_2$ or d_3
11		6S	26	X0			⌊	
12		2T	22	X0		Z	⇒	
⇒ 13		2A	8		A			Exit type-punch program
14		2T	13	D0	A		⇒	
⇒ 15		6B	9	X2				TPWW=Subr.Type-punch pr."Wait switch"
16		2B	4		A			
17		2T	17	D0	A		⇒	
11 DO ⇒ 18		2Z	1	XP			⌈	read pentad ⇒ (24 X0)
19		6S	24	X0			⌊	
20		2A	11	X2			⌈	
21		2S	12	X2			|	T_1 ⇒ STATE
22		2B	13	X2			|	
23		2T	14	X2		E	⇒ ⌊	continue tape read program
⇒ 24		3S	2		A			Subr. Obliterate TPWW
25		2T	10	D1	A		⇒	
⏚ 26		0Y	64	XS				non 6 in I.P. INTERRUPTION 6
27		0Y	0	XS				X1 susceptible
28		6B	5	X2			⌈	
29		2B	0		A		|	to the recording of T_{E6}
30		2T	17	D0	A		⇒ ⌊	
⏚ 31		6A	0	X2				INTERRUPTION 7

		DA	0	D2		DI			
0		6S	1	X2					STATE ⇒ T_{E7}
1		6B	2	X2					
2		7Y	4	XP					read 7th class word (=key)
3	U	1A	15		A	Z			key H? if so, restore
4	Y	2T	0	D3	A		→		neutral state
5		2S	27	X0		Z			distinction between
6	Y	4P		AB					single and multiple
7	N	2B	15		A				autostarts
8		4P		SS		P			or is it decimal input?
9	N	2T	8	D16	B		→		if not, autostart
10		2B	21	X2					digit count in B
11	U	1A	10		A	Z			if point, record
12	Y	6B	16	X2					current digit count
13	Y	2T	2	D3	A		→		and finished
14	U	1A	9		A	P			
15	Y	2T	12	D3	A		→		if non-digit
16		0B	1		A	P			count digit; numerical part?
17	Y	2S	31	X0					decimal to binary
18		0X	10		A				conversion for
19	Y	6S	31	X0					numerical part (Y)
20	Y	2S	30	X0					or exponent (N)
21	Y	0X	10		A				
11 D3 → 22	Y	6S	30	X0					
23		6B	21	X2					preserve digit count
24		2T	1	D3	A		⇒		and finished
9 D2 ⇒ 25		2B	1	D0		P			Autostart 4
9 D2 → 26		2A	31	X0					Autostart 3
27	N	2T	30	D2	A		→		in case of 3
28		6A	1	X0	B				store tail
29		2A	30	X0					
27 D3;27 → 30		2B	1	D0					store head, console word
31		6A	0	X0	B				or single length number

		DA	0	D3		DI		
20 D21] 4 D2, 25]	→0	3S	0		A			bring keyboard program
24 D2	→1	6S	27	X0				into its neutral state
13 D2	→2	2A	0	X2				
	3	2S	1	X2				$T_{E7} \Rightarrow$ STATE
	4	2B	2	X2				(and stop in the main program)
	5	2T	23	X0		P	⇒	
9 D2	⇒6	6A	17	X2				store sign. Autostarts + and -
	7	2S	0		A			
	8	2B	0		A			
	9	7S	16	X2				set counts etc. to zero
	10	6S	31	X0		P		and remove neutral
	11	2T	22	D2	A		⇒	state
15 D2	⇒12	U 1A	14		A	Z		key G?
	13	N 2T	24	D16			→	if F or sign of exponent
	14	2A	16	X2		P		point encountered?
	15	N 2B	0		A			if not, whole number
	16	Y 5B	16	X2				if so, adjust decimal exponent
	17	Y 2B	26		A			if mixed number
	18	Y 4P		AA		Z		or was it a double
	19	Y 2B	52		A			length fraction
	20	6T	1	D5	14		⇒	Subr.Decimal→Binary Floating
	21	N 6T	4	D4	14		⇒	Subr.Shift for fixed point
	22	6T	31	D3	14		⇒	Subr. $\pm$(AS) ⇒ (AS)
	23	6A	30	X0				store result
	24	6S	31	X0				
	25	2T	0	D3	A		⇒	finished
9 D2	→26	2A	3	D0				Autostart 5
	27	2T	30	D2	A		⇒	with (A) = console word
15 D17	⇒28	2A	29	X0		Z		Directive DC
	29	Y 7A	29	X0				
	30	2T	9	D9	A		⇒	
	⇒31	3B	12		A			Subr. $\pm$(AS) ⇒ (AS)

			DA	0	D4		DI		
	0		0B	17	X2		Z		minus sign?
	1	Y	5P		AA				if so -(AS) ⇒ (AS)
	2	Y	5P		SS				
	3		2T	22	X0		Z	⇒	
10 → ⇒	4	U	3A	0		A	E		Subr. Shift for fixed point
	5	Y	7P						Stop: overflow of capacity
	6		1B	26		A	E		is one shift sufficient?
	7	Y	3P	25	AS	B			shift one place less than needed
	8	N	4P		AS		Z		if shift over more than
	9	N	2A	0		A			26 places is still to come
	10	N	2T	4	D4	A		→	
	11		0S	1		A	P		rounding-off
	12	N	2S	0		A			and final shift
	13	N	0A	1		A			over one place
	14		3LS	1		A			
	15		1P	1	AS		Z		
	16		2T	22	X0		Z	⇒	finished
14 D5 ⇒	17		3A	30	X0				
	18	U	0A	15	D14	B	P		divide by 10 [B] ;
	19	N	3P	1	AS				halve if necessary
	20		0D	15	D14	B			and adjust the
	21		7S	30	X0				binary exponent = [15 X2]
	22		3S	15	D14	B			that is under construction
	23		3P	1	SS				
	24		0D	15	D14	B			
	25		3A	23	D14	B			
	26	N	0A	1		A			continue, using its
	27		4A	15	X2				current link, with:
⇒	28	U	0B	16	X2		P		Subr. Negative power of ten
	29	Y	3B	16	X2		Z		reduction completed?
	30	N	4B	16	X2				
	31		2T	22	X0		Z	⇒	

			DA	0	D5		DI		
⇒	0		2B	52		A			Subr. <u>Decimal→Binary Floating</u>
⇒	1		2A	22	X0				] transport link
	2		6A	20	X2				
	3		3A	30	X0				
	4		3S	31	X0				
	5		6B	15	X2				store preset value binary exponent
	6		6P		AS		Z		normalize
	7	Y	2T	15	D5	A		→	if numerical part = 0
	8		7A	30	X0				
	9		5B	15	X2				
	10		2B	8		A			
	11	U	2A	16	X2		P		decimal exponent ⩾ +0?
	12	Y	2T	25	D16			→	
	13		6T	28	D4	14		⇒	Subr.Neg.power of ten. Finished?
	14	N	2T	17	D4	A		→	if not, divide by power of ten
7 →	15		5P		SS				] is either the numerical part
	16		2A	30	X0		Z		or the binary exponent = 0?
	17	N	3B	15	X2		Z		(in these cases no shift for
	18		2T	20	X2		Z	⇒	fixed point representation)
10 D7 21 D11 25 ⇒	19		6T	15	D0	14		⇒	LWW] new pentad after "Skip X",
	20		2T	23	D5	A		⇒	"Skip 0" and U,Y,N
⇒	21		2S	22	X0				Subr. <u>Analyse c</u>
	22		6S	20	X2				] transport link
20 →	23		2S	24	X0				
	24	U	1S	31		A	Z		] transition pentad c = X?
	25	Y	2T	19	D5	A		→	if so, skip X
	26	U	1S	30		A	Z		c = D (irective)?
	27	N	2T	20	X2		Z	→	non D: return to assemblage
	28		6T	15	D0	14		⇒	LWW] if directive
	29		2A	24	X0				next pentad ⇒ (A)
	30		6T	15	D0	14		⇒	LWW for 2nd pentad after D
	31		2S	0	D17	A			

			DA	0	D6		DI	
	0		6S	15	X2			] set distribution address
	1		4A	15	X2			] for directives
	2		2S	24	X0		Z	
	3		2T	22	X2		Z	⇒ via link of assemblage routine!
⇒	4		6S	17	X2			Subr. Read decimal number
⇒	5		2S	0		A		
⇒	6		2A	22	X0			] transport link
	7		6A	20	X2			]
	8		2A	0		A		
	9		2B	0		A		] set dec. exp. = -0
	10		7B	16	X2		Z	] and condition to Yes
24 →	11	N	0B	1		A		digit count
	12	N	0X	10		A		] decimal to binary
	13		6S	31	X0			conversion for double
	14	N	2S	30	X0		Z	length whole number
	15	Y	4P		AS			(with 1 multiplication
	16	N	0X	10		A	Z	] as long as head is = 0)
	17	N	7P					Stop: overflow of capacity
	18		6S	30	X0			
	19		2S	31	X0			
26 →	20	N	6B	16	X2			if point is detected
10 D11 →	21		6T	15	D0	14		⇒ LWW
	22		2A	24	X0			
	23	U	1A	9		A	P	
	24	N	2T	11	D6	A		→ new decimal
	25	U	1A	10		A	P	
	26	N	2T	20	D6	A		→ point
	27	U	2A	16	X2		P	] has point been encountered?
	28	Y	5B	16	X2			] then adjust decimal exponent
	29		2T	20	X2		Z	⇒ return, retaining condition
2 D10 ⇒	30		6T	21	D5	14		⇒ Subr. Analyse c Assemblage (DN)
	31		6T	4	D6	14		⇒ Subr. Read decimal number

			DA	0	D7		DI		
	0	N	2E	0		A			if whole number
	1	Y	2B	26		A			if fraction
	2		6T	1	D5	14		⇒	Subr. Decimal→Binary Floating
	3	N	6T	4	D4	14		⇒	Subr. Shift for fixed point
	4	N	7P						(A) ≠ 0: overflow of capacity
	5		6T	31	D3	14		⇒	Subr. ±(AS) ⇒ (AS)
	6		2T	27	D12	A		⇒	to end of assemblage
2 D10 ⇒	7		6T	21	D5	14		⇒	Subr. Analyse c Assemblage (D0)
(10→)	8		4P		SS		Z		
	9	N	7P						Stop if not blank
	10		2T	19	D5	A		⇒	back to Analyse c for next pentad
	11	DN	+	100	000		DI		(see 10 D20)
9 D2 ⇒	12	U	2A	3	D0		P		if writing +0 ⎫ Autostart 1
	13	Y	3A	0		A			if checking -1 ⎭ ⇒ check switch
	14		7A	29	X0				
	15		2A	31	D9				place quasi-link in (A)
	16		3S	5		A			0 ⇒ d_0 and 0 ⇒ d_2
	17		6T	9	D1	14		⇒	
	18		2T	29	D7	A	P	⇒	
4 D0 0 D1 ⇒	19		2A	8		A	Z		"empty" main program
0 D1 →	20	Y	2A	4		A			that merely waits for
22 →	21	U	2LA	26	X0		Z		completion of communi-
	22	Y	2T	21	D7	A		→	cation program
	23		2T	0	D3	A		⇒	Stop X1, restore status quo
⇒	24		2A	0		A			Internal Subr. Start Tape Reading
	25		6A	29	X0				
	26		0Y	64	XS				Standard
	27		2A	8	X0				Entrance
	28		6T	7	D0	14		⇒	Tape read program
2 D9 18 →	29		2S	7	D7	A			Set current type indication
	30		6S	18	X2				to D0
	31		6T	15	D0	14		⇒	LWW for 1st transition pentad

		DA	0	D8		DI		
2,10; 7 D28 →	0	6T	0	D10	14		⇒	Assemblage Subr. Processing cycle
⇒	1	2T	15	X2			⇒	after detection] of a directive
⇒	2	2T	0	D8	A		⇒	after processing]
⇒	3	2B	28	X0				when word is delivered
	4	2A	29	X0		P		test check switch
	5	Y 6S	0	X0	B			store] under control of the
	6	1S	0	X0	B	Z		check] transport address
	7	N 7P						Stop if discrepancy
	8	2S	1		A			] Increase transport
	9	4S	28	X0				] address by one
	10	2T	0	D8	A		→	close the cycle
2 D10 ⇒	11	6T	21	D5	14		⇒	Subr. Analyse c Assemblage (DB)
	12	2LS	3		A			
	13	2A	24	X0				
	14	2B	6		A	Z		] set pentad count
	15	6B	30	X0				]
26 →	16	Y OP	5	SS				
	17	Y OS	24	X0				
	18	3B	4		A			set count for
22 →	19	1P	1	SS		P		] cycle investigating
	20	N OLA	4		A			] five bits with a
	21	OB	1		A	P		] view to the
	22	N 2T	19	D8	A		→	] parity check
	23	OP	5	SS				
	24	6T	15	D0	14		⇒	LWW
	25	5B	30	X0		P		pentad count
	26	Y 2T	16	D8	A		→	word not yet complete
	27	2LA	4		A	Z		test parity
	28	N 2T	27	D12	A		→	to end of assemblage
	29	7P						Stop, wrong parity
14 D17 ⇒	30	2S	11	D8	A			Directive DB
	31	2T	8	D9	A		⇒	

			DA	0	D9		DI	
⇨	0		2A	0		A		⎤ Restart tape reading
	1		6A	29	X0			⎥ in the writing mode
	2		2T	29	D7	A	⇒	⎦
0 D17 ⇒	3		2S	7	D7	A		Directive DO
	4		2T	8	D9	A	⇒	
29 D17 ⇒	5	U	1S	18		A	Z	Directive DN
	6	Y	2T	26	D16		→	if DNE
	7		2S	30	D6	A		
16 D11 ⎤	8		6S	18	X2			set current type indication
31 D8;4 ⎦→								
30 D3;								
21; 24;27 ⎤→	9		2A	1		A		⎤ Return from assemblage
	10		2T	28	D12	A	⇒	⎦ after processed directive
16 D17 ⇒	11		2X	13		A		Directive DP
	12		1S	256		A		
	13		6T	15	D0	14	⇒	LWW for 2nd letter
	14		0S	24	X0			
	15		6S	17	X2			
	16		6T	15	D0	14	⇒	LWW for 1st address symbol
	17		6T	4	D10	14	⇒	Subr. Read address
	18		2B	17	X2			⎤ enter address read
	19		0B	27	D16			⎥ in paragraph table
	20		6S	18	X0	B		⎦
	21		2T	9	D9	A	⇒	directive processed
13 D17 ⇒	22		6T	5	D10	14	⇒	Subr.Read Address Directive DA
	23		6S	28	X0			new transport address
	24		2T	9	D9	A	⇒	directive processed
31 D17 ⇒	25		4S	28	X0			Directive DX
	26		6T	15	D0	14	⇒	LWW for transition pentad
	27		2T	9	D9	A	⇒	directive processed
18 D17 ⇒	28		6T	5	D10	14	⇒	Subr.Read Address Directive DE
	29		6S	16	X2			⎤ and jump
	30		2T	16	X2		⇒	⎦
	31		0X	20	D7	A		(X1 susceptible and 6 permitted see 15 D7)

		DA	0	D10		DI		
⇒0		2A	22	X0				Assemblage Subroutine
1		6A	22	X2				transport link
2		2T	18	X2			⇒	distribute on type indication
⇒3		6A	17	X2				Subr. Read Address
⇒4		2S	24	X0		Z		
⇒5		2A	22	X0				
6		6A	21	X2				transport link
7	Y	2T	28	D16			→	extension poss. octal address
8		6T	6	D6	14		⇒	Subr. Read decimal number
25 → 9		6A	30	X0				⎤ after conversion of
10		1A	16		A	P		⎥ the line number
11	Y	2P	4	AA				⎥ analyse next pentad
12		4P		AB				⎦
13	N	2T	8	D11	B		→	distribution jump if + - ABCP
14		1B	24	X0				⎤ paragraph letter
15		1B	24	X0				⎥ (possibly the first!)
16		1B	24	X0				⎦ times thirteen
17	U	1A	224		A	Z		= D?
18	Y	0S	0	D0	A			
19	N	1A	240		A	Z		if not, then = X?
14 D11→20		6T	15	D0	14		⇒	LWW for page number
21		2A	24	X0				or 2nd paragraph letter
22	Y	2T	30	D10	A		→	if D, X or A
23	U	1A	16		A	P		two explicit paragraph letters?
24	Y	6B	19	X2				change "current 1st par. letter"
25	Y	2T	9	D10	A		→	and jump back
26		2B	19	X2				⎤ addition of
27		0B	30	X0				⎥ the start address
28		0B	27	D16				⎥ of the paragraph
29		0S	18	X0	B			⎦
12 D11;22→30		2P	5	AA				
31		0X	1		A			addition of page number

			DA	0	D11		DI	
	0		6T	15	DO	14		⇒ LWW] next symbol ⇒ (A)
	1		2A	24	XO			
	2		2T	21	X2		E	⇒ address complete
	3		OA	29	D16			+] distribution
	4		OA	9	D11			- addresses
	5		OA	11	D11			A for special
	6		OA	30	D16			B closing symbols
	7		OA	13	D11			C of the line number
	8		OA	31	D16			P]
13 D10 ⇒	9		5P		SS			- after line number
	10		2T	21	D6	A		⇒
13 D10 ⇒	11		2A	16384		A		A after line number
	12		2T	30	D10	A		⇒
13 D10 ⇒	13		OS	16384		A	P	C after line number
	14		2T	20	D10	A		⇒
1 D17 ⇒	15		2S	17	D11	A		Directive DI
	16		2T	8	D9	A		⇒
2 D10 ⇒	17		2A	0		A		Assemblage (DI)
	18		6T	21	D5	14		⇒ Subr. Analyse c
(21→)	19		1S	26		A	P	= U, Y or N?
	20	Y	3P	12	SA			] if so, shift bits in A
	21	Y	2T	19	D5	A		→] and read new pentad
	22		6A	15	X2			
	23		OS	19		A	P	] Stop, if function digit > 7
	24	Y	7P					]
	25		2A	24	XO			
30 →	26		6T	15	DO	14		⇒ LWW for function letter
	27		2S	24	XO			
	28	U	1S	22		A	Z	= L?
	29	Y	OA	20		A		
	30	Y	2T	26	D11	A		→ ask for function letter anew after L
	31	U	1S	16		A	Z	

			DA	0	D12		DI		
	0	Y	2T	30	D12	A		→	function letter P
	1	U	1S	17		A	Z	]	convert function digit
	2	Y	OLS	25		A			and function letter(s)
	3	U	1S	30		A	Z		into the required
	4	Y	OLS	25		A			representation in
	5		OS	14		A			the six least significant
	6		OX	32		A			bits of the A-register
	7		OD	216		A			
	8	U	1A	171		A	P		4T or 6T? (i.e. index m obligatory)
	9		6T	15	DO	14		⇒	LWW for first address symbol
	10		6T	3	D10	14		⇒	Subr. Read address
	11	N	2T	18	D12	A		→	if neither 4T nor 6T
	12	U	2LA	8		A	Z	]	transport most significant
	13	N	OA	1	D14				bit if m ⩾ 8
22 →	14		1P	8	AA			]	add to the variants
	15		4A	15	X2				under construction
0 D14 →	16		6T	15	DO	14		⇒	LWW] after index m, ABC, PZE a new symbol
	17		2A	24	XO				
11 →	18	U	1A	18		A	P		
	19	N	1A	15		A	E		not PZE?
	20	N	1P	2	AA				
	21	Y	OA	3		A	E		not ABC?
	22	N	2T	14	D12	A		→	if PZE or ABC
	23		OP	6	SS			]	construct order
	24		2A	17	X2				word from its
	25		1P	6	SA				constituent parts
	26		OS	15	X2				
28 D8;6 D7 →	27		2A	2		A		]	exit of the assemblage
10 D9 →	28		4A	22	X2				subroutine with a
	29		2T	22	X2		E	⇒	new word in S
0 ⇒	30		6A	17	X2				if P-order
	31	U	1A	7		A	Z		

			DA	0	D13		DI		
	0	Y	2S	16352		A			1f 7P
	1	Y	2T	25	D13	A		→	
	2	U	1A	5		A	E		4P or 5P?
	3	Y	1A	3		A	P		
	4		6T	15	D0	14		⇒	LWW
	5		2S	24	X0				
	6	Y	0P	5	SS				Construction of the address bits for the register transport orders 4P and 5P
	7	Y	6T	15	D0	14			
	8	Y	0S	24	X0				
	9	Y	2LS	231		A			
	10	Y	6S	24	X0				
	11	Y	3P	1	SS				
	12	Y	0S	24	X0				
	13	Y	0S	33		A			
	14	Y	2LS	99		A			
	15	Y	0S	224		A			
	16	Y	2T	25	D13	A		→	
	17		2A	128		A	E		= 6P? (ohterwise 0-3 P)
	18	Y	0P	27	SA				
	19	N	6T	6	D6	14		⇒	Subr. Read decimal number
	20		6T	15	D0	14		⇒	LWW for 2nd circuit letter
	21	U	1A	24	X0		Z		take the circuit letters into account
	22	N	0S	64		A			
	23	U	1A	13		A	Z		
	24	N	2B	62		A			
16,1 →	25	Y	2B	54		A			
	26		2A	17	X2				take the function digit into account
	27	U	3LA	1		A	Z		
	28	N	0S	32		A			
	29	U	2LA	1		A	Z		
	30	N	0B	1		A			
	31		6B	17	X2				

		DA	0	D14		DI		
0		2T	16	D12	A		⇒	now final test for variants
1	DN	+	6710	8856		DI		constant for m ⩾ 8, see 13 D12
⇒ 2		2A	22	X0				<u>Internal Subr. Punch Binary Word</u>
3		6A	25	X2				transport link
4		2A	422		A			
8 → 5		1P	1	SS		P		cycle for the
6	N	OLA	3		A			determination
7		1A	16		A	P		of the parity bit
8	Y	2T	5	D14	A		→	
9		1P	4	SA		P		
14 → 10		OP	5	AS		E		cycle for the
11		6T	15	D1	14		⇒	TPWW punching of
12		6Y	1	XP				six pentads
13	N	3A	2		A	E		
14	Y	2T	10	D14	A		→	
15		2T	25	X2		E	⇒	finished
16	DN	+	4194	3040				table of normalized powers of ten
17		+	5242	8800				
18		+	6553	6000				
19		+	4096	0000				
20		+	5120	0000				
21		+	6400	0000				
22		+	4000	0000				
23		+	5000	0000				
24		+		4				table of corresponding binary exponents
25		+		7				
26		+		10				
27		+		14				
28		+		17				
29		+		20				
30		+		24				
31		+		27				

		DA	0	D15		DN		
0		-6710 8863				DI		isolated sign digit
⇒ 1		3P	5	AA				Auxiliary subroutine
⇒ 2		6Y	1	XP				for punch programs
3		2T	15	D1	A		⇒	(punches pentad)
⇒ 4		2B	25		A			
⇒ 5		2A	22	X0				
6		6A	25	X2				Auxiliary subroutine
7		2A	0		A			for punch programs
10 → 8		6T	2	D15	14		⇒	(punches pieces of
9		1B	1		A	P		blank tape)
10	Y	2T	8	D15	A		→	
11		2T	25	X2		Z	⇒	
⇒ 12		2A	22	X0				Auxiliary subr. for simple type routines
13		6A	29	X2				transport link
14		4P		SS		P		
15	Y	2A	13		A			
16	N	2A	12		A			take absolute
17	N	5P		SS				value and type
18		6T	15	D1	14		⇒ TPWW	original sign
19		6Y	2	XP				
20		2A	0		A			type 1st decimal
21		0D	31	D15				facultatively;
22		1S	0		A			prepare for
23		6T	15	D1	14		⇒ TPWW	deconversion and
24		6Z	2	XP		P		set count;
25		2D	31	D15				return, retaining
26		0S	1		A			the condition for
27		2B	6		A			zero suppression
28		2T	29	X2		Z	⇒	
9 D2 ⇒ 29		2A	1	D18	A			<u>Autostart 7</u>
30		2T	28	D18	A		⇒	jump with quasi-link in A
31	DN	+ 1000 0000						(see 25 D15)

	DA	0		D16		DI	
0	2T	0	Z	Z0			⇒ Autostart 0
1	6T	0	Z	E0	9		Interruption jump class 1
2	6T	0	Z	F0	10		" " " 2
3	6T	0	Z	H0	11		" " " 3
4	6T	0	Z	K0	12		" " " 4
5	6T	0	Z	L0	13		" " " 5
6	6T	26		D1	14		" " " 6
7	6T	31		D1	15		" " " 7
8	0A	0		D16			Distribution address autostart 0
9	0A	12		D7			" " " 1
10	0A	11		D21			" " " 2
11	0A	26		D2			" " " 3
12	0A	25		D2			" " " 4
13	0A	26		D3			" " " 5
14	0A	4		D0			" " " 6
15	0A	29		D15			" " " 7
16	0A	22		D21			" " " 8
17	0A	14		D19			" " " 9
18	0A	0	Z	R0			" " " .
19	0A	6		D3			" " " +
20	0A	6		D3			" " " -
21	0A	0	Z	S0			" " " F
22	0A	0	Z	T0			" " " G
23	0A	0	Z	W0			" "multiple autostart
24	0A	0	Z	U0			if F or sign exponent (see 13 D3)
25	0A	0	Z	Y0			if pos.dec.exp. (see 12 D5)
26	0A	0	Z	N0			if DNE (see 6 D9)
27	0A	0	Z	Z0			start address paragraph tabel (see 9 D16)
28	0A	8		D10			possible octal address (see 7 D10)
29	2T	0	E	Z0	A	⇒	if + after line number (see 3 D11)
30	2T	0	E	E0	A	⇒	if B " " " (see 6 D11)
31	2T	0	F	E0	A	⇒	if P " " " (see 8 D11)

	DA	0	D17	DI	
0	2T	3	D9	A	⇒ DO
1	2T	15	D11	A	⇒ DI
2	2T	0 E	FO	A	⇒ D2
3	2T	0 E	HO	A	⇒ D3
4	2T	0 E	KO	A	⇒ D4
5	2T	0 E	LO	A	⇒ D5
6	2T	0 E	RO	A	⇒ D6
7	2T	0 E	SO	A	⇒ D7
8	2T	0 E	TO	A	⇒ D8
9	2T	0 E	WO	A	⇒ D9
10	2T	0 E	UO	A	⇒ D.
11	2T	0 E	YO	A	⇒ D+
12	2T	0 E	NO	A	⇒ D-
13	2T	22	D9	A	⇒ DA
14	2T	30	D8	A	⇒ DB
15	2T	28	D3	A	⇒ DC
16	2T	11	D9	A	⇒ DP
17	2T	0 F	ZO	A	⇒ DZ
18	2T	28	D9	A	⇒ DE
19	2T	0 F	FO	A	⇒ DF
20	2T	0 F	HO	A	⇒ DH
21	2T	0 F	KO	A	⇒ DK
22	2T	0 F	LO	A	⇒ DL
23	2T	0 F	RO	A	⇒ DR
24	2T	12	DO	A	⇒ DS (Standard exit tape read program)
25	2T	22	D26	A	⇒ DT
26	2T	0 F	WO	A	⇒ DW
27	2T	0 F	UO	A	⇒ DU
28	2T	0 F	YO	A	⇒ DY
29	2T	5	D9	A	⇒ DN
30	2T	0 F	NO	A	⇒ DD
31	2T	25	D9	A	⇒ DX

			DA	0	D18		DI	
⇒ 0			2A	22	X0			Internal subr. Type [S]
3 D19 → 1			6A	25	X2			transport link
2			6T	12	D15	14		⇒ type sign and 1st decimal
10 → 3		Y	4P		AA		P	imperative typing?
4			6Z	32	XX			next decimal ⇒ (A)
5		N	1A	0		A		
6			1B	1		A	P	digit count
27 → 7			6T	15	D1	14		⇒ TPWW
8			6Y	2	XP			type digit (or the point)
9		N	5P		BA•		Z	ensure last digit imperative
10		Y	2T	3	D18	A		→ as long as (B) ⩾ -0
18 → 11		N	4A	30	X2		Z	column count (or setting to zero)
12		Y	2T	25	X2		E	→
13			3A	30	X2			
14		U	0A	31	X2		P	not at end of line?
15		N	3S	11		A		otherwise NLCR
16			6T	15	D1	14		⇒ TPWW
17			7Z	2	XP			type space or NLCR
18			2T	11	D18	A		⇒
19			0X	19	D7	A		quasi link (see 29 D18 and 20 D19)
⇒ 20			2A	22	X0			Internal subr. Type {S}
3 D19 → 21			6A	25	X2			transport link
22			2X	31	D15			⎡ rounded-off
23			3P	25	SS			⎢ multiplication by 10^7
24			0X	1		A		⎣
25			6T	12	D15	14		⇒ type sign and 1st decimal
26			2A	14		A	P	insertion of the point
27			2T	7	D18	A		⇒ with condition affirmative
23 D21] ⇒ 28			6A	25	X2			Entrance for the
30 D15] 29			2A	19	D18			type-punch program
30			3S	10		A		after autostart 7 and 8
31			6T	9	D1	14		⇒ $0 \Rightarrow d_1$ and $0 \Rightarrow d_3$

		DA	0	D19		DI		
0		2B	1	D0				Place number to
1		2S	0	X0	B			be typed in S
2		2A	13	D1	A			quasi-link = standard exit
3		2T	25	X2			⇒	to 1 D18 or 21 D18
⇒ 4		2A	22	X0				Internal subr.Extra line blank
5		6A	25	X2				transport link
6		2A	30	X2		Z		carriage at beginning of line?
7		2S	11		A			
8	N	5A	30	X2				if not, set count to zero
9	N	6T	15	D1	14		⇒	TPWW and give extra
10	N	6Z	2	XP				NLCR-signal
11		6T	15	D1	14		⇒	TPWW
12		6Z	2	XP				NLCR
13		2T	25	X2		Z	⇒	finished
9 D2 ⇒ 14		2A	3	D0				Autostart 9
15		6A	26	X2				
16		3S	10		A			
17		6T	10	D1	14		⇒	$0 \Rightarrow d_1$ and $0 \Rightarrow d_3$
18		2S	1	D0				
19		2B	2	D0				
20		2A	19	D18				Standard entrance type-punch pr.
31 D21→ 21		6S	28	X2				
22		6T	8	D1	14		⇒	
23		6T	25	D19	14		⇒	Internal subr.Punch binary tape
24		2T	13	D1	A		⇒	Exit type-punch program
⇒ 25		6B	27	X2				Internal subr.Punch binary tape
26		2S	26	X2				
27		2A	22	X0				transport link
28		6A	29	X2				
29		6T	15	D1	14		⇒	TPWW
30	U	2LS	1		A	Z		Blank tape to be skipped?
31	N	2B	75		A			

		DA	0	D20		DI		
0	N	6T	5	D15	14		⇒	Subr. Punch piece of blank tape
1	U	2LS	2		A	Z		Skip DB?
2	N	2A	30	X14	A			
3	N	6T	2	D15	14		⇒	Punch D
4	N	6T	1	D15	14		⇒	Punch B
5	U	2LS	4		A	Z		Skip address?
6	N	2A	30	X13	A			
7	N	6T	2	D15	14		⇒	Punch D
8	N	6T	1	D15	14		⇒	Punch A
9	N	2A	28	X2				
10	N	2D	11	D7				
11	N	OS	1		A			
12	N	2B	5		A			
16 → 13	N	6Z	32	XX				⎡ cycle punch
14	N	6T	2	D15	14		⇒	⎢ address in
15	N	1B	1		A	Z		⎢ five decimals
16	N	2T	13	D20	A		→	⎣
17		2A	31		A	E		NB. Punch code positive!
18	N	6T	2	D15	14		⇒	Punch X
19	N	6T	1	D15	14		⇒	Punch 0
20	N	2S	26	X2				
21	U	2LS	8		A	Z		Skip binary words?
22	N	6T	24	D1	14		⇒	Obliterate TPWW
23	N	2B	28	X2				
29 → 24	N	2S	0	X0	B			⎡
25	N	6T	2	D14	14		⇒	Punch binary word ⎢
26	N	OB	1		A			⎢
27	N	2A	1		A			⎢
28	N	5A	27	X2		Z		number count ⎢
29	N	2T	24	D20	A		→	⎣
30		2S	26	X2		E		NB. Punch code positive!
31	N	6T	15	D1	14		⇒	TPWW

			DA	0	D21		DI		
	0	U	2LS	16		A	Z		Skip DO and blanks?
	1	N	2A	30		A			
	2	N	6T	2	D15	14		⇒	Punch D
	3	N	6T	4	D15	14		⇒	Punch blanks
	4	U	2LS	32		A	Z		Skip DS and blanks?
	5	N	2A	30	X24	A			
	6	N	6T	2	D15	14		⇒	Punch D
	7	N	6T	1	D15	14		⇒	Punch S
	8	N	6T	4	D15	14		⇒	Punch blanks
	9		6T	24	D1	14		⇒	Obliterate TPWW
	10		2T	29	X2		E	⇒	finished
9 D2 ⇒	11		2A	12		A			<u>Autostart 2</u>
	12		6A	26	X0				
	13		6A	3	X2				
	14		6A	4	X2				
	15		6A	5	X2				
	16		6A	30	X2				
	17		6A	31	X2				
	18		2S	18	D1	A			]
	19		6S	25	X0				] set to "no translation"
	20		2T	0	D3	A		⇒	end of autostart
	21		3LS	0	X0	C			X1 susceptible and all classes allowed
9 D2 ⇒	22		2A	21	D18	A			<u>Autostart 8</u>
	23		2T	28	D18	A		⇒	
⇒	24		6A	0	X1				<u>Subr.Punch binary tape</u>
26 →	25		6T	5	D1	0		⇒	Type-punch program active?
	26	Y	2T	25	D21	A		→	if so, wait
	27		0Y	64	XS				remove 6 from interruption permit
	28		2A	0	X1				]
	29		6A	26	X2				]transport parameter
	30		2A	9	X0				link in A (for the
	31		2T	21	D19	A		⇒	standard entrance)

		DA	0	D22		DI		
⇛ 0		6A	3	X2				Subr.Type (S)
6 → 1		0Y	126	XS				wait cycle for
2		2A	8		A			non-activity
3	U	2LA	26	X0		Z		of type-punch
4		2A	3	X2				program
5		0Y	0	XS				
6	Y	2T	1	D22	A		→	
7		0Y	64	XS				Standard
8		2A	1		A			entrance
9		0A	8	X0				type-punch
10		6S	4	X2		P		program
11		6B	5	X2				
12		6T	8	D1	14		⇛	
13		2B	8	X0				type code ⇛ (A)
14		2A	0	X0	B			
15		2S	4	X2		E		
16		6T	22	D22	14		⇛	to internal type routine
17		2T	13	D1	A		⇛	Exit type-punch program
⇛ 18		7B	26	X2		P		preserve (B). Internal subr.Type (S)
19		2B	22	X0				type code ⇛ (A)
20		2A	0	X0	B			
21		3B	26	X2		E		restore (B)
⇛ 22		6S	27	X2				store initial number
23		6A	26	X2				store initial type code
24		2S	22	X0				transport link
25		6S	25	X2				
26		2S	1		A			
27	N	4S	25	X2				increase link by 1
28		4S	30	X2				column count
21 D26 → 29		2A	27	X2		P		
30	U	2S	26	X2		E		
31	N	2S	13		A	E		for + sign

		DA	0	D23		DI		
0	Y	2S	12		A	E		for -sign
1	N	6T	15	D1	14		⇒	TPWW
2	N	6Z	2	XP				type sign
3		6T	15	D1	14		⇒	TPWW
4	N	2Z	2	XP				] check sign
5	Y	5P		AA		P		if it has
6	N	1S	12		A	P		been typed
7	Y	5P		AA				
8		6T	7	D25	14		⇒	Subr.Prepare for conversion
9		6A	28	X2				transformed number
10		2A	192		A			set 001 ⁝ 1000000 =
11		2S	26	X2				state+count
12		OP	3	SS				
13		6S	29	X2		P		transformed type code;non-typing cycles?
15→14	Y	6T	21	D24	14		⇒	Subr. Produce next decimal
15	Y	2T	14	D23	A		→	still more non-typing cycles
20 D24→16	U	2LA	95		A	Z		finished
17	Y	2T	24	D25	A		→	start checking
18		2S	29	X2				Punctuation
19		OP	1	SS		P		
20	Y	2T	25	D23	A		→	if 00 or 01
21		OP	1	SS		P		
22	Y	2S	14		A			if 10, type .
23	N	2S	63		A	P		if 11, type space
24		2T	27	D23	A		⇒	with affirmative condition
20 ⇒ 25		OP	1	SS		P		if 01, type nothing
26	Y	2S	12		A			if 00, type -
24 → 27	Y	6Z	2	XP				]
28	Y	6T	15	D1	14		⇒	TPWW] type punctuation
29		2S	29	X2				] shift transformed type
30		OP	4	SS				code over four places
31		6S	29	X2				

		DA	0	D24		DI		Set new state:
0	U	2LS	1		A	Z		is it I(mperative)?
1		OP	1	SS				001 if I
2		2LS	3		A			make (S). = 011 if L
3	N	OS	1		A			100 if F
4	U	2LA	896		A	Z		old state F'(=000)?
5	Y	2LS	1		A			if so F'⇒F and I⇒L
6		1P	7	AA				shift bits of new state
7		1P	20	SA				to the left of the count
19 → 8		6T	21	D24	14		⇒	Subr.Produce next decimal
9	N	3LA	256		A			I ⇒ L, if last decimal
10	U	2LA	768		A	Z		I or F'?
11	Y	OS	64		A			NB. -0+64 = +64
12		6Z	2	XP		P		type digit; not a space?
13	Y	3LA	768		A			I ⇒ L and F'⇒F
14		6T	15	D1	14		⇒	TPWW
15		3Z	2	XP				read digit back
16	N	OS	63		A			correct for space
17		4S	28	X2				add to transformed number
18		2S	29	X2		P		
19	Y	2T	8	D24	A		→	more digits in same group
20		2T	16	D23	A		⇒	end of group
⇒ 21		6A	7	X2				Subr. Produce next decimal
22		2A	28	X2				
23		2S	4		A	P		
24		5S	7	X2				count
27→25	N	OS	4		A			try to exceed
26		OA	22	D14		P		the new digit
27	N	2T	25	D24	A		→	in steps of 4
30→28		1S	1		A			go back
29		1A	31	D15		P		in steps of 1
30	Y	2T	28	D24	A		→	
31		OP	1	AA				

		DA	0	D25		DI		
0		6A	28	X2				] multiply the
1		OP	2	AA				remainder
2		4A	28	X2				by ten
3		2A	29	X2				] double the type code:
4		4A	29	X2		P		test for end of group
5		2A	7	X2				restore (A)
6		2T	22	X0		Z	⇒	finished
⇒ 7		2S	26	X2				Subr. <u>Prepare for conversion</u>
8		OP	1	SS		P		whole number?
9	Y	2T	22	X0		Z	→	
10		6A	7	X2				preserve fraction
11		2A	9		A			]
12		OP	1	SS		P		for m digits
15 → 13	Y	1A	1		A			after point,
14		OP	1	SS		P		form m+1 ⇒ [A]
15	Y	2T	13	D25	A		→	
16		2S	2		A			]
19 → 17	Y	6Z	33	XX				$2.10^m \Rightarrow [S]$
18		1A	1		A	P		
19	Y	2T	17	D25	A		→	
20		2X	7	X2				] multiply the fraction
21		1A	1		A			by 10^m and
22		3P	1	AA				round-off
23		2T	22	X0		Z	⇒	
17 D23 ⇒ 24		2A	27	X2		P		] - absolute value of
25	Y	5P		AA				initial number
26		6T	7	D25	14		⇒	Subr. Prepare for conversion
27	U	1A	28	X2		Z		Check; is it correct?
28		3S	30	X2				- column count
29		2A	11		A			
30	N	2T	15	D26	A		→	if incorrect, NLCR, etc.
31	U	OS	31	X2		P		still busy on line?

		DA	0	D26		DI		
0	N	4S	30	X2				if not, set column count
1	N	2T	9	D26	A		→	to zero, and then give NLCR
2		2A	26	X2				
3	U	2LA	4		A	Z		XS or XT?
4	N	6T	24	D1	14		⇒	Obliterate TPWW (if XN)
5	N	2T	10	D26	A		→	and type nothing (if XN)
6	U	2LA	2		A	Z		XS?
7	Y	3A	0		A			if so, -0 for space
8	N	3A	53		A			if not, -63+10 for Tab
1→ 9		6Y	2	XP		P		layout
5→10		2A	26	X2				type code used
11		2S	27	X2				number typed
12		2T	25	X2		Z	⇒	finished
20⇒13		3A	53		A			Position carriage
14		6T	15	D1	14		⇒	TPWW
30 D25→15		6Y	2	XP		P		NLCR and then Tab
16	N	3A	12000		A			
18 →17	N	0A	1		A	P		delay cycle
18	N	2T	17	D26	A		→	
19		0S	1		A	Z		count
20	N	2T	13	D26	A		→	another Tab
21		2T	29	D22	A		⇒	type again
25 D17⇒22		2S	24	D26	A			Directive DT
23		2T	8	D9	A		⇒	
2 D10⇒24		6T	21	D5	14		⇒	Subr. Analyse c Assemblage (DT)
25	U	1S	13		A	Z		
26	N	2A	12		A			if non-A
27	Y	2A	8		A			
28	Y	6T	15	D0	14		⇒	LWW (if A)
29	Y	2S	24	X0				
30	U	1S	3		A	Z		= G?
31	N	0LA	3		A			if B or E

		DA	0	D27		DI		
0	U	1S	18		A	Z		= E?
1	Y	OLA	1		A		⎤	if E
2	Y	2S	8		A		⎦	
3	N	2S	9		A	P		if G or B, set cond.affirmative
22 → 4	N	1P	2	SS			⎤	form the 4 bits
5	N	OP	2	SA				specifying punctuation
6	N	2S	24	XO				and zero suppression
7	N	1P	2	SS				for the next group
8	N	OP	2	SA			⎦	
9		6T	15	D0	14		⇒	LWW for length of group
10	N	2S	24	XO				for typing cycle ⎤ set count
11	Y	1S	24	XO				for non-typing cycle ⎦
14 → 12		OP	1	AA			⎤	add correct
13		1S	1		A	P		number of zeros
14	Y	2T	12	D27	A		→ ⎦	by shifting
15		OLA	1		A			and set a 1
16		6T	15	D0	14		⇒	LWW
17		2S	24	XO				
18	U	1S	24		A	Z	⎤	for punctuation S
19	Y	2S	3		A		⎦	
20	U	1S	31		A	Z		= X?
21		6T	15	D0	14		⇒	LWW
22	N	2T	4	D27	A		→	new group
23		6P		AA			⎤	
24		OP	2	AA				take layout
25		2S	24	XO				symbol into
26		2LS	5		A			account
27		OX	1		A		⎦	
28		6T	15	D0	14		⇒	LWW for transition
29		2T	27	D12	A		⇒	to end of assemblage
DE 30 D27 ⇒ 30		2A	11		A			Tab-tape
9,1 D28 → 31		6Y	2	XP		Z		NLCR or space

			DA	0	D28		DI		
	0	Y	1S	1		A	P		count for number of spaces
	1	Y	2T	31	D27	A		→	
	2		2A	20000		A			] delay to enable
4 →	3		1A	1		A	P		operator to set
	4	Y	2T	3	D28	A		→	tab-stop
	5		6T	0	D10	14		⇛	Assemblage subroutine
⇒	6		2T	15	X2			⇒	after detection directive
⇒	7		2T	0	D8	A		⇒	to processing cycle!
⇒	8		3A	0		A			] new number of
	9		2T	31	D27	A		⇒	spaces in [S]
⇛	10		2S	8	X0				Subr. Extra line blank
12 →	11		6T	5	D1	0		⇛	TPA? wait for non-activity
	12	Y	2T	11	D28	A		→	type-punch program
	13		0Y	64	XS				] standard entrance
	14		4P		SA				for type-punch
	15		6T	8	D1	14		⇛	program
	16		6T	4	D19	14		⇛	Internal subr. Extra line blank
	17		2T	13	D1	A		⇒	Standard exit type-punch program

In the above program the 2T-order occurs in the Z- and E-version as well. For the time being the Z-version will be equivalent to the normal case and the E-version to the P-version.

S T E L L I N G E N

1.
In een automatische rekenmachine zou de faciliteit, het teken van operaties conditioneel te kunnen inverteren, van groot nut zijn.

2.
Als Ernst Mach het begrip van de trage massa op grond van botsingsproeven invoert, laat hij onvoldoende duidelijk uitkomen, dat hiervoor botsingsproeven met drie willekeurige massapunten noodzakelijk zijn; de door hem in dit betoog genoemde experimenten zijn onvoldoende.

E. Mach. Die Mechanik in ihrer Entwicklung. 7te Auflage F.A. Brockhaus, Leipzig, 1912. pag 213-214.

3.
E.A. Guggenheim blijft in zijn "Thermodynamics" zijn belofte: "... that we do and shall consistently use symbols to denote physical quantities, not their measure in terms of particular units." niet getrouw.

E.A. Guggenheim. Thermodynamics. 2nd Edition North Holland Publishing Company, Amsterdam, 1950 pag 20.

E.A. Guggenheim. Phil.Mag. 33 (1942) pag 479.

Jeffreys & Jeffreys. Methods of Mathematical Physics. University Press, Cambridge (1946) pag 3-4.

4.
Voor het vinden van de kortste boom tussen n punten bestaat een betere methode dan die gepubliceerd door Loberman en Weinberger of Kruskal.

H. Loberman and A. Weinberger. Formal Procedures for Connecting Terminals with a Minimum Total Wire Length. Journal of the A.C.M. 4 (1957) pag 428-437.

J.B. Kruskal Jr. On the Shortest Spanning Subtree of a Graph and the Travelling Salesman Problem. Proc.Amer.Math.Soc. 7 (1956) pag 48-50.

E.W. Dijkstra. A Note on Two Problems in Connexion with Graphs. Num.Math. 1 (1959) pag. 269-271.

5.
Het slordig gebruik van automatische rekenmachines, zoals bv. door R.C. Minnick, die een tafel publiceert met in een oogopslag detecteerbare fouten, moet ten zeerste afgekeurd worden.

R.C. Minnick. Tshebycheff Approximations for Power Series. Journal of the A.C.M. 4 (1957) pag 487-504.

6.
Als in een isotroop medium een electrostatisch veld heerst, hetzij tengevolge van discrete, starre ladingen dan wel van ladingen op een geleider, dan kan een geladen voorwerp, dat uitsluitend onderworpen is aan krachten door het veld er op uitgeoefend, zich niet in stabiel evenwicht bevinden.

7.
In een algebraïsche uitdrukking kan men de prioriteit der operaties behalve met behulp van haakjes ook aangeven door aan elk operatieteken twee binaire kenmerken toe te voegen. Dit is mogelijk met behoud van de volgorde der operatietekens en operandsymbolen, zonder dat daardoor de analyse verzwaard wordt.

8.
Het is wenselijk, dat gepropageerd wordt, de symbolen "$\geqslant$" respectievelijk "$\leqslant$" uit te spreken: "minstens" respectievelijk "hoogstens".

9.
Gegeven zijn n volledig geordende elementen a_i. Uit elke greep van k elementen uit deze n kiezen we het minimale element; de - formele - som van deze $\binom{n}{k}$ elementen duiden wij aan met s_k. Dan is - formeel -

$$\max a_i = \sum_{k=1}^{n} (-)^{k-1} s_k .$$

Dit resultaat kan elegant bewezen worden met behulp van een symbolische vermenigvuldiging.

10.
Indien A een antisymmetrische matrix is van de orde 3, dan is gemakkelijk in te zien, dat geldt $A^3 = c.A$, waarbij de scaler c negatief is als de elementen van A reëel zijn. Van dit feit kan met vrucht gebruik gemaakt worden bij een beschouwing over draaiingen in een driedimensionale ruimte.

11.
In een systeem ter beschrijving van informatieverwerkende processen zij het toegestaan, dat verschillende namen dezelfde betekenis hebben.

12.
In een systeem ter beschrijving van informatieverwerkende processen zij het toegestaan, dat eenzelfde naam met verschillende betekenissen gebruikt wordt.

13.
Het is wenselijk dat onderzocht wordt of de gebrekkigheid der documentatie, waarmee veelal de industrie haar producten op de markt brengt, een gevolg is van een mogelijke oppervlakkigheid der toekomstige kopers.

14.
Bij matrixinversie door eliminatie moet, indien de berekening met drijvende komma wordt uitgevoerd, het criterium, op grond waarvan steeds de "pivot" gekozen wordt, in principe ongevoelig zijn voor vermenigvuldiging van rijen en/of kolommen met willekeurige factoren.

N.P.L. Notes on Applied Science No.16 Modern Computing Methods. Her Majesty's Stationery Office, London (1957).

15.
In programmeurshandleidingen voor (geheel of gedeeltelijk) binair werkende rekenmachines moet het gebruik van het achttallig stelsel vermeden worden.