HAHN-MEITNER INSTITUTE FOR NUCLEAR RESEARCH, BERLIN
ELECTRONICS AND NUCLEAR PHYSICS SECTIONS

Report HMI-B 96

A MODULAR PROGRAMMING SYSTEM FOR PROGRAMMING EXPERIMENTS ON THE PDP-9

K.H. Degenhardt, L. Frevert and W. Woletz

Berlin 1970

Translated at CERN by J. Nicholls
(Original: German)
Not revised by the Translation Service

(CERN Trans. 71-18)

Geneva
March 1971

HAHN-MEITNER INSTITUTE FOR NUCLEAR RESEARCH, BERLIN

Electronics and Nuclear Physics Sections

B-NDV5 HMI - B 96

Nuclear Data Processing July, 1970

K.H. Degenhardt

L. Frevert

W. Woletz

A modular experiment programming system for the PDP-9

The research work described in this report was carried out, with the support of the BMBW, within the compass of research programme NDV1 -- "The structural analysis of experiments made with the aid of a computer".

BERLIN-WANNSEE

Summary

The report contains a description, arranged in the same format as a manual, of the fundamental considerations upon which our modular system for programming computer controlled nuclear physics experiments is based. A program written with this system consists of subroutine calls interconnected by standard routines (macro-instructions), some of which may also consist of standard routines. Calls and standard routines are put into their actual form by a preassembler which assembles parameters given in dialogue by the user into prefabricated code units. One unit of a formalized language and a special flow-chart symbol represents each standard part, and hence the programs can be represented in both language and flow-chart form. Both representations can easily be derived from the parameters to be communicated to the preassembler.

The description of the programming system is supplemented by descriptions and instructions for the use of important service routines.

Contents

1. Introduction

The spur to the programme of research on "The Structural Analysis of Nuclear Physics Experiments" was given by the fact that it was recognized as impossible to control nuclear physics experiments satisfactorily by means of process computers if the language of the assembler is to be used as the sole basis for programming. Assembler programming is too time-consuming and liable to error to make a usable tool for experiment programming. Attempts therefore had to be made to create a system of sub-programs allowing flexible programming with existing units. The intended aim of the structural analysis was to give clear indications as to which units were needed for a modular system and also, if possible, to develop the basic features of such a system.

One essential factor which had to be borne in mind when the exercise was begun was that there is no problem-oriented language available for programming nuclear physics experiments and, in the present state of the computer art, it was quite possible that the prospects of creating such a language were not very bright. The reason for this, in fact, was that the program compiled for experiments would probably be too inefficient (i.e. too long and too slow) quite apart from the fact that formulation and the construction of compilers went far beyond

the scope of our research programme. Attempts were therefore first made to develop a modular program system not based on a special, problem-oriented language. The basic premise for the attempt to develop the required modular system in parallel with the structural analysis was that the structural analysis of computer-controlled experiments can effectively be carried out only with the aid of programs. Here, we hope we have succeeded in finding acceptable detail solutions. Meanwhile, we have also become convinced that it is only by means of a problem-oriented language that the problem of programming experiments can be finally solved.

The present report is divided into four parts. The first and second parts set out general basic concepts, while the third describes the special methods devised for use with a particular computer (the PDP-9). Detailed program descriptions and instructions for use will be found in part four.

2. "Modular System" problem analysis

2.1 Modular systems

For our purposes, the best definition of the term "modular" is "capable of assembly in a variety of ways from component parts". Examples of modular systems include construction kits (e.g. of the "Meccano" type), languages (in which words are the modules) and

words (in which letters are the modules). A more accurate definition of the expression "modular system" would be: "a collection of components which can be divided up into N different types is modular if any number of components of all kinds can be combined to form logical complexes of any size".

2.2 The size of the modules

The lower limit to the size of the modules into which a complex can be broken down is given by natural factors (the bit as the smallest possible module in a computer program, or the alpha-numerical symbol as the smallest module of a language). It is often useful to combine these smallest units into larger modules (e.g. to combine letters to form words). Such a combination may be carried out in several stages (bit -- computer word -- micro-instruction -- subroutine).

In general, it is true to say that:

the smaller the module in relation to the whole complexes,

the more flexible the construction;

the harder it is to grasp the representation of the complexes;

the fewer different types of modules are required;

and,

the larger the module,

the easier it is to grasp the representation of the complexes;

the more specific the properties of the modules;

the less flexible they are in use;

the wider the variety of modules required.

Where, therefore, large modules are used for the sake of clarity, a good compromise is reached by also using the type of module from which the large ones are built up (interconnecting procedure calls by statements).

2.3 Interconnection of modules

The most important factors in a modular system are the rules by which the modules are interconnected to form entire complexes. These rules determine the structure of the points of intersection between the modules and vice versa (a procedure definition with pseudo-variables defines the point of intersection of program and procedure, while the rule for the procedure call follows from the definition of this point of intersection). The more complicated such points, the more complicated the interconnection rules. For this reason, the points of intersection should be determined in the simplest possible way. In addition, the way in which the modules are interconnected to form complexes must thereafter be borne in mind when stating rules.

2.4 Representation of modular interconnections

There are two methods of representing a complex formed by the interconnection of modules, viz:

1) graphic

By "graphic representation" is meant any representation consisting solely of drawn symbols and where no connection is indicated except by graphical means. Such a representation, therefore, should in theory be completely comprehensible without the need for any additional alpha-numeric symbols.

2) description by language

A mixture of both methods (the flow-chart) is frequently used. Where linguistic means are used to describe a complex built up from modules, the desired significance and clarity can usually be attained only by the use of a formalized artificial language.

A modular complex can be properly illustrated only by a full-scale representation (block diagrams of circuits made up of digital units, ALGOL programs). It is often useful to have both a graphic representation and one expressed in a formalized language (digital circuits reduced with the aid of Boolean algebra). The advantage of the graphic methods is that relationships and connections become clearer in a two-dimensional representation. Their limitations

appear in the illustration of complexes consisting of a very large number of individual modules. Dividing the drawing into several sheets tends to make it difficult to grasp. The only remedy is to break down the complex into super-modules (theoretical circuit diagrams).

Where module complexes are described with the aid of a formalized language, the unidimensional nature of the medium makes it necessary to cast the language in such a way that structures are made clear from the linguistic description. It is only by means of language that multi-dimensional structures can be adequately described (the parallel operation of cyclic programs).

2.5 Modular programming systems

The most easily learned, most flexible and hence best solution to the problem of creating a modular programming system for nuclear physics experiments is derived from the analysis of the problem and is, in fact, the problem-oriented language. However, its formulation and the construction of a compiler to translate the programs written in the language into machine code involves a great deal of work.

It is somewhat simpler to break the experiments down into processes common to the largest possible number of experiments and to

write programs for these processes at the assembler level. That the experiments investigated can be broken down in this way was demonstrated in two research reports [3,4]. The sub-programs must be put together to form the program for an actual experiment.

The sub-programs may be written either as macro-instructions into which the parameters are inserted with the aid of macro-calls on assembly, or as subroutines into which the parameters are transferred from the parameter lists of the subroutine calls during the program run. Programming with subroutines is more rational if the same sub-programs are used several times in the same experiment with different sets of parameters.

Experience has shown that an experiment program cannot be composed purely of subroutine calls of existing sub-programs, but that standard sequences of instructions, preferably combined to form macro-instructions, must exist to link the sub-programs.

The drawback of such a programming system is that it is very much more unwieldy and liable to error than a problem-oriented language. Both faults are due to the fact that errors regarding the significance, sequence and number of parameters can easily arise in

composing the lists of parameters for the subroutine calls and macro-calls, and such errors are difficult to detect. For this reason, the inexperienced user will find such a system manageable only if he has the support of special auxiliary programs when composing his own programs.

3. Program Structures

3.1 Interrupt-controlled programs

Unlike off-line computer programs which, when once started, run in a predetermined way, on-line control and data processing programs consist of a number of routines, each activated by device flags (interrupts). Since interrupts must often be processed as rapidly as possible, important ones can interrupt the running processing of another call, which is not continued until the computer has reacted to the more important one.

The importance of messages, and hence the degree to which their processing routines can be interrupted, is allowed for by the allocation of priorities. A message with a given priority can interrupt only those routines with a lower priority, and it can, in turn, itself be interrupted only by messages with a higher priority. The processing

of messages can also be broken down into parts with different priorities. A higher-priority part can require the computer to process another part with a lower priority. Thus, the computer takes note of the problem and deals with it later after all higher-priority routines are finished.

The structure of an on-line control program becomes clearer if it is divided into sequential processes. Here, a sequential process proper should be a cycle in which the individual steps follow one another in a fixed time-order dictated by the program. Such a sequential process, for instance, might consist of the cycle: read-in magnet tape; process data read-in magnet; output of results on teleprinter. In this example, the magnetic tape control peripheral device is first activated by the processor. The end of data transmission is signalled by an interrupt to the processor, which then processes the data and finally activates the teleprinter output, so that the process is dealt with further by the teleprinter. A closer look at this cycle shows that, for instance, during the input from the magnetic tape, the peripheral device does not operate alone, but that this first step consists of a series of control instructions from the processor, their execution by the external device, the acknowledgement of their execution by means of an interrupt, and thus the activation of fresh control instruction sequences by the processor. All the operations in this

interplay between external devices and the processor follow one another, but in a rigidly fixed order.

Several of these sequential processes generally run in parallel in the actual experiment. Each process attempts to make use of an operating component in the processor/external device system. Provided that the timing of such attempts does not actually conflict, the processes can, in fact, run parallel to one another. If two processes need to use the same component simultaneously, an executive must decide, on the basis of the allocated priorities, which process is to be allowed to take place first, while the other is put on stand-by until the component can be allocated to it. If a process is using the central processor, it is withdrawn immediately if a process with a higher priority also wants to use that component. The processor thus operates alternately for all parallel sequential processes. A lower-priority process has its turn when all higher-priority processes are occupied by external operations. It is, in fact, impossible to forecast the way in which individual processes will be delayed in this way. The time-sequence of parallel-running calculations and checks cannot, of course, be completely arbitrary. The reaction encompassing two different messages, for instance, cannot take place until both have been

received. Thus, it must be possible to co-ordinate in time and synchronize the processes running virtually parallel.

The computer will frequently not be concerned with the processing of interrupts; there will be pauses during which the computer is either awaiting fresh calls or dealing with some background program not belonging to the experiment at all. The programmer's problem is to ensure that the times during which the computer is not doing any useful work are kept to the minimum.

3.2 The strategy of the allocation of priorities

Interrupts may be divided into two types for the purposes of allocating priorities to the individual interrupt operating programs, viz:-

1) Interrupts from devices which the computer cannot control in such a way as to be able to determine the frequency of the interrupts (alarms), and those specifying the beginning of a mechanical process (the indication that the magnetic tape is in position and ready for data transfer). In such cases, the interrupts must be fully processed within specific times.

2) Interrupts from devices where the computer controls their frequency and where they do not have to be processed completely within specific times.

The time condition for the first type can be fulfilled only by the allocation of a suitably high priority, whereas with the second, the priorities can be allocated in such a way that the computer is used as efficiently as possible.

The computer should react as rapidly as possible to an alarm, which should therefore be allocated a high priority.

The data from experimental equipment in nuclear physics experiments is often statistically distributed in time. The devices cannot make any further measurements until the last measured value has been read out. Since each measured value is coupled to a discrete event, the faster results are output, the fewer events are lost. For this reason, such output routines, too, should be given a high priority.

Otherwise, the allocation of priorities is a means of obtaining virtually parallel operation and thus of making full use of

the computer (this aspect is often overlooked in discussions on priorities). If peripheral devices can run out of synchronism with the computer and hence truly parallel to the processor, they should be activated and operated at a higher priority to avoid unnecessary waiting times in their parallel work. It is a logical consequence of the observation of this principle that the slower a mechanical device is the higher the prioroty at which it should be operated.

Priorities should be allocated to evaluation routines in such a way that the times during which the computer is not usefully employed are as short as possible. An example is given in the next section.

An operation divided into quasi-parallel parts is not, of course, completed until all the parts have been finished, including the one with the lowest priority. Therefore where two different operations divided into quasi-parallel parts are processed, the lowest priority occurring in them dictates which is to be completed first, i.e. which one has the higher "total priority".

3.3 The structure of nuclear physics data reduction

The salient feature of nuclear physics experiments is that the data provided by the equipment must almost invariably be evaluated statistically. Here, a very large quantity of primary data must be investigated, which must be very considerably reduced on the way from primary data to final result. In addition, it is often impossible to process intermediate results further until there are enough of them available. Frequently, therefore, it is necessary and usually highly desirable to divide the data processing cycle into individual steps. These individual steps generally involve the output from a buffer of a number of input data from which a smaller number of results data are obtained. These are then written into another buffer, there being no analytical connection between the two quantities of data (e.g. the sorting of data to form a spectrum). The complete process consists of a "chain" of such steps, each one using the input buffer of the preceding step as its output buffer.

The effect of this great reduction in the quantity of data is to reduce the frequency with which a step in the processing chain is used, assuming roughly the same length for all the intermediate result buffer stores. A given processing step will, then,

be used at time intervals which become longer, the further back the step is in the chain. The computer can do other work during such intervals.

Basically, all steps which do not alter any common data simultaneously can operate in parallel. Where every step has a common buffer with the step immediately following and one with the immediately preceding, it may not operate in parallel with these steps, although it may well do so with others. Two successive steps may operate in parallel only if they have two buffers which are always alternately filled and emptied (alternating buffers).

Where less data are passed to the computer, giving an excess of unused time, the machine is most efficiently used if the less frequent steps are dealt with in the intervals between the frequent ones, i.e. when the former can be interrupted by the latter. To this end, the less often the steps are used in the processing chain, the lower their priorities must be.

Where more data are offered to the computer than it has time to process, there is quite obviously no point in transferring them all. Such transfer can be prevented by assigning priorities in precisely the reverse order to that in the preceding example,

for example by giving input and the most frequent processed steps the lowest priorities. In such a case, however, the intervals between the use of the most frequent steps cannot be filled and the computer stands idle during these times.

The transfer of too much data is prevented in the first priority allocation system by blocking the input to each intermediate result buffer store after it has overflowed until it is emptied once more. This gives a system in which the computer is automatically supplied with as much data as it can process, and is therefore used to the best advantage. Here too, of course, the overall priority of the processing cycle is the lowest one obtaining in it.

3.4 The structure of the steps in data processing

The structure of the individual steps in a nuclear physics data reduction program is derived from the way, described in the previous section, in which the steps operate virtually in parallel.

During each data processing step, data (or parameters) are read out from one or more input buffers and written into one or more output buffers. Once an input buffer is full, the write

program must be halted and the next component in the processing chain must process all the information into the buffer forming its output store. Only then can the previous step be continued. In such a case, all the pointers to the output buffer may remain unchanged, but those of the input buffer must be reset. This is best done by a write call to the halted step in the program, with which the output buffer address is provided as a parameter.

If, however, the output buffer cannot be filled from the input stores by the processing of the data, the pointers to the output buffers may remain unchanged until a fresh read call indicates a filled input buffer.

It is useful, therefore, for a data processing step to receive the same number of read and write requests as there are input and output buffers in simultaneous use. Data transfer, accompanied at the same time by data processing, does not begin until all read and write requests have been made. The transfer cycle is stopped when an output buffer overflows, and can be continued only on receipt of the appropriate write request, just as the work is continued by a fresh read request after a buffer has been emptied.

The way in which a data processing cycle is to be finally ended remains to be explained. For this purpose, all buffers containing intermediate results should be emptied by the processing of these results. Therefore, every program step must have a request (finish) which, when given, simulates an output buffer overflow, so that the subsequent steps in processing are activated. Obviously the parts of the program then being read out must be informed of the extent to which the buffers have been filled. This is done by stipulating that each write program enters the number of data input under a buffer header before indicating an overflow, and that each read program refers to these headers.

3.5 The structure of programs for the operation of devices

Peripheral devices transfer data from data carriers into the computer or vice versa. Real data carriers, like punched or magnetic tape, supply limited quantities of data, whereas measuring instruments may be regarded as data readers from imaginary data carriers of infinite capacity. The same cycle takes place as in a processing step. Data is read from peripheral carriers and written into a buffer region in the core store, or vice versa.

Traditionally, writing into the buffer inside the computer is actuated by a read call, just as reading out of it is triggered by "write". Reading out of the data carrier is actuated by instructions like "seek data", while, in data-oriented media, "close data" is equivalent to "finish".

As in the processing steps, the data are actually transferred only if both the read and write requests for the two data stores, the carrier and core, have been made.

There are control commands, e.g. "start" and "stop", in experimental devices, in addition to the instructions initializing data transfer, as well as instructions ("init" and "close") enabling and disabling the interrupts of the devices.

3.6 Synchronization problems

The two previous sections may be summarized in the following way:

It is easy to divide an experiment program up into parts carrying out one step in data reduction or in the input or output processes. Each of these parts has various inputs. The calls to them must be made in a given order and, within the parts, operate

processes which are controlled by interrogating flags. Any further subdivision is prevented by the fact that the indicators and flags are common to the processes within the parts, a factor which renders it essential for such structures with several calls to be combined into units.

It has already been pointed out that quasi-parallel working is impossible without synchronization. Steps must be taken, for example, to ensure that two parallel processes never operate simultaneously when one of them writes into and the other reads out from one and the same buffer. This can clearly be ensured only by setting and interrogating flags common to both processes. It is possible, basically, to take out from both processes those parts which have common synchronization flags and call them an independent part of the program which takes over the job of synchronization. The flags then become internally declared variables of such a synchronization section. Sub-programs are thus produced which, for example, control the filling and emptying of an alternating buffer store (alternating buffer program).

An alternating buffer program must have at least four inputs through which it calls other parts of the program,

i.e. fill buffer 1, fill buffer 2, empty buffer 1, empty buffer 2. It must also have four outputs indicating that the called programs have completed their tasks. It is quite likely that the connection of synchronization programs to processing programs and vice versa could give rise to highly complicated structures which, where the scope of tasks becomes wider, would rapidly become impossible to grasp. After a brief series of tests, therefore, the method of synchronization employing specially written sub-programs was rejected as impracticable.

Instead, a proposal made by DIJKSTRA[1] was adopted. Here, synchronization is effected by means of operations changing the indicator variables (semaphores) common to several sub-programs running virtually in parallel. The semaphore operations are described in section 4.7.16. It will merely be mentioned here that the various parallel-running sub-programs must be interlinked by common variables.

3.7 Program interconnections and their representation

The simplest method of interconnecting two programs is by means of jumps from one to the other. It can, moreover, readily be represented purely graphically, on flow-charts. Interconnections

can also be provided, however, by the use of common variables. In principle, this second method of interconnectinn can also be reduced to program-to-program jumps, but this time with the transfer of parameters, with the variable declared in only one program. Simultaneously with a jump, its address is transferred as a parameter to the other program so that parameters common at run-time are inserted into the sub-programs. This method is, however, extremely clumsy and gives rise to very complicated systems which are difficult to grasp, since the parameters must be inserted in the programs and often even passed on from program to program before ever the actual computing process can be started.

It is better, therefore, to insert the names of variables into the sub-programs before assembly. This does mean, however, that the sub-programs must be interconnected by language units (variable names), that therefore the interconnections can no longer be represented purely graphically and that it would thus be logical to describe the entire program, in all its interconnections, by linguistic means. Nevertheless, because graphic symbols often give a more instructive representation than linguistic ones, a mixed system should be created in which the graphic symbols are partly supplemented by language units (names).

3.8 Summary of part 3.

To summarize, then, it may be said that it should be a simple matter to subdivide programs for on-line data processing and checking in nuclear physics experiments into sub-programs. The interconnections between the modular sub-programs are such that they cannot be represented by graphic symbols alone. A modular experiment program must, therefore, be described linguistically to some extent. To eliminate ambiguity and to preserve meaningfulness, the language used for such a description should be highly formalized.

4. A Modular Programming System for the PDP-9

4.1 The form of the interrupt handling routines

It has already been said that an on-line program for checking a nuclear physics experiment consists entirely of interrupt handling routines. The interrupts bring about calls to the device handlers from which a further call is then made of devices by calls to device handlers, giving rise again to new interrupts.

When an interrupt occurs, the contents of all registers used by the interrupt handling routine must first be saved, and they are once more restored before returning to the interrupted program.

If several devices are connected to an interrupt channel, the computer must first decide which device or devices caused the interrupt. It codes the hardware interrupt into logical interrupts. Each of the latter has its own processing routine, and they are processed in succession.

Saving and restoring the register, and coding the hardware interrupts are standard processes which should be carried out, not by the user program, but by a standard executive. The interrupts are then processed partly by the executive and partly by the user program. Each logical interrupt causes a call from the executive to a part of the user program. Once this part has been processed, a return must be made to the executive.

There are two alternative methods of producing a return to the executive, viz:-

1) A jump is made to a given address in the executive. This implies that there is a definite division between the executive and user program.

2) The jump from the executive to the user program is made by subroutine calls and the reverse by returns.

In a modular system, the interrupt processing system in the user program is made up of individual units. There are two ways of making it up, viz:-

a) The individual units of their calls are written one after the other and are thus passed through in succession.

b) The individual units or their calls represent formal subroutines. Each one jumps to the next at each subroutine call; subroutines are nested.

In the case of combination 1-a the sequence of the individual units must be completed by a return (exit) into the executive. With combination 2-a the sequence of the individual units must be integral with a subroutine. After the innermost subroutine has been processed, the subroutine returns in combination 2-b take place in the opposite order to the subroutine calls and lead back to the executive. Combination 1-b is pointless.

In combination 2-b, the borderline between executive and user program is ill-defined, whereas there is a clear distinction between the two in combinations 1-a and 2-a. With the latter two, therefore, a decision must be taken on whether device handlers are to be considered part of the user program or the executive, while such a differentiation is unnecessary in the case of 2-b.

4.2 Preparing for interrupt handling

The handling of a logical interrupt must be indicated to the executive by informing it of the connecting address for interrupt handling in the user program. This is often implicitly included in programming systems at assembler level by a wait call, so that the user program is continued with the instruction following the wait call on encountering the logical interrupt. Providing the connecting address on the actuation of the external device operation ended by the logical interrupt is an alternative here.

The connecting address can be transferred by means of a subroutine call in which it is one of the parameters to be communicated. Such a method gives the following sequence: The user program activates an external device by calling the device handler and gives as a

parameter the point in the user program to which the handler is to jump when the end of the operation is signalled by a logical interrupt. In its turn, the device handler calls the hardware interrupt decoding program and informs it of the address in the device handler to which a jump should be made once the interrupt has been decoded. From the point of view of the user program, preparation for the processing of the interrupt takes place in nested subroutines. Which of the nested subroutines is considered still part of the user program or already part of the executive is immaterial. This method of preparing for interrupts, used together with form 2-b (section 4.1) of the interrupt handlers, takes place in the following way (fig. 1.).:-

An external device operation is initiated by a subroutine call of the device handler in which the connecting address is one of the parameters. This connecting address must be that of a part of a program in the form of a subroutine. A subroutine call is used if a jump is made from this part of a program to a further sub-part, e.g. to a sub-program of a modular system. Here, pre-programmed sub-programs are informed of their continuation address as a parameter when the call is made in exactly the same manner as the device handlers. Of course, the continuation address here, too, is that of a formal subroutine. The logical interrupt, by means of which the

external device signals the end of its work, causes the nested subroutines to be run through until a new external device operation is activated or the machine must wait for the execution of another quasi-parallel operation, or until the work is finished. The new external device operation is once more activated by a subroutine call with the connecting address as a parameter. The wait is brought about by a subroutine call which interrogates a memory variable (see 4.7.16.5). In neither case is a direct jump into a new subroutine made, and thus the return to the executive takes place from these innermost subroutines through the subroutine return sequence. If the sequence of nested subroutines is not ended with a wait or external device call, the innermost subroutine to be used must be one consisting of only one subroutine return (EXIT).

4.3 Advantages and drawbacks of nesting the subroutines

It has already been shown that the borderline between the user program and the executive disappears in this cycle where parts of the executive and the device handler and user programs are nested as subroutines. If it is decided to call everything which need not be programmed by the user the executive, the latter can be extended at any time by a process of logical nesting, involving the once-for-all addition of the part written by the user. In the first stage, for

example, the executive may be extended by an ADC handler which, in its turn, will form the nucleus for a multi-channel program already containing simple data reduction routines. If this collection is also considered part of the executive, the extended executive additionally includes an external multi-channel device called in the same manner as the other external devices and indicating back to the user when its work is finished by means of a logical interrupt.

The number of logical interrupts which the user program can await is not limited by the features of the system. Any number of external devices may be activated simultaneously. This means that any number of sequential processes can run in parallel in the user program, their number's being unrestricted by the length of executive lists.

There is a drawback to subroutine (SR) nesting in that SR calls with their parameter transfer and also the SR returns take time. It must be possible to recall the evaluation programs repeatedly because each of them calls a further SR to indicate its completion and can therefore also (implicitly) call itslef, i.e. the program must be re-entrant.

As the executive has no means of accounting for the sequential processes that have been started, each sequential process can change only its own priority, not that of the other processes.

The user must, of course, accustom himself to the fact that he must always explicitly supply the address of the subroutine with which his program is to be continued whenever he gives a subroutine call. This minor inconvenience, however, is compensated for by the external uniformity of external device calls with evaluation routine calls thus achieved. The user should always bear in mind that returns in the subroutines used sérve to provide a return to the executive.

4.4 "Waiting"

It will frequently be impossible to complete a process activated by an interrupt immediately if, in fact, another interrupt or the result of a calculation must be awaited. In the second case particularly the computer may not remain in a wait loop but should do useful work, e.g. finish the calculation of which the result is "awaited", until receiving the event which has to be awaited.

Processing the interrupt must, therefore, be broken off by the execution of the subroutine returns of all nested subroutines through until that moment, so that the part of the program [illegible] by the interrupt is continued. Care must also be taken to ensure that the processing of the remainder of the interrupt is activated by the event to be "awaited". More details on this point will be found in the description of synchronizations (4.7.16).

4.5 The construction of a modular programming complex

The hierarchical construction of our modular system is of the two-stage type. Its lower stage contains device handler and evaluation program with the structure described in sections 3.4 and 3.5. Each consists of a number of subroutines belonging logically together. For these programs to run, they must be provided with parameters, e.g. the addresses of data fields and the continuation address. This is effected by "calls" which contain the subroutine calls with parameter lists and, according to their external configuration, also represent subroutines. The calls must also be interconnected in order to provide a usable complex. Short standard subroutines (SSR) which can be regarded as macro-instructions (indeed, some of them are so defined) are used to this end. The programs are thus called and interconnected by a higher-order program of calls and SSR's.

It is the user's problem to insert parameters generating the desired experiment program into the calls and the SSR's forming the "cement" between the calls. In this he is assisted by a special program (preassembler, see 5.1.1) which fetches the calls and SSR's from a library, describes the necessary parameters to the user and thereby requests them, checks the parameters provided by the user for format errors and inserts them into the calls and SSR's, which can then be assembled. The PDP-9 loading program loads the programs appropriate to the calls at the same time as it loads the experiment program from the library.

All SSR's with internal labels not appearing on the outside are defined as macro-instructions. Here, the parameters to be provided by the user are inserted into the prefabricated macro-instructions. The internal labels are generated by the assembler on assembly and named. The remaining SSR's are formed from prefabricated sequences of instructions.

The sub-programs themselves, which are called by the calls, are written, as far as possible, with the aid of the SSR's. They also contain sections written in assembler code. If certain rules are observed, they may also contain parts written in FORTRAN.

Every SSR, with very few exceptions, contains one or more subroutine calls which jump into standard parts or calls which carry on further. The points to which they jump represent parameters to be inserted by the user. With every call, a jump point must be given as a parameter, to which the called program jumps at each subroutine call once it has completed its task. Since a jump point may also be a call in the program with a different set of parameters, it must be possible to recall the programs repeatedly, i.e. the program must be re-entrant.

An experiment program made up of SSR's and calls may be divided into super-modules which, with slight modifications, and if suitably arranged, can also be used in other experiments.

4.6 Representation of the program complexes

It has been found that faster programming is often achieved through a graphical representation of the SSR interconnections. Linguistic representation is suitable in other cases, especially for programming synchronizations. It was explained in section 3.7 that not all the interconnections occurring in the structure of our problems can be represented purely graphically. We therefore arranged for series of letters (variable names) to be included in some of the graphic symbols.

To allow the user to take advantage of both methods of representation, we have tried to develop comprehensible graphic symbols for our standard subroutinesand also to give linguistic formulations so that, depending on the actual problem involved, one or the other method of representation, or both together, can be used for programming. We thought it important in this connection for each basic unit in both methods to correspond to a symbol or linguistic combinations, so that either method of representation can be transformed into the other without any special mental effort. The method used is immaterial on the input of the parameters into the preassembler. The graphic representation of a program need not, therefore, be translated into the linguistic one, but a program can be created in dialogue with the preassembler simply on the basis of its graphic representation.

4.7 Description of the standard subroutines (SSR)

4.7.1 General

The first factor to be mentioned in the description of an SSR will be its purpose, followed by an example of an excerpt from a program written in the form

SSR type labels generated linguistic symbols.

By SSR type is meant the combination of symbols which must be passed to the preassembler so that it can search for the SSR under this type name in its library. The SSR has at least one call label formed by the preassembler from characteristic letters (generally the type code) and the SSR number to be provided by the programmer. Where SSR's have several call labels, they are differentiated by the addition of letters A, B, etc. Calls setting and resetting the flags are labelled with the suffix .T or .F. Examples : P1, P2, S1A, S1B, S1C, S1.F, S1.T.

"Texts" briefly indicate the function of the SSR and contain the parameters which must be inserted by the programmer. Nearly all SSR's end with "go to" followed by the address of the subroutine to which a jump must be made after the SSR has been executed, per subroutine call. Reference is made to the graphic symbols of the SSR, beside which the SSR flow-charts are shown in the conventional representation.

The "texts" may be considered as language units in a programming language. No rigid rules of syntax are imposed, since this programming language is not translated into a computable program by a compiler. The formulations were chosen in such a way as to provide similarities to existing programming languages.

The number, sequence and syntax of the parameters to be input are determined by the indication of the SSR type. These parameters are given in the correct order in the "texts".

For the purposes of the input of parameters into the preassembler, it is best to arrange the SSR's by types, their sequence being completely arbitrary in the source program composed by the preassembler, since then the preassembler can carry out its search in the SSR library much more rapidly. As, however, a program arranged by SSR types is not easily understood by another user, the program is reduced in the course of rearrangement to indications of type and parameter. It is only these data, underlined in the "texts", that are inserted into the preassembler. An additional advantage here is that time is saved in punching the cards.

4.7,1.1. Indirect parameter transfer; global parameters

The names of semaphore variables and data (fields) can indirectly be transferred by the SSR or calls. In other words, the addresses at which the parameter names are to be found may be given instead of the names themselves on the input of the parameters. In the latter process, therefore, whether the address of the name or the name of the parameter itself was given must be indicated by the addition of YESindirect or NOindirect.

The names of semaphore variables and data may be declared as global names valid for several, separately translated programs. They are declared in one of the programs in which they are an internal global address, and must be indicated as global addresses in all programs by the SSR GLOBL.

External global names declared in another program must be specified YESindirect, to be transferred indirectly, when input as parameters of SSR's and calls.

4.7.1.2 Declaration of data fields

Data and parameter fields must be declared in a data directory to be written for every experiment program. The data directory formally represents a program with the name DATBnn, where nn is the segment number in segmented programs. In unsegmented programs, nn = 1. The names of data fields are external addresses for all programs using them.

The form

field name 0; field length .GLOBL field name

is prescribed for entry in the data directory.

A core allocator connected with the executive divides the core stores not occupied by programs into data fields, their lengths' corresponding to the entry "field length" which is given as a decimal number. The core store allocator enters the calculated initial addresses of the data fields, under the address "field name", in the data directory from which they are taken by the processing programs.

When the field lengths are given, it must be remembered that each data field must have a header into which those programs which write-in data enter the number of data words actually inserted.

Fields may be declared as overlapping:

The directory entry

field name 2 field name 1; .GLOBL field name 2
50

means that a core store area of 50 words receives the same core store address as the previously declared field with the name "field name 1". The data directory must be closed with the exit symbol -1.

4.7.1.3 Instruction input

Every instruction consists of a keyword and the actual instruction. The keywords must be explained in a directory ADRBnn,

The entries in the directory must be made in the following form:

.ASCII 'KEY' (15)

.DSA KEY; .GLOBL KEY

Here, KEY is the keyword (of up to four symbols) defined as the address in the decoding section of the user program (subroutine). Part of the system program (ERSTnn), the first to be loaded by the user, uses the keyword directory to fetch the address of the decoding section and, with the aid of the decoding section, to fetch the address of the subroutine to be started as a result of the instruction.

The exit symbol

-1

must be entered in the keyword directory.

4.7.2 Program serialization P

Use: activation of "parallel" - running sub-programs with the same priority.

Example:

P p1 go to 3 branches: G1.P, P6, P7 *)

SSR type: P

SSR no.: 1

where nn is the segment number of segmented programs, and nn = 1 for unsegmented programs.

Label generated: p1

Execution: jumps are made to the indicated subroutines (up to 7) in succession (fig.12).

*) Note: On the input of parameters into the preassembler, a total of 7 branch addresses must be given, those which do not apply being provided as dummies.

4.7.3. Indirect jump-out J

Use: jump to an address adopted as a parameter

Example:

J j1 Jump out by N1B

SSR type: N

SSR no.: 1

Label generated: j1

Execution: Indirect subroutine jump via address N1B, into which the jump point was written as a parameter by SSR N1 (fig. 3).

4.7.4 Parameter transfer N

Use: transfer of parameter lists from subroutine calls.

Example:

N nl get 5 parameters by label INIT

SSR type: N

SSR no.: 1

Labels generated: nl; nla, nlb, nlc, nld, nle

Execution: 5 parameters are transferred, behind the last call made of subroutine INIT (only fifteen-bit addresses are permitted as parameters). The parameters accepted are found under addresses nla, nlb, etc. Transfer is carried out with API and PI disabled by .CB or .DA (fig. 3).

4.7.5 Print M

Use: issue of messages via teleprinter.

Example:

MELD ml print INPUT ERROR

SSR type: MELD

SSR no.: 1

Label generated: M1

Execution: A subroutine issuing the message via a teleprinter is added to a queue at main program level (fig. 4).

4.7.6 Lower priority (request level) R

Use: Add a subroutine to a lower-priority queue.

Example:

R r1 request level 5 for P1

SSR type: R

SSR no.: 1

Label generated: R1

Execution: Subroutine P1 is added to the queue for level 5 and started later (fig. 4).

4.7.7 Raise priority L

Use: Raising the priority.

Example:

L ℓ1 raise priority to level 3 and go to P1

SSR type: L

SSR no.: 1

Label generated: L1

Execution: The priority is raised to 3 and a subroutine jump is made to P1. After the return from P1, the priority is lowered to its original value (DBK) (fig. 5).

4.7.8. Gate G

Use: Making a continuing subroutine call only when a condition is satisfied.

Example:

G gl if gl.f do nothing else go to P1

gl.f do gl.f

gl.t undo gl.f

initially YES gl.t

SSR type: G

SSR no.: 1

Labels generated: gl, gl.f, gl.t

Alternatives to initial condition: NO gl.t

Execution: A flag is set or reset by jumps gl.t and gl.f. Depending on the value of the flag, a jump to the label given as parameter is made or not after jump gl. (fig. 2).

4.7.9 Switch S

Use: Conditional branching

Example:

S slA if sl.f go to P1 else to P2

sl.f do sl.f

sl.t undo sl.f

S slB if sl.f go to P3 else to P4

SSR type: S

SSR no.: 1

Labels generated: sla, sl.f, sl.t (the last two only if "arm A" is stated).

Execution: Jumps sl.f and sl.t set and reset flag. At jumps sla, slb, ..., branching occurs, dictated by the value of the common flag. The "arms" are continuously indexed A, B, etc. (fig. 2).

4.7.10 Decode instruction D

Use: Starting sub-programs by means of the input of instructions via a teleprinter. Each instruction consists of a keyword and the actual instruction itself.

Example:

D decode with key ATLA 4 commands

on command GO go to P1

on command END go to S1A

on command LOS go to S1B

on command CLOS go to S1C

(Up to seven instructions per keyword may be set*)

SSR type: D

Labels generated: none

Global address: keyword

*Note: The input must be parameter pairs for seven instructions (unused ones as dummies).

Execution: A decoding directory is generated containing the command word (up to four symbols) in 5/7-ASCII followed by the address of the sub-program to be started by the user as a parameter. The final symbol of the decoding address directory is -1. The keywords are also written in 5/7-ASCII followed by the appropriate decoding section addresses in a directory to be written by the user (with the available auxiliary program). Decoding is carried out by part of the executive ERSEn (n = 1, ... as a segment number is segmented programs) which adds the subroutine address found during decoding to a queue running at main program level.

Note: The subroutine address given in the decoding section must contain 0 in order to be entered in the queue; do not, therefore, make subroutine jumps in the program (Fig. 3).

4.7.11 Declaration of program label M

Use: Declaration of call labels, in subroutines to be translated separately and written by the user, as external addresses for other parts of the sub-programs.

Example:

M declare label MARA = P1

SSR type: M

Program label (global address): MARA

Execution: A SSR in the form

```
MARA    0
        .GLOBL MARA
        JMS P1
        JMP* MARA
```

is generated (fig. 4).

4.7.12 Declare global parameter GLOBL

Use: Declaration of parameters to (internal or external) global parameters.

Example:

GLOBL BUF1, BUF2, SPEK

SSR type: GLOBL

Execution: The parameter names given are regarded as global addresses.

4.7.13 Declare internal buffers BUF

Use: Declaration of intermediate buffers inside the program.

Example:

BUF declare BUFFER with 100 words

SSR type: BUF

Buffer length: 100 (decimal)

Execution: An entry is generated which is the same as the entries of the core allocation in the data directory:

```
BUFFER    .DSA  ..0013
          100
          .DEC;  ..0013        .BLOCK 100;    .OCT
```

4.7.14 Save and restore parameters SAV

Use: To obtain subroutines which can be re-entrant.

Each subroutine so written jumps at every subroutine call to an SSR or call address indicated in the call to indicate the completion of its work. The program may, via this subroutine jump, pass to a call which calls the same sub-program with a different set of parameters and thus writes in the flag and any other parameters in the sub-program. Before the subroutine call, these parameters must be saved, to be restored after return.

Example:

SAV savl save entrance INIT (having delivered 4 parameters) and 4 locations: P6, READER, PAR1, DATUM; done go to P5

SSR type: SAV

SSR No.: 1

Execution: The program is initialized by the subroutine call INIT; There are four parameters in the call after INIT, and these must be transferred. Five cells must be kept free behind these parameters for the entry of the content of cell RUAD inside the SSR and the contents of the four locations given. The content of INIT (the return address) is saved in RUAD. The program is then continued by a subroutine jump to P5. After the return from this subroutine, which may also implicity call INIT, the saved contents are restored with the aid of the content of RUAD which is taken back to INIT (flow-chart, see fig. 3).

4.7.15 And-interconnection A

Use: Further jump to a subroutine only if two actions with the same priority have taken place in any order.

Example:

A a1a if a1b done undo both, go to P1

   a1b if a1a done undo both, go to P1

initially YES both done

SSR type: A

SSR no.: 1

Labels generated: ala, alb

SR jumped to: P1

Initial conditon: jump made to both inputs

(alternative NO both done)

Operation (fig. 5): the jump to subroutine P1 is made only if jumps are successively made to ala and alb. There is an initial exception after the program has been loaded. The further jump takes place when a jump is made to one of the two inputs.

4.7.16 SSR's for coordinating sequential processes

It was stated in section 3.1 that an experiment program consists of sequential processes running virtually parallel. The work of these processes must be coordinated in time. Thus, for instance, two processes may not attempt to change the same variables simultaneously or, in other cases, two phases of two processes must take place in a given time-sequence. It must be possible, in both instances, for one of the processes to "wait", where conditions require it.

Coordination is effected in our system according to a proposal made by DIJKSTRA[1] with the aid of special variables, the semaphore variables, which can only be integers and can be

changed only by special operations which cannot be interrupted. Semaphore variables may be declared as global addresses. There are the following SSR's or macro-instructions:

4.7.16.1 Declaration of a semaphore variable SEM

Use: Declaration of a semaphore variable.

Example:

SEM declare sema SEMNAM

SSR type: SEM

Execution: Two computer words with the identifier SEMNAM are kept free. The content of the first word is set at -1 and that of the second 0.

4.7.16.2 Sequence declaration SEQ

Use: Declares a subroutine at the beginning of a sequential process and allocates a name and priority to it.

Example:

SEQ Declare sequence WORK with priority 5

beginning with P1

SSR type: SEQ

Execution: A sequence heading with the address WORK is generated.

The sequential process will begin with SSR P1, started with priority 5.

A subroutine

```
WORK ∅
     JMP .+2
     5
     JMS P1
     JMP* WORK
```

is gener ted (fig. 6).

4.7.16.3 Activate call ACT

Use: Activating (starting) a sequence.

Example:

ACT act 1 activate WORK; go to ACT2

SSR type: ACT

SSR no.: 1

Activated sequence: WORK

The program is continued with SSR ACT2.

Execution: The sequence WORK is started with the priority given in the declaration by a subroutine call if the activate call has the same priority as or a lower priority than the sequence; otherwise, it is added to a queue. Thereafter the program is cnntinued with the SSR ACT2 (fig. 6).

4.7.16.4 Assign statement ASS

Use: Assigning an initial value to a semaphore variable.

Example:

ASS assl assign to SEMNAM (NOindirect) 10; go to ASS2

SSR type: ASS

SSR no.: 1

Semaphore variable: SEMNAM

Allocated value: 10

The program is continued with SSR ASS2.

Execution: The first word of the semaphore variable SEMNAM is set to 10 and the second to 0 (fig. 6).

4.7.16.5 Dyke call DYK

Use: "Waiting" until a semaphore variable is not negative.

Example:

DYK dykl dyke SEMNAM (NOindirect); go to P1

SSR type: DYK

SSR no.: 1

Semaphore variable: SEMNAM

The program is continued with P1 (fig. 6).

Execution: The semaphore variable SEMNAM is interrogated by a

jump to a system subroutine DIJKST. If its value is ≥ ∅, it is reduced by 1 and a jump is made to subroutine P1. If its value is negative, it is not changed and subroutine P1 is entered in a queue which can be compiled for any semaphore variable. The second word of each semaphore variable contains the indicator to the first entry in the queue, and each Dyke call contains the indicator to the next entry in the queue belonging to the appropriate semaphore variable. The end of the queue is given by the pointer ∅. The entries in the semaphore queue are in order of priorities (see description of DIJKST, 5.2.4).

The jump to DYKER takes place with API and PI disabled. They are both enabled again before return.

4.7.16.6 Undyke call UDY

Use: Providing the indication that "waiting" is not (or no longer) necessary.

Example:

UDY udyl undyke SEMNAM (IOindirect); go to P1

SSR type: UDY

SSR no.: 1

Semaphore variable: SEMNAM

The program is continued with SSR P1 (fig. 7).

Execution: The value of semaphore variable SEMMAN is increased by 1. If this causes the value $\emptyset$ to be attained, a check is made on whether there is an entry in the queue appropriate to the semaphore variable (second word of the variable $\neq \emptyset$). If so, the value of the semaphore variable is once more reduced by 1 and the SR to which a jump should be made from the first waiting DYKE is started or entered in a starting queue depending on whether the priority of the Undyke call is lower (or equal to) or higher than that of the waiting DYKE. The next item in the Dyke queue is advanced to first position.

4.7.16.7 Activating cyclic processes

Sequences consisting of closed cycles may be activated provided that the following rules are observed.

1) The work of an external device must be contained in the cycle, or

2) the cycle must contain a Dyke call, the semaphore variable of which has such an initial value on activation that the Dyke call is awaited; the cycle can then be started later by an Undyke on the same semaphore variable.

4.7.17 Input of any texts TEXT

Use: Input of any texts, e.g. subroutines written by the user in assembler code.

Example:

```
TEXT     READ    0
                 .
                 .
                 .
                 JMP*  READ
```

SSR type: Text

Execution: The text provided by the user is regarded as a parameter of the statement "TEXT" to the preassembler and inserted by the preassembler into the program generated.

4.7.18 Call C...

Use: Calling sub-programs with lists of parameters

Example: (fig. 7).

```
CADCR    c1  Call adc1  to read into BUF1
                 (NOindirect);  finished go to P1
                 (NO indirect)
```

SSR type: CADCR

SSR no.: 1

Label generated: c1

Execution: A subroutine

```
C1              0

YES = 400000

NO  = 0;        JMS* ADC1R;       .GLOBL   ADC1R

                JMP  .+3

                .DSA BUF1+NO

                .DSA P1+NO

                JMP* C1
```

is generated with which the sub-program ADC1R is called. It is a part of the operating program of ADC1. The calling of ADC1R has the effect of causing ADC no. 1 to read data into buffer BUF1. When buffer BUF1 is full, a logical interrupt is generated which continues the user program with SSR P1.

The parameters buffer address and jump address may be external addresses or given indirectly (i.e. given by parameter addresses instead of in the form of the parameter itself) (YESindirect).

4.7.19 End of a sequence EX

Use: Closing a sequence

Example: (fig. 7)

EX ex1 exit

SSR type: EX

SSR no.: 1

Execution: a subroutine

```
EX3     Ø
        JMP*  EX3
```

is generated.

4.7.20 Jump to external address E

Use: Jump to an explicitly specified label of a sub-program translated separately.

Example:

```
E       e1      jump out to ENABLE
```

SSR type: E

SSR no.: 1

Execution: A subroutine

```
E1      0
        JMS*  ENABLE;    .GLOBL ENABLE
        JMP*  E1
```

is generated (fig. 4).

4.8 Example: multi-channel program

The basic program described here is an analog-digital converter (ADC) operating program containing calls for the ADC, which stores the measured values of the input pulse height, converted to binary by the ADC into a data field. Together with this is a sorting program which allocates a memory location in an output field (the spectrum) to each binary number and always adds 1 to the content of the spectrum word if there is a corresponding binary number in the input data field. Here, the spectrum words are arranged in the order of the size of the possible input values. Both programs are intended to operate together on the alternating buffer system, so that the ADC program fills one buffer, while the other is being processed by the sorting program. It must be possible to call the program thus produced in the same way as a device directly providing pulse-height spectra (multi-channel analyser).

The ADC is signalled to the operating system and the alternating buffer operation initialized. The INIT part of the program is called by two parameters, i.e. the address of a start-stop handler capable of starting or stopping other devices simultaneously with the multi-channel, and an address to which a jump is made when the INIT part is ready.

All parts of the multi-channel analyser (MCA) are called at level 5.

M		declare label MCA1I = P1
P	p1	go to 2 branches: N1, L1
N	n1	set 2 parameters by label MCA1I
L	l1	raise priority to level 3, go to C1
CADC1	c1	call adc1 to init with start-stop-handler N1A (YESindirect), go to R1
R	r1	request level 5 for ASG1
ASS	asg1	assign to P1EMPTY (NOindirect) -1, go to ASS 2
ASS	asg2	assign to P2EMPTY (NOindirect) 0, go to ACT1
ACT	act1	activate sequence RDLOOP, go to UDY1
UDY	udy1	undyke P1EMPTY, go to J1
J	j1	jump out by N1B

Read-in takes place by blocks into the intermediate buffer at level 1, and this buffer is processed parallel to it at level 5.

SEQ		declare sequence RDLOOP with priority 1 beginning with DYK1
DYK	dyk1	dyke P1EMPTY (NOindirect), go to C2/ A wait is made here until the first intermediate buffer is clear
CADCR	c2	call adc1 to read into BUF1 (NOindirect), done go to ACT2

ACT	act2	activate WORK1, go to DYK2
DYK	dyk2	dyke P2EMPTY (NOindirect), go to C3
CADCR	c3	call adc1 to read into BUF2 (NOindirect), done go to ACT3
ACT	act3	activate WORK2, go to DYK1
SEM		declare sema P1EMPTY
SEM		declare sema P2EMPTY
BUF		declare BUF1 with 100 words
BUF		declare BUF2 with 100 words
SEQ		declare sequence WORK1 with priority 5 beginning with C4
CSRTG	c4	call sorting program srt1 to get data out of BUF1, go to UDY2
UDY	udy2	undyke P1EMPTY (NOindirect) go to EX1
EX	ex1	exit
SEQ		declare sequence WORK2 with priority 5 beginning with C5
CSRTG	c5	call sorting program srt1 to get data out of BUF2, go to UDY3
UDY	udy3	undyke P2EMPTY, go to EX2
EX	ex2	exit

The sorting program, which has already received the GET call on initialization, is set in operation by the PUT call from the read part of the program. The terminating signal is given when one of the spectrum words overflows.

M		declare label MCA1R = P2
P	p2	go to 2 branches: M2, C6
N	n2	get 2 parameters by label MCA1R
CSRTP	c6	call sorting program srt1 to sort into spectrum M2A (YESindirect), on overflow go to J2
J	j2	jump out by M2B

The spectrum overflow is simulated in the terminating part of the program.

M		declare label MCA1F = P3
P	p3	go to 2 branches: M3, C7
N	n3	get 1 parameter by label MCA1F
CSRTF	c7	call sorting program to finish (to simulate overflow), go to J3
J	j3	jump out by M3A

In the termination part, the intermediate buffers are emptied after the ADC has acted as if it had filled its buffer as a result of its CLOSE call. A spectrum overflow is then simulated by the termination of the sorting program and the program is reset to its initial state.

M		declare label MCA1C = P4
P	p4	go to 2 branches: N4, ACT4
N	n4	get 1 parameter by label MCA1C
ACT	act4	activate END, go to L2
L	l2	raise priority to level 3, go to C8
CADCC	c8	call adc1 to close, go to R2
R	r2	request level 5 for J4
J	j4	jump out by N4A

Level 0 is awaited in the sequence END until both intermediate buffers are empty. The END loop is stopped by level 0.

SEQ		declare sequence END with priority 0 beginning with DYK3
DYK	dyk3	dyke P1EMPTY (NOindirect), go to DYK4
DYK	dyk4	Dyke P2EMPTY, go to R3

```
R         r3          request level 5 for C9
CSRTC     c9          call sorting program srt1 to close, go to EX3
EX        ex3         exit
```

At START and STOP, the ready message addresses are taken over and the priority allocated, since the ADC program is called at level 3 except on READ.

```
M                     declare label MCA1A = P5
P         p5          go to 2 branches: M5, I4
N         n5          get 1 parameter by label MCA1A
L         l4          raise priority to level 3, go to C10
CADCA     c10         call adc1 to start, done go to R4.
R         r4          request level 5 for J5
J         j5          jump out by N5A
M                     declare label MCA10 = P6
P         p6          go to 2 branches: M6, L5
N         n6          get 1 parameter by label MCA10
L         l5          raise priority to level 3, go to C11
CADCO     c11         call adc1 to stop, done go to R5
R         r5          request level 5 for J6
J         j6          jump out by M6A
EXIT
```

Fig. 8 is a graphic representation of the program. From it, all parameters for insertion into the preassembler may also be derived. Fig. 9 is a representation of the ADC program. An alternating buffer program is shown for purposes of comparison in Fig. 10. This program undertakes the synchronization processes in parts MCAlI and MCAlC of the program instead of in the user program via the executive.

There are four jumps in program SRTl, SRTl is re-entrant. Jump SRTlG is called with two parameters, viz: the address of the buffer containing the data words to be sorted into a spectrum, and the address designed to activate SRTl when all data words from that buffer have been sorted. Jump SRT2P is also called with two parameters, i.e. the address of the buffer containing the spectrum and that to be activated by SRTl when one of the spectrum words overflows. Overflow simulation is effected by a third call SRTlF. The overflow is also simulated on call SRTlC, while, in addition, the program is reset to the initial state.

4.9 Reduction of programs consisting of SSR's

A program composed with the aid of the preassembler consists of a number of standard subroutines and the appropriate calls. Here, there will very probably be a number of once-only subroutine calls within this program.

There is, however, no point in subroutines without parameter transfer unless they are called several times, since they require three more memory locations (SR call, entry point, return) than the corresponding sequences of instructions replacing the SR calls. Examples have shown that the storage requirements of a program generated by the preassembler can be reduced by about 30% if once-only SR calls are replaced by the subroutine sequence of instructions (without entry and returns).

A previously translated program in the form of a listing must be used as a starting point in the reducing process because the macro-calls in this program are supplemented by the macro-framework (of the program itself) into which the assembler has inserted the actual parameters during translation. The macro-definitions are thus rendered superfluous and can be deleted, as can the macro-calls. Then, an investigation must be made at each

subroutine call to find out whether the subroutine concerned is called again at another point in the program. If not, it is not a true subroutine and the subroutine code (without entry and return) may be inserted instead of the subroutine call. The name of the subroutine also disappears with the entry.

A specially written auxiliary program, the reduction program CUSORT, deletes from a listing lines which have become irrelevant and rearranges once-only subroutines instead of their calls. This program reads-in the listing, presented in the form of ASCII text, of the program to be reduced and, on reading-in, erases all lines which have not led to an entry in the address or code column of the listing.

The address and instruction codes are also deleted on reading-in; of the listing, therefore, only the source program remains. The latter is interrogated for all subroutine calls found several times in the program, and these are entered on a list. The source program is then issued, line by line, as a new, reduced source program. Output lines are deleted from the original source program.

Whenever a subroutine call is to be issued, reference is made to the list to find out whether it occurs once only in the program. If so, the subroutine concerned is issued instead of the subroutine call; excluding the entry line and its corresponding return.

5. Appendix; Program description and instructions for use

5.1 Preassembler and program generator

5.1.1 The preassembler

5.1.1.1 Purpose

The preassembler assembles macro-definitions, prefabricated macro-calls and standard subroutines found in a library with the parameters provided by the user to form programs. The parameters to be supplied by the user are described by texts also taken from the library. Input can take place either in dialogue with the computer or with the aid of parameter lists written off-line.

The parameters provided by the user are checked for correct format (e.g. their composition from permitted symbols). Corrections may be made by on-line dialogue. Experience has shown that source programs composed with the aid of the preassembler hardly ever contain any format errors.

5.1.1.2 Method of operation

The first requirement made of the user by the preassembler is the name of the program to be composed. It then reads-out all the macro-definitions from the library and issues them as the first part of the program. It then requires the user to indicate an SSR type, which it seeks in the library and from which it obtains the descriptions of the parameters which must be inserted in the SSR by the user in order to make it part of a processable program. It requests each parameter, describes it, checks it for format errors, indicates such errors and expects corrections, and from these parameters builds up a list of them into an SSR. Once all the parameters for an SSR have been provided, it asks for the input of four up-arrows, and then inserts the parameters into the prefabricated SSR text which is also fetched from the library. Thereupon, it requests the next SSR type.

5.1.1.3 Separator

Each of the parameters provided by the user must be enclosed between separators, for which the preassembler uses up-arrows (7-bit ASCII code: 136). Whenever the preassembler asks for a parameter by its description, all the symbols provided by the user are read up to the first separator, which signals the beginning of the parameter.

As a check on the number of parameters input for an SSR, the sets of parameters in the individual SSR's are separated from one another by four up-arrows, requested from the user by the preassembler in dialogue.

After the input of the name of the program (enclosed in separators), the preassembler also requests a line of four up-arrows.

5.1.1.4 Completion of parameter input

Once all the SSR's of a program, with their parameters, have been provided by the user, the signal EXIT is input instead of the four up-arrows required by the preassembler, which then finishes the program issued with .END, prints out on the teleprinter the number of SSR's composing the program and goes into a wait loop from which it can once more be started (CONTROL P). The monitor is loaded on the insertion of any other symbol via the operating teleprinter.

5.1.1.5 Input and output devices

The preassembler uses the following .DAC SLOTS (logical input and output channels) :

.DAT SLOT	Function
-2	Input of instructions for use
-3	Output of error messages, requests for instructions for use, output of parameter descriptions in dialogue
1	Input of macro-definitions, SSR's and parameter descriptions from the library (.LIBR MAC)
2	Output of the finished source program
3	Input of the parameters
5	Output of the program statement

5.1.1.6 Program file

The parameter descriptions may be issued by the preassembler together with the parameters provided as a program file. Its output can be stopped (see operation). In it, the SSR's and parameters are continuously numbered in such a way that the last three figures of a parameter number indicate the number of the parameter in the SSR and the first three that of the SSR in the program.

The program file is issued so that it may be used to reinsert the parameters (fig.11).

5.1.1.7 Operation

After the start, the preassembler announces itself by

PROGRAMPREASSEMBLER (fig.12).

It then requires the user to reply to a few questions to determine its mode of operation. The questions may be answered YES or NO. In an abbreviated mode, the RETURN symbol (ASCII 015), given as a reply, corresponds to the YES.

The abbreviated mode is set up when the preassembler's first question

SWITCH FAST SETTING YES OR NO

is answered by YES.

The next question is

IMPROVE VIA TELETYPE.

If the answer is yes, the preassembler awaits the insertion of the corrected parameter again after each format error.

The question

MONITOR OUTPUT

needs no answer.

The question

COMMENT LONG

signifies that there are two descriptions for each parameter in the library, one detailed and one abbreviated (for experienced users).

The question

COMMENT VIA TT

enables the parameter description to be suppressed from .DAT SLOT -3 when it is unnecessary, for instance when the parameter input is via a punched tape written off-line.

When the reply NO is given to the question

EVERY PARAMETER,

the user is given the opportunity of correcting parameter lists issued in the form of a program file and at the same time to create the program provided from the corrected parameters. To this end, the preassembler must be informed, on request, of the number of the first parameter to be altered, and this may be taken from the printed-out program file. The preassembler uses the old program file input via .DAT SLOT 3 as a parameter list and first asks the user to indicate the parameter to be altered by describing it.

After its input, the number of the next parameter to be altered must be provided, and so forth. The preassembler issues the new program file and, on output, creates the new program on a magnetic store in a second pass, using the new program file as a parameter list (the two passes are necessary in this mode since otherwise four files - old and new program files, library and new program - would have to be opened at once).

5.1.1.8 Names of parameter lists

Once the questions determining the preassembler working mode have been answered, the preassembler asks for the names under which it can find the parameters on the input medium and under which the program file is to be issued. Names may consist of six alphanumeric symbols. PLS is accepted as an extension (see program file manual). The up-arrow separator must be used to indicate the end of the input of the parameter list name. After these preparations, the actual work of the preassembler, described in section 5.1.1.2, begins (fig.12).

5.1.1.9 Error correction

On the input of incorrectly expressed parameters, the nature of the error is indicated and, with the correction mode

operational, a fresh input of parameters is expected. This cycle is repeated until the parameter input contains no format errors.

If there is an error in reading, the incorrectly read line is printed. It may be either transferred unaltered (RETURN), or deleted [D(RETURN)] or corrected by typing the correct text.

5.1.1.10 Phase errors

After the parameters for an SSR have been input, the preassembler waits for the input of a sequence of four up-arrows as a separator. If this sequence is provided too early (insufficient parameters) or too late (too many parameters), the error message PHASE ERROR is printed out via the operating teleprinter with the serial number of the SSR in whose set of parameters the phase error was discovered. The SSR with the incorrect number of parameters is not inserted into the program. Since the parameter following a sequence of four up-arrows is invariably the type designation of an SSR, the parameter inserted after incorrectly given sequences of up-arrows is interpreted by the preassembler as an SSR type designation.

5.1.1.11 Erroneous SSR type designation

If the preassembler is given an SSR type which does not appear in the library, it gives the error message NOT FOUND. In dialogue, the question IS THIS TO BE CORRECTED? is then printed via the operating teleprinter. If the answer is YES, (or the RETURN symbol), the SSR type designation may be repeated. If it is NO, and on the input of off-line written parameter lists, the preassembler skips all inputs up to the next sequence of four up-arrows, and interprets the parameter following those as the next SSR type designation.

5.1.2 Writing the SSR library

5.1.2.1. Library name

The preassembler looks for the SSR library on the external store (magnetic tape or disc) under the name .LIBR MAC.

5.1.2.2 The configuration of the SSR library

The SSR library is formed by an ASCII text divided into lines. The individual SSR's are entered in random order. Each SSR in the library consists of (fig.13):

1) two successive lines, each of four up-arrows;

2) the SSR type designation;

3) the descriptions of the parameters;

4) a line of four up-arrows, and

5) the actual SSR.

The actual SSR is written as a normal part of a program. Every sequence of symbols which, when the SSR is inserted into a real program, are to be altered (formal parameters) must be enclosed in two up-arrows.

Each parameter description consists of:-

1) a detailed text;

2) an abbreviated text as an aid to memory;

3) the sequence of symbols of the appropriate formal parameter;

4) the check code.

Each part of a parameter description is termined by an up-arrow. The parts may be written one after the other in a line or distributed over several lines. The SSR type designation must be terminated by an up-arrow.

5.1.2.3 Check codes

The check codes inform the preassembler about the aspects used in testing the actual parameters fed in, which are to be inserted into the SSR instead of the formal parameters. There are check routines in the preassembler for the following codes:-

1) MC6 A check is made to see whether the actual parameters satisfy the criteria laid down for variable names in the assembler manual.

2) PN Check as in 1); in addition, the sequence of symbols input is interpreted by the preassembler as the name of the program generated.

3) PN5 as 2); however, with a maximum of only five symbols.

4) JON YES or NO are expected as input.

5) GO2 Exactly two octal figures must be typed be the user.

6) GO1 One octal figure must be typed by the user.

7) DZ2 A maximum of two decimal figures must be typed by the user.

8) NOT No check.

Back-arrows are transcribed in all tests. Frameworks of formal programs (see 5.1.3) can thus also be generated.

5.1.2.4 Use of macro-instructions

SSR's containing internal labels or addresses not appearing externally and intended to be given another name every time the SSR is inserted in a program are best written as macro-instructions, so that the assembler can generate a new name at every call (see assembler manual). The macro-definitions are entered in an SSR with the type designation MACDEF, containing no formal parameter and automatically input by the preassembler (it must therefore be included, empty, in the library, even when no macro-definitions are used) (fig. 14).

The macro-calls appropriate to macro-definitions are processed like an ordinary SSR.

5.1.2.5 The indication of errors in the SSR library

The preassembler indicates a few errors which may have arisen in writing the SSR library:-

EDERR indicates that a formal parameter found in the SSR does not correspond to any of the formal paramters in the parameter descriptions.

CODE NOT FOUND means that a check code not shown on the preassembler's code list is given in one of the parameter descriptions.

IOPS 01 is given when a parameter description does not finish with the prescribed up-arrow according to the check code.

5.1.3 The program generator

The program generator inserts actual parameters in the place of formal ones in prefabricated formal programs containing formal parameters. In this way, many actual programs, e.g. operating programs for similar devices, can be generated from one formal program. By and large, the program generator can be used to alter any texts at previously determined points with a particular aim in view; indeed, a text is altered in the same way at all points indicated by the same formal parameter.

The program generator operates in exactly the same way as the preassembler, and therefore any description of it is superfluous. The only differences are these:-

1) The program generator uses back-arrows as separators (ASCII code 137).

2) Once the working mode has been determined (5.1.1.7), the program generator requires the user to input an explanatory comment.

3) In the case of the program generator, the "formal programs" correspond to the BSM's. They must be arranged individually

as files in an external store under a name with the extension FPR. The formal programs are constructed in exactly the same way as an SSR entered in the SSR library (5.1.2.2).

4) There is a check code BIN instead of the GOl (5.1.2.3). When this code is given in the description of the formal parameters, the program generator requires the input of any number of (different) decimal numbers from 1 to 16 and from them generates an octal number with which the insertion points of a line control, numbered from 1 to 16, can be called on a multiple address system, i.e. the bits indicated are put in the bit group 1 - 16 (PDP-9 word; bit numbering from left to right, beginning with bit 0 and going up to 17), while the computer word thus produced is recoded as an octal word in the alphanumeric code.

5.2 Executive

5.2.1 Instruction decoding ERSTnn

In segmented programs (nn = segment number), the decoding program ERSTnn must always be the first to be loaded. On being loaded, it brings the remaining parts of the executive with it from the program library.

ERSTnn is started by CONTROL S via the teleprinter. At its first start, it initiates the software queue handler and, by making a jump to the core store allocation program DATINT, divides the free core store with the aid of the data directory (see 4.7.1.2) into the fields given therein. It then goes into a wait loop in the ORG program.

At every new start given by CONTROL'S, it awaits the input of an instruction keyword (see 4.7.1.3), fetches the program address specified by the instruction following the keyword and, when CONTROL S is repeated, arranges the appropriate subroutine in the uppermost supervisor queue SORQ10.

5.2.2 Supervisor ORG

The supervisor ORG organizes the processing of up to eighteen queues at main program level (queue handler SORQnn with nn = 10 ... 31 (octal)) Processing of the queues is started by ORG in the order of their importance (with SORQ10 as the most and SORQ31 the least important After a program from a queue has been processed, ORG checks to find out whether a more important queue has been initialized meanwhile and starts its higher-priority processing.

5.2.3 Queue handler SORQnn and SORAnn

Queue handlers SORQnn (with nn as the level number) and SORAnn arrange subroutines required at higher priority levels in queues from which the lower-priority subroutines are being processed. Here, nn may assume the values 4 - 31 (octal). Processing the queue at level 4 - 7 is started by software requests, and those at levels 10 - 31 by the supervisor ORG called by the queue handlers.

Addresses SORQnn and SORAnn denote different entries into the same handler with the number nn. Unlike SORA, SORQ enables the API and PI again.

Call (example):

```
        ISA+10              /API OFF
        IOF                 /PI  OFF
        JMS*   SORQnn
        .GLOBL SORQnn
        JMP    .+2
        .DSA   SR           /ADDRESS OF SR TO BE QUEUED
```

Operation:

The subroutines requested are entered in a chain-structured queue. Each link in the chain indicates the following one, while the end of the chain is indicated by a special entry. After all higher-priority work has been done, the subroutines are processed in the order of their entry. The first cell of the subroutine serves as storage cell for the pointer. Each SR may not, therefore, be entered more than once in the chain. Subroutines where the first unit is $\neq 0$ are not entered without the subsequent issue of an error message.

5.2.4 Synchronization program DIJKST

The program DIJKST carries out the organizations necessitated on activations, Dyke calls and Undyke calls. It possesses four entry points: DYKER, UNDYKER, ACTIV and ASSIGN.

5.2.4.1 DYKER

Call:

```
ISA+10              /API OFF
IOF
JMS*   DYKER
.GLOBL DYKER
```

```
                .DSA    DYKER
                .DSA    SEMA        /SEMA VARIABLE (+4000000, IF INDIRECT)
                -1                  /PRIORITY ENTERED HERE
                0                   /NEXT UNIT IN QUEUE ENTERED HERE
                .DSA    DYKEWAIT
                .DSA    DYKEFURTHER
                .DSA    HILEVEL
                .DSA    SAMLEVEL
                .DSA    LOLEVEL
HILEVEL         0
                JMS     SAMLEVEL
                DBK
                JMP*    HILEVEL
SAMLEVEL        0
                LAC     SAMLEVEL
                DZM     SAMLEVEL
                DAC     RUAD
                SKP
DYKEFURTHER     ISZ     COUNT
                JMS     SR          /CONTINUING SR
                LAC     COUNT
                SAD     (0
```

```
                JMP*   RUAD
                TAD    (-1
                DAC    COUNT
DYKEWAIT        .....          /NEXT INSTRUCTION
COUNT           0
LOLEVEL         =SAMLEVEL
RUAD            0
```

Operation:

DYKER finds out whether the semaphore variable SEMA has a positive value. If so, the value is reduced by one, the second of the exit addresses given as parameters (DYKEFURTHER in the call example) is fetched, and an exit made to the given address. If the semaphore variable has a negative value, the content of the priority cell is fetched. If the latter is -1, the priority of the level at which the DYKER call took place is calculated and entered in the priority cell as a binary number. This entry is made only when the DYKER call is executed for the first time.

Thereafter cell 2 of the semaphore variable is checked. The semaphore variable consists of two cells. (The value of the variable is entered in the first, while the second indicates the

first unit in the appropriate waiting list.) If it is 0, the address of the call priority unit is entered in it. If the address of another DYKER call is already entered in cell 2, and thus a queue has already been opened, the units of this queue are examined for their priorities and the call which has just been processed is arranged according to its priority in the queue. The latter is itself in the form of chain. The semaphore variable indicates the first unit, and each subsequent unit the following one. The units are arranged in the order of priority of the DYKER calls by DYKER.

After the units have been arranged in the queue, an exit is made from DYKER to the first of the transfer vectors given in the call (DYKEWAIT in the example). The subroutine SR called after DYKEFURTHER is thus started immediately after the DYKER call only if the value of the semaphore variable is positive. Otherwise, the start can be given only by an UNDYKER call with the same semaphore variable.

The subroutine SR, in its turn, may (implicitly) call a further subroutine, which causes the DYKER call to be run through again. This repetitiveness is to be permitted by the code around JMS SR.

DYKER once more enables the API and PI.

5.2.4.2 UNDYKER

Call:

```
        ISA+10
        IOF
        JMS*   UNDYKER
        .GLOBL UNDYKER
        JMP    .+3
        .DSA   SEMA        /+4000000, if indirect
        -1
```

Operation:

UNDYKER raises the semaphore variable SEMA by one. If the value 0 is thereby obtained, UNDYKER checks to find out whether a queue had been opened for the semaphore variable (word 2 of the semaphore variable $\neq 0$). If so, the semaphore variable is once more reduced by one, and, on the first execution of the UNDYKER call, the priority with which the call was made is calculated and entered in the second parameter cell. The first entry is then taken from the queue, its priority (calculated by DYKER) fetched and the difference in priority from UNDYKER call calculated. Depending on the result, the last subroutine in the DYKER call address list is arranged in a software request queue, or the last but one or last but two

subroutine is started, after the priority has been suitably raised. Starting in such a case takes place in the following way: the address immediately following the last parameter of the DYKER call is input in the first entry of the subroutine concerned and a jump is made to entry 2 of the subroutine.

If the priority was raised in UNDYKER, it must be lowered once more by DBK in the subroutine (the DYKER call HILEVEL in the example).

A program arranged in a software request queue is not arranged for the second time by the queue handler until its first entry has the content 0. For this reason, this transfer entry is erased in the subroutine SAMLEVEL (= LOLEVEL).

UNDYKER enables the API and PI.

5.2.4.3 ACTIV

Call:

```
ISA+10
IOF
JMS*   ACTIV
```

```
            .GLOBL ACTIV
            JMP    .+6
SEQAD       .DSA   SEQ          /ADDRESS OF SEQUENCE TO BE STARTED
            -1                  /PRIORITY ENTRY UNIT
            .DSA   HILEVEL
            .DSA   SAMLEVEL
            .DSA   LOLEVEL
            with
HILEVEL     0
            JMS    SAMLEVEL
            DBK
            JMP*   HILEVEL
SAMLEVEL    0
            JMS*   SEQAD
            JMP*   SAMLEVEL
LOLEVEL     =SAMLEVEL
```

and

```
SEQ         0
            JMP    .+2
            5                   /PRIORITY 5
            JMS    SR           /FIRST SSR IN THE SEQUENCE
            JMP*   SEQ
```

Method of operation:

Like UNDYKER, ACTIV establishes whether the priority of the sequence SEQ to be activated is higher, the same or lower. Again, like UNDYKER, it ensures the start of the corresponding subroutine (HILEVEL, SAMLEVEL or LOLEVEL) from the call address list. A DBK must be made in HILEVEL. The sequence (SEQ in the example) must carry the entry of its priority at the prescribed place in its head.

ACTIV enables the API and PI.

5.2.4.4 ASSIGN

Call (example):

```
ISA+10
IOF
JMS*   ASSIGN
.GLOBL ASSIGN
.DSA   SEMA        /+4000000, IF INDIRECT
3                  /ALLOCATED VALUE
```

Method of operation:

ASSIGN enters the value given as a parameter in the call into the first word of the semaphore variable SEMA and erases the

second word of the semaphore variable. The indication of any possible Dyke queue is thereby erased (the indications to the next unit in each unit of the waiting list chain are erased or transcribed by DYKER when a fresh entry is made).

ASSIGN enables the API and PI.

5.3 Reducing program CUSORT

The program CUSORT abbreviates programs containing subroutines called once only by setting in order the sequences of instructions in the subroutines instead of their calls. In doing so, it uses the listing of the translated program as a basis. In addition, all lines which have not directly generated a code in the listing (macro-definitions) are erased.

The source programs generated by CUSORT can be translated without error if the following rules have been observed in the original program:

1) Non-executable statements must be in a line together with an executable statement (e.g. .GLOBL MAIN; ONE = 1).

2) Each subroutine may contain only one return instruction, which should be at the end of it.

3) Labels may not be placed in front of subroutine returns:

instead of

```
END     JMP*  SUBROU
```

the program should read

```
END=.

        JMP*  SUBROU
```

4) No instruction within the subroutine may refer to the subroutine itself (e.g. LAC* SR)

CUSORT uses the following .DAT SLOTS:

1 Error messages

2 Operating inputs

3 Input (listing)

4 Output (new source program)

After starting, the program requires a program name (LST is accepted as an extension).

Error messages:

BUFFEROVERFLOW the program to be read in is too long.

FILE LST NOT FOUND the listing of the program given has not been found.

In both cases, a different program name is requested.

READ ERROR — an error in input. The incorrect line is issued and can be corrected (input of the new line via the operating teleprinter) or accepted (input of (~~CARRIAGE~~ RETURN)).

5.4 Parameter handler PARHAN

With the aid of PARHAN, integers and real numbers can be given as output or input in decimal form via the teleprinter (and can thus be checked and changed). This gives single and double precision and any data field length.

Let it be assumed that the names of the parameters and parameter fields of a user program PRG are in a list PARLI to which PARHAN has access. The further organization is as follows: In his program PRG, the user calls program PARHAN and checks the initial address of the parameter list:

PRG

```
        JMS*   PARHAN;     .GLOBL PARHAN
        JMP    .+2
```

```
        .DSA    PARALI
        .
        .
        JMP*    PRG
```

The form of the parameter list is:

```
PARALI  N
        NAME1
        NAME2
```

Here, N is the number of parameters or parameter fields in the list. In the case of the ith parameter, the parameter description and value (or the parameter values in the case of fields) are to be found in NAME1 and the following lines:

```
NAME1   VALUE1
        200000
        0
        .ASCII  "DESCRIPTION" <15>
```

VALUE1 600002

0

0

0

0

0

0

The length of the parameter field is given on the right in VALUE1. The significance of the bit positions 0 and 1 is:

00 : integer

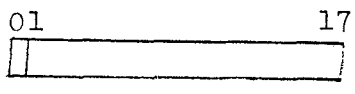
01 : integer double precision

10 : real

11 : real double precision

600002 typifies, for instance, a parameter field of two real numbers of double precision. The parameter value (or the values in the case of fields) in the dual representation within the PDP-9 is stored in units VALUE+1 et,seq. This representation is in the form:

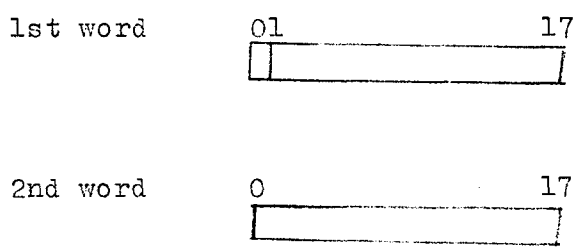
a) integer:



Bit 0 contains the sign (0 = plus, 1 = minus).

The absolute value of the integer number is stored on the right in the remaining 17 bits. Thus $-131071 \leq$ integer number $\leq +131071$ (maximum 7 places).

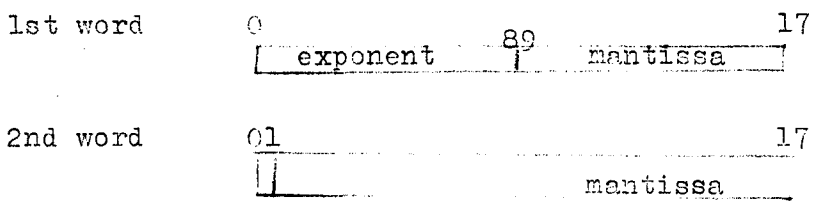
b) double precision integer:



Bit 0 in the first word contains the sign, and the remaining 35 bits the absolute value of the integer number.

$-2^{35}+1 \leq$ integer number $\leq +2^{35}-1$ (maximum 12 places).

c) real:



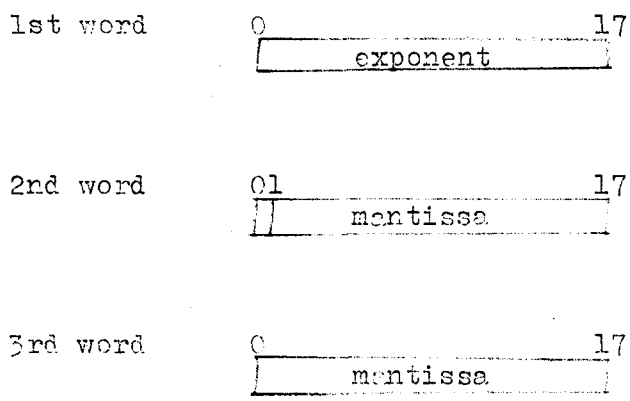
Bits 0 to 8 of the first word contain the 2-complement of the exponent, and bit 0 of the second the sign of the mantissa.

The mantissa is found in the remaining 26 bits (beginning with bit 1 of the second word and ending with bit 17 of the first one). Here, the mantissa is in the "normalized" form, i.e. for a number $\neq 0$, there is always a 1 in bit 1 of the second word.

The method of writing is as in FORTRAN:

$\pm$0.123456E$\pm$12, i.e. 6 places after the decimal point are accurate (maximum 13 places).

d) double precision real:



The first word stores the 2-complement of the exponent and bit 0 of the second word the sign of the mantissa. The remaining 35 bits contain the mantissa in normalized form. Method of writing: $\pm$0.123456789D$\pm$12, i.e. 9 places are accurate (maximum 16 places).

Care must be taken on the input of parameter values to ensure that no more places are used than given above, and that, in the case of real numbers, the form

±0.--------E±--

or

±0.--------D±--

is chosen.

After the jump to PARHAN, the description of the first parameter in the list and its value are printed out. The input of (RETURN) leaves the value unaltered and permits the issue of the next parameter with description and value (or the next parameter value in the case of a field). If the input is a value (followed by RETURN or ALT MODE), PARHAN checks for format errors (but not all of them!), e.g. the number of places, the size of the numbers, and the correct method of writing (in the case of real numbers) etc. If an error message is given, PARHAN waits for a fresh input until the value is accepted. The input of the symbol ALT MODE (alone or after the input word) causes a return to the calling program PRG.

5.5 Standardization of the names of program labels

It has been found an advantage to apply a standard system, indicating the function carried out, to the names of entry points

into the programs. Since, with the PDP-9, program labels may consist of a maximum of 6 alphanumeric symbols, the first one always being a letter, the following standard was laid down:

The final symbol in the name indicates the function of the part of the program:

A	START
C	CLOSE
F	FINISH
G	GET
I	INIT
O	STOP
P	PUT
R	READ
W	WRITE

All other letters and figures may be used as desired.

The one or two-figure program number is given before the close symbol. The program itself is designated by the (up to) three initial symbols, the last of which must be a letter. E.g. MCAlI is the INIT call of multi-channel program MCA No.1.

Bibliography

1) E.W. Dijkstra: Co-operating Sequential Processes (in: Programming Languages, NATO Advanced Study Institute, Ed. by Dr. F. Genuys; Academic Press, London and New York (1968)).

2) PDP-9 MACRO-9 Assembler
(Digital Equipment Corporation Program Library No. DEC-9A-AMZA-D).

3) L. Frevert: Zur Strukturanalyse kernphysikalischer Experimente als Grundlage für ihre Steuerung durch einen Digitalrechner (On the structural analysis of nuclear physics experiments with a view to controlling them by means of a digital computer). (BMwF-FB K 68-50 (1968)).

4) L. Frevert: Eigenschaften eines Programmsystems für rechnergeführte kernphysikalische Experimente, hergeleitet aus der Analyse eines einfachen Experimentes (Features of a programming system for computer-controlled nuclear physics experiments, derived from the analysis of a simple experiment). (BMwF-FB K 69-22 (1969)).

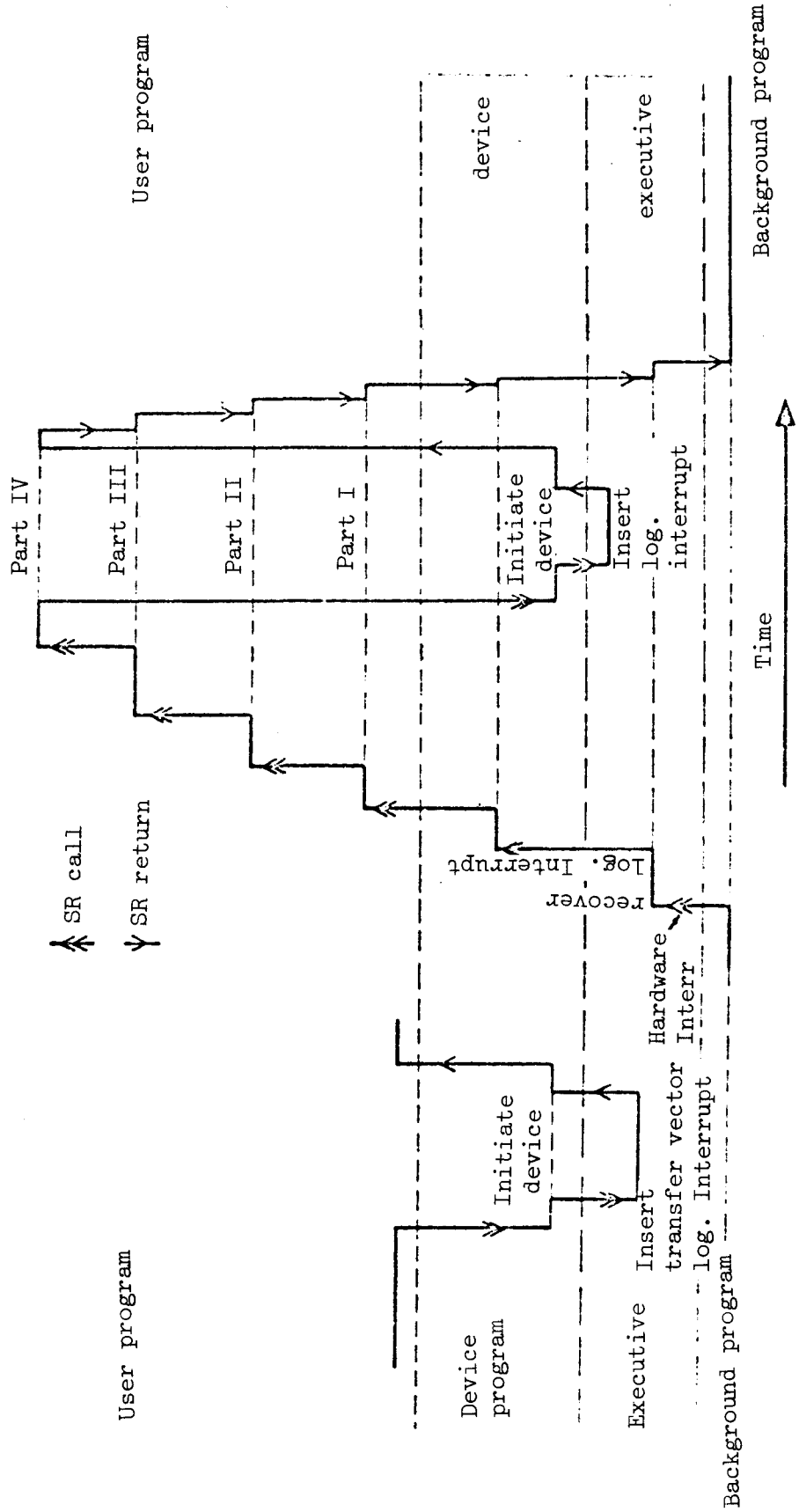


Fig. 1 Interrupt processing

In the part on the left, the user program calls a device program and thus prepares the logical interrupt. The hardware interrupt of the equipment later causes the jump to the executive which generates the logical interrupt into the device program from which a jump is made into parts I, II, III and IV of the user program. A device is initiated from part IV, whereupon a return is made into the background program.

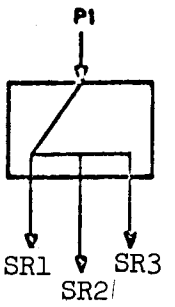
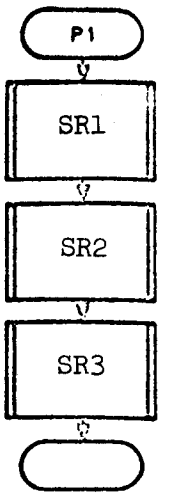
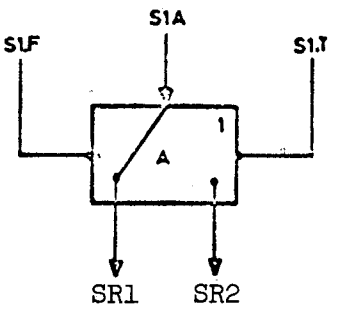
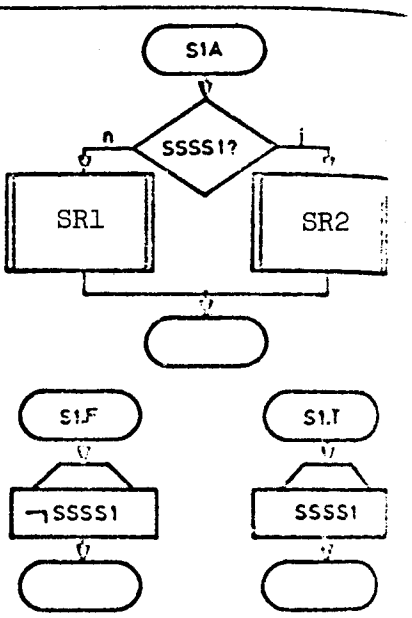
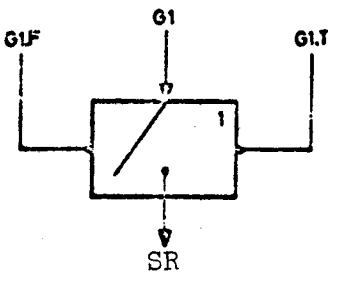
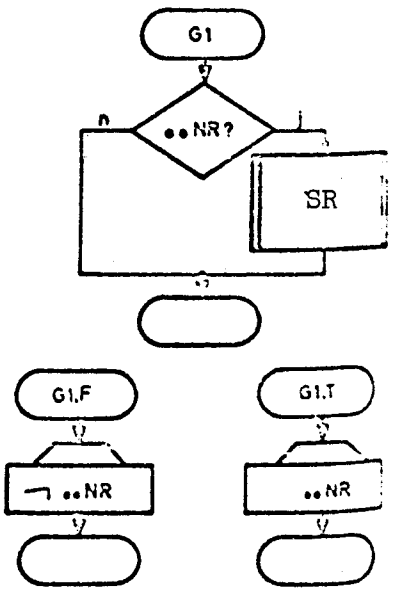
P Up to m. 7 subroutine connections possible		
S Further branching with interrogation of SSSS 1 are continuously numbered SB 1 ... SZ 1		
G Initial condition after loading (..NR or ¬ .. NR) may be selected		

Fig. 2

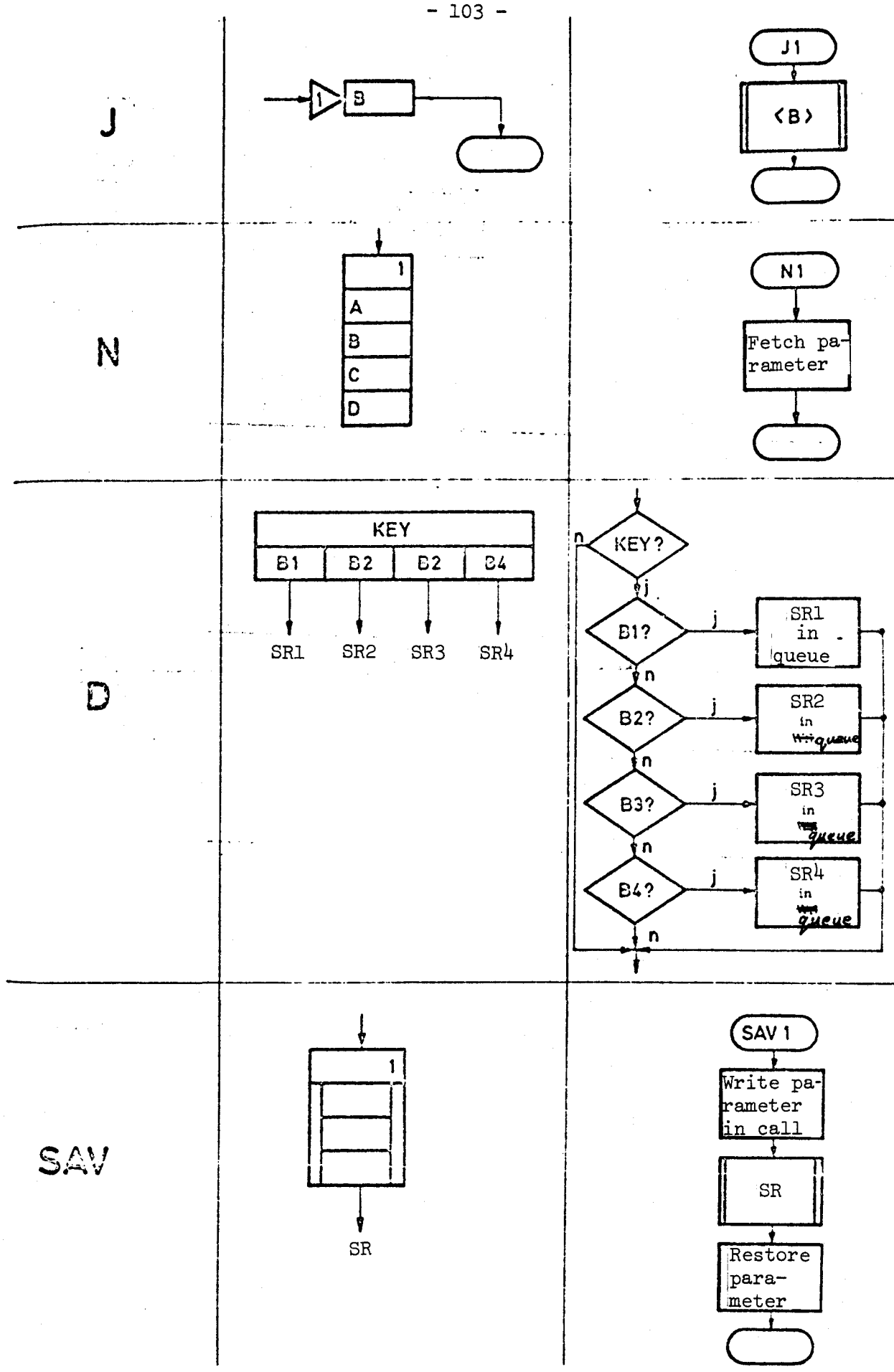

J
1
B
J1
<B>
N
1
A
B
C
D
N1
Fetch parameter
D
KEY
B1
B2
B2
B4
SR1
SR2
SR3
SR4
KEY?
n
j
B1?
SR1 in queue
B2?
SR2 in queue
B3?
SR3 in queue
B4?
SR4 in queue
SAV
1
SR
SAV 1
Write parameter in call
SR
Restore parameter

Fig. 3

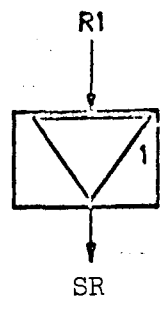
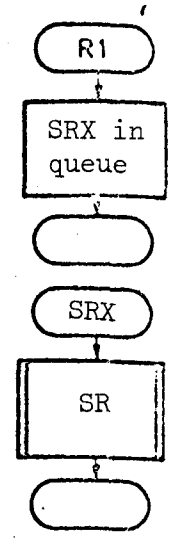
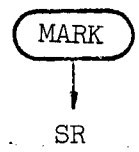
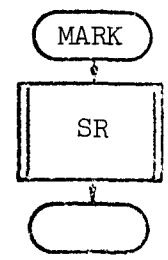
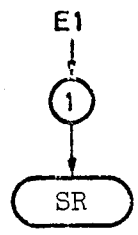
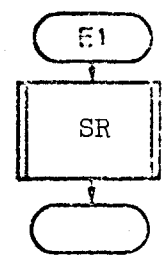
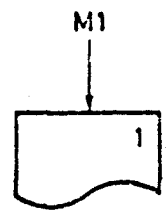
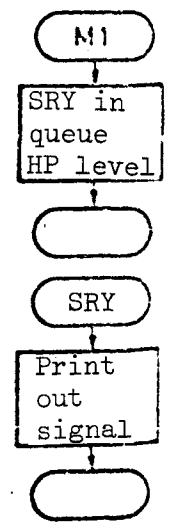
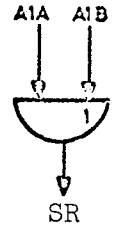
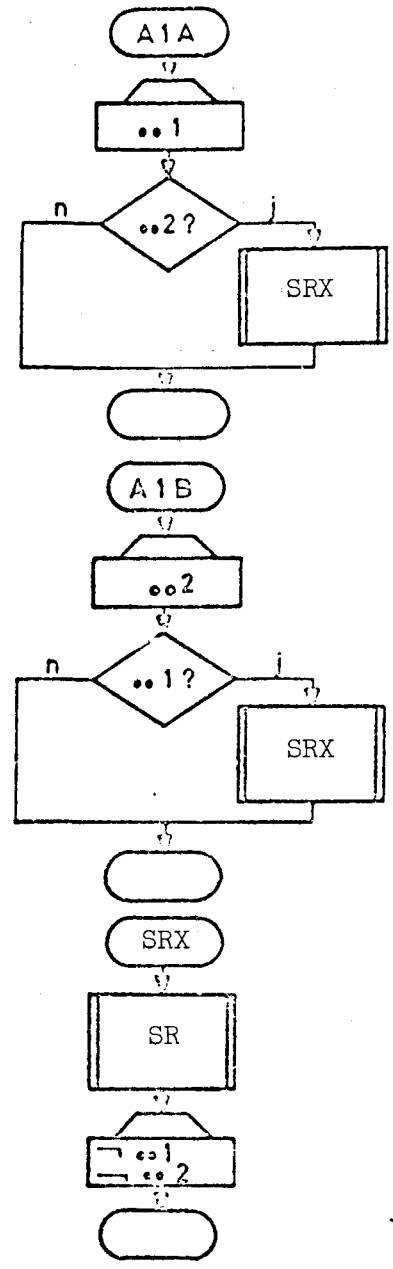
R		
MARK is defined global address		
E SR is external address		
M(ELD) Name in library is MELD; Macro with characteristic letter M		

Fig. 4

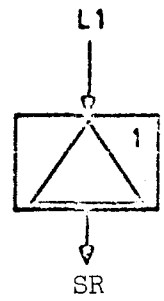
A





Initial condition
.. 1 and .. 2 or
¬ .. 1 and ¬ .. 2
may be chosen

L



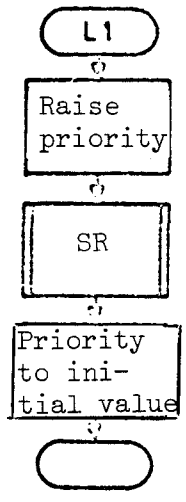


Fig. 5

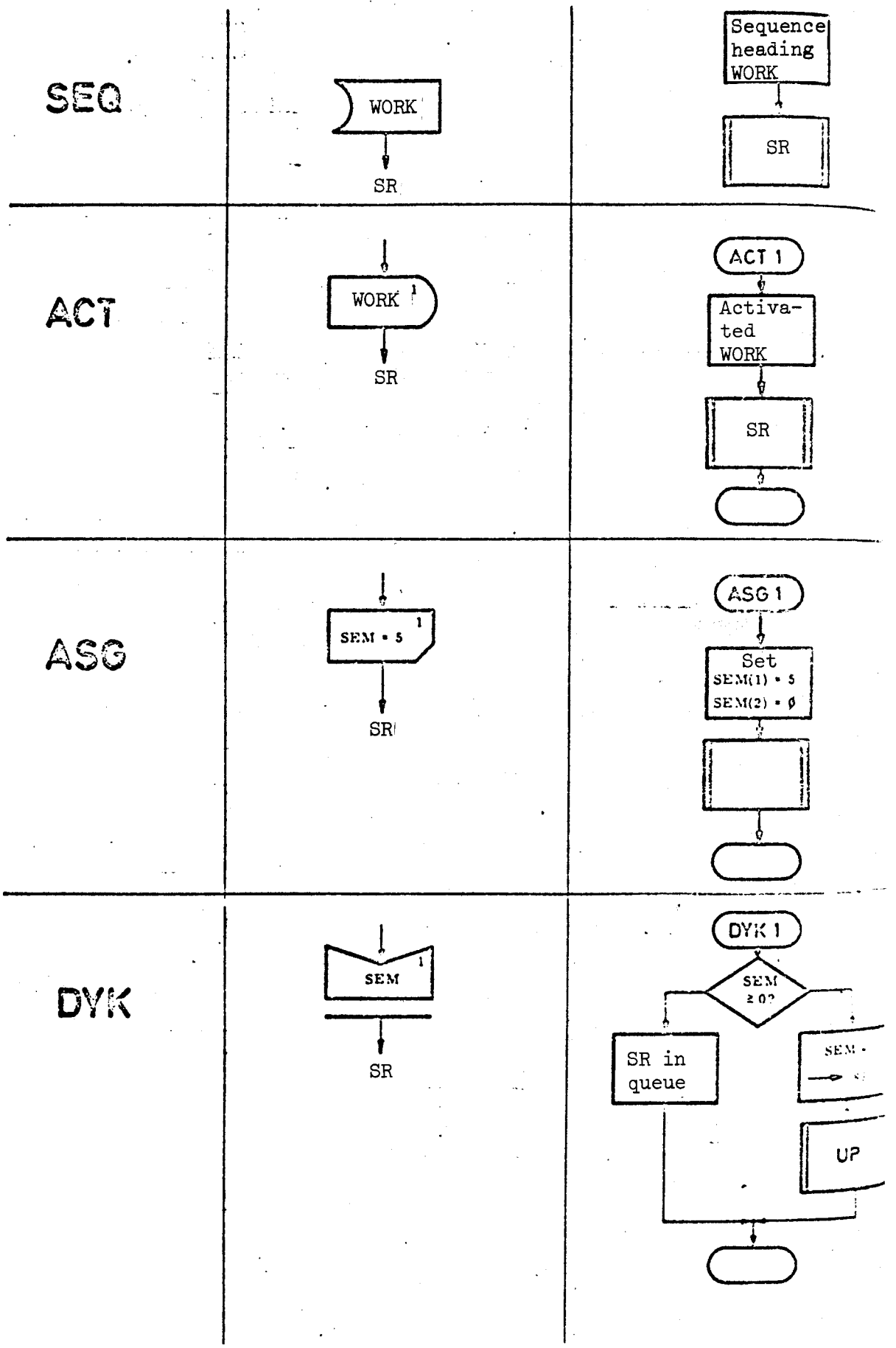

SEQ
WORK
SR
Sequence heading WORK
SR
ACT
WORK 1
SR
ACT 1
Activated WORK
SR
ASG
SEM = 5 1
SR
ASG 1
Set SEM(1) = 5 SEM(2) = 0
DYK
SEM 1
SR
DYK 1
SEM ≥ 0?
SR in queue
SEM -
UP

Fig. 6

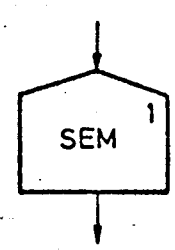
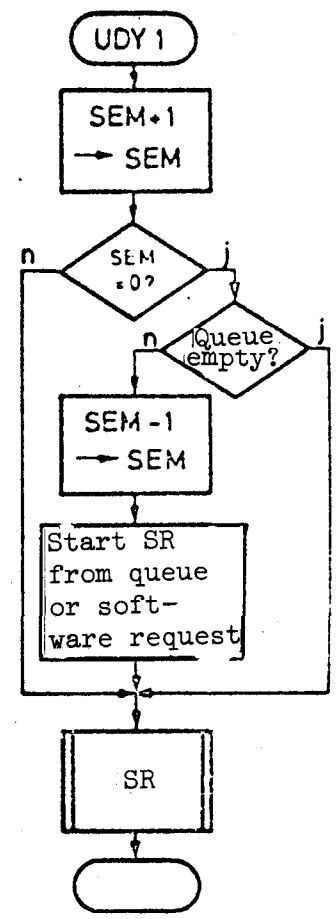
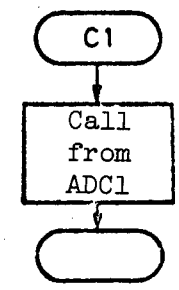
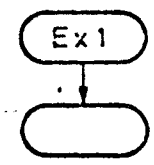
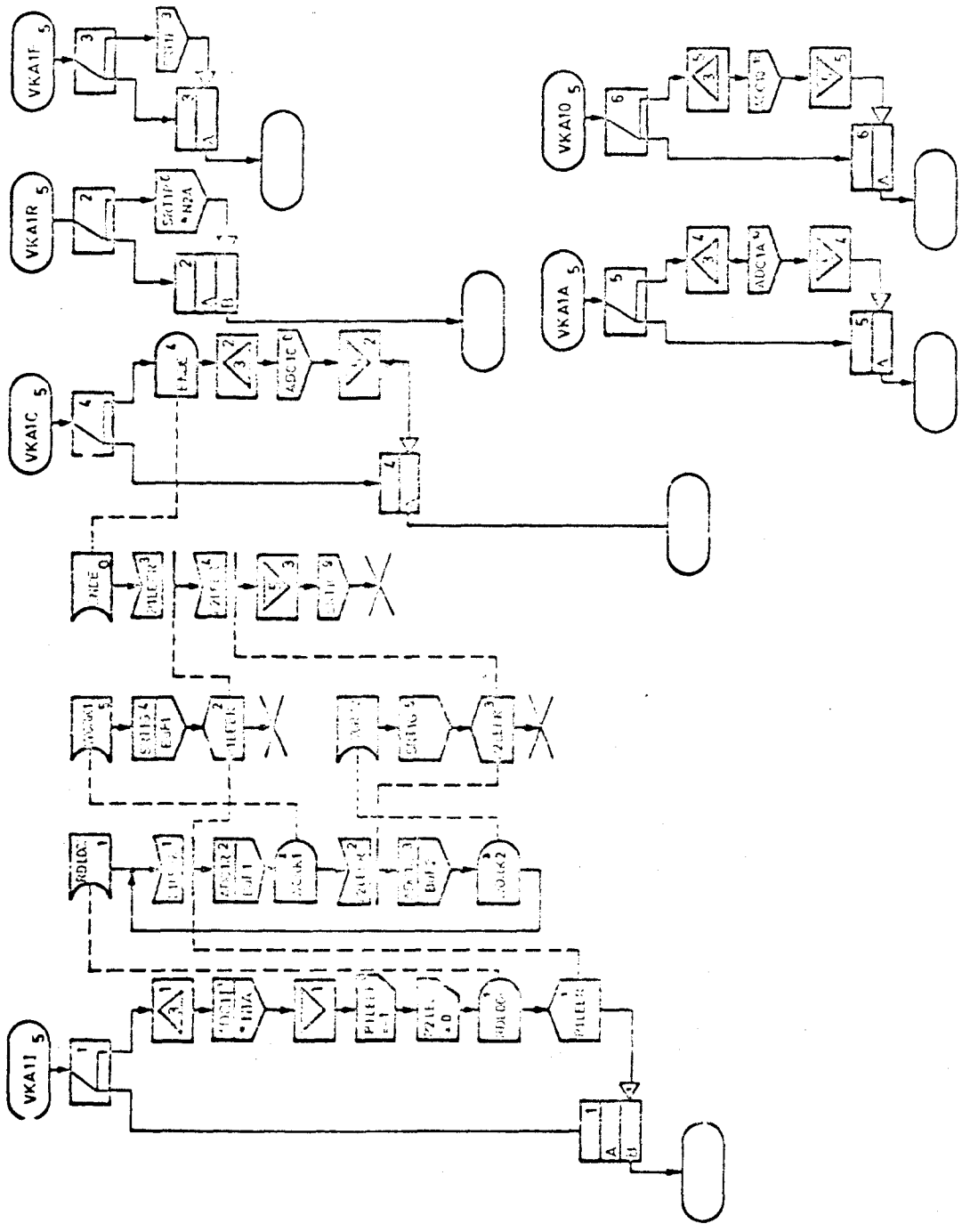
UDY	 SEM 1 SR	 UDY 1 SEM+1 → SEM n SEM =0? j n Queue empty? j SEM-1 → SEM Start SR from queue or soft- ware request SR
C	 ADC1 1	 C1 Call from ADC1
EX	 1	 Ex1

Fig. 7



Key:
LEER = empty
ENDE = end
VKA = MCA

Fig. 8 Multi-channel program MCA 1

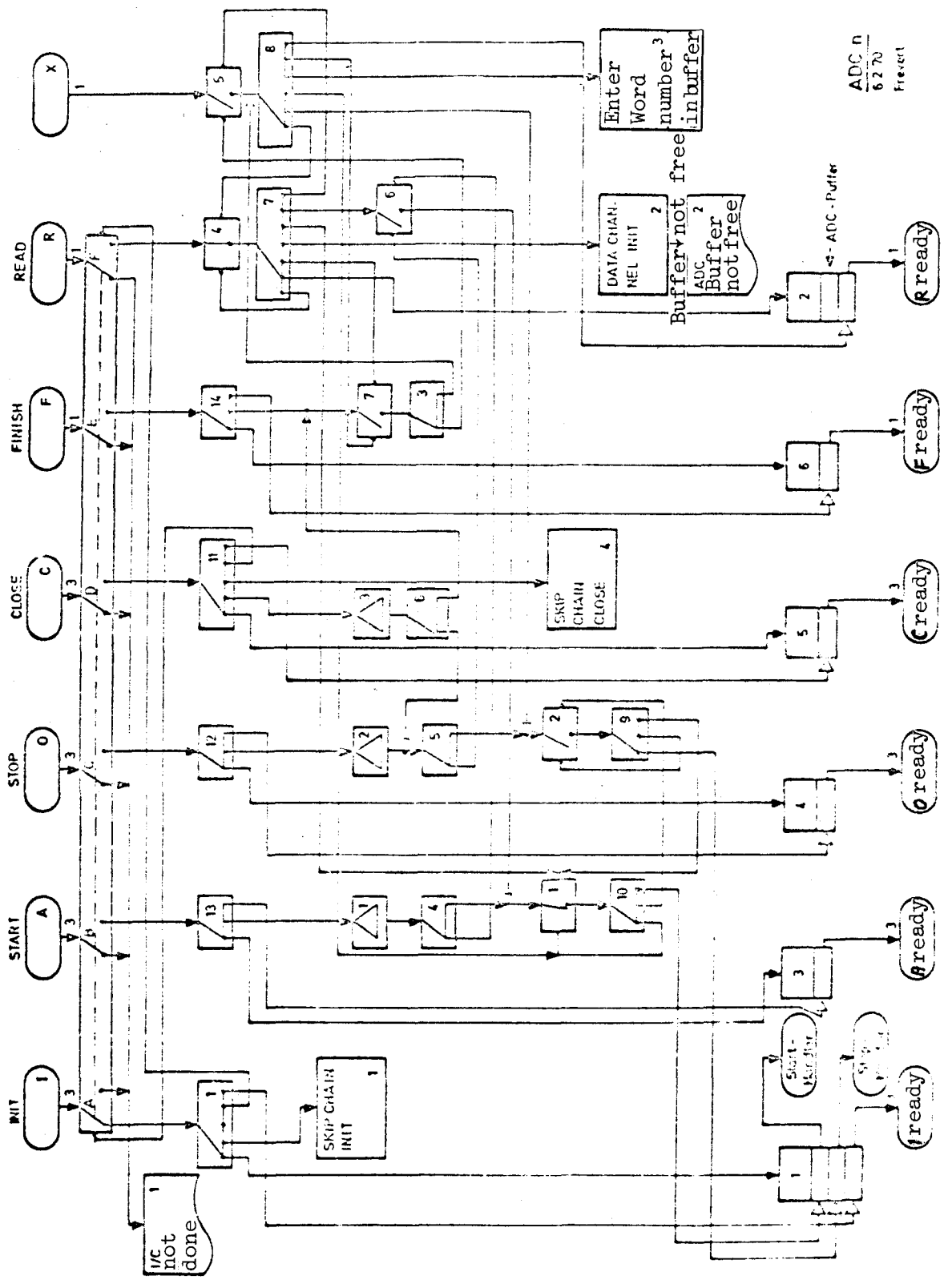

READ
FINISH
CLOSE
STOP
START
INIT
Enter Word number in buffer
DATA CHANNEL INIT
Buffer not free
ADC Buffer not free
ADC n
SKIP CHAIN CLOSE
SKIP CHAIN INIT
I/C not done
Rready
Fready
Cready
Oready
Aready
Iready

Fig. 9

Key:

Datenlieferer 1 fertig = data supplier 1 ready
Nicht gemacht = not done
Leer = empty
Voll = full

Fertig = ready
Füllen = fill
Leeren = empty
Wechselpuffersteuerung = alternating buffer control

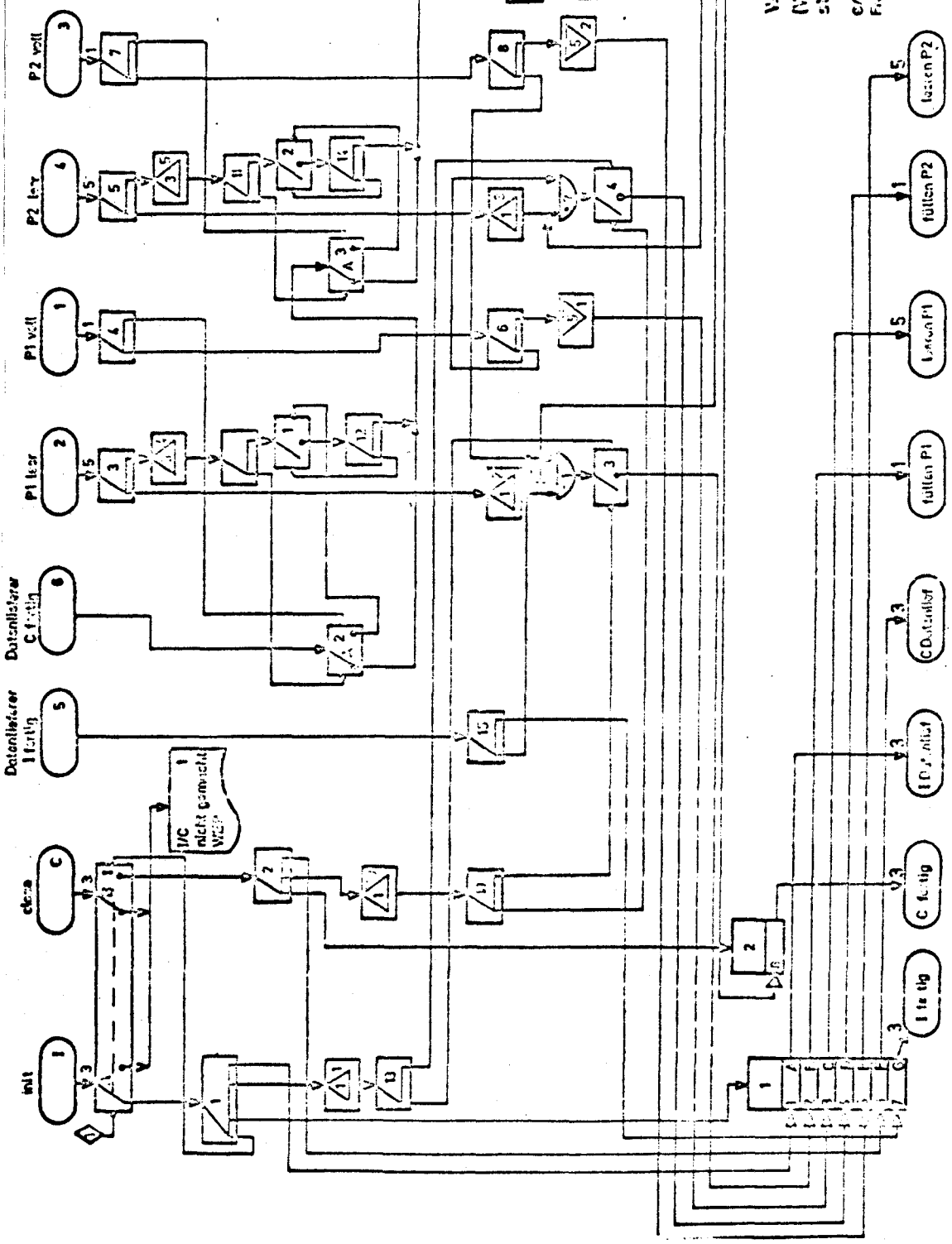


Fig. 10

```
         PROGRAMMNAME =                    ↑EXAMPL↑
↑↑↑↑
         MACRONAME =                       ↑A↑
1001     AND-INTERCONNECTION
         MACRONR.                          ↑1↑
1002     EXIT VECTOR                       ↑M1↑
1003     INITIAL STATE BOTH INPUTS         ↑NE↑
↑↑↑↑
         MACRONAME =                       ↑MELD↑
2001     SIGNAL M
          MACRONR.                         ↑1↑
2002     SIGNAL                            ↑PROGRAM ENDED↑
↑↑↑↑
         MACRONAME =                       ↑DYK↑
3001     DYKE CALL
         MACRONR                           ↑1↑
3002     SEMAVARIABLE                      ↑BUF1↑
3003     INDIREKT?                         ↑NE↑
3004     FURTHER JUMP ADDRESS              ↑A1A↑
↑↑↑↑
         MACRONAME =                       ↑M↑
4001     PROGRAMMMARKE M
         NAME OF LABEL                     ↑EXA11↑
4002     SAME ADDRESS IN PROGRAM           ADR↑A13↑
↑↑↑↑
         MACRONAME =                       ↑M↑
5001     PROGRAMMMARKE M
         NAME DER MARKE                    ↑EXA1A↑
5002     SAME ADDRESS IN PROGRAM           ADR↑DYK1↑
EXIT
```

Fig. 11 Preassembler monitor (same parameters as in fig. 12).

```
PROGRAMMPREASSEMBLER
FAST SWITCH YES OR NO > NO
CORRECT VIA TT > YES
MONITOR OUTPUT > YES
COMMENT LONG > YES
COMMENT VIA TT > YES
EVERY PARAMETER > YES
PARAMETER LIST NAME > PARALI↑

PROGRAMMNAME =>↑EXAMPL↑
GIVE 4 ↑
>↑↑↑↑
MACRONAME =>↑A↑
AND-INTERCONNECTION
MACRONR.>↑1↑
JUMP ADDRESS > ↑M1↑
INITIAL STATE BOTH INPUTS ACTIVATED >↑AN↑

YES OR NO PLEASE
↑AN↑
>↑NE↑

GIB 4 ↑
>↑↑↑↑
MACRONAME =>↑MELD↑
SIGNAL M
 MACRONR.>↑1↑
SIGNAL >↑ PROGRAM ENDED
GIB 4 ↑
>↑↑↑↑
MACRONAME =>↑DYK↑
DYKE CALL
MACRONR>↑1↑
SEMAVARIABLE>↑BUF1↑
INDIREKT?>↑NE↑
FURTHER JUMP ADDRESS >↑A1A↑
GIB 4 ↑
>↑↑↑↑
MACRONAME =>↑M↑
PROGRAMMMARKE M
NAME DER MARKE >↑EXA1I↑
SAME ADDRESS IN PROGRAM; ADDRESS: >↑A1B↑
GIB 4 ↑
>↑↑↑↑
MACRONAME =>↑M↑
PROGRAMMMARKE M
NAME DER MARKE >↑EXA1A↑
GLEICH ADRESSE IM PROGRAMM; ADRESSE: >↑DYK1↑
GIB 4 ↑
>EXIT

CALL NUMBER: 5

>X

OK-KM09 V4E

$
```

Fig. 12 Example of preassembler dialogue input. All symbols to the right of > are supplied by the user.

```
ΔΔΔΔ
ΔΔΔΔ
LOWΔ
LOWERING OF A SEMA VARIABLE
SSR No.:ΔSUP-NRΔ:ΔRΔDZ2Δ
SEMANAME:ΔSEMΔ:ΔSEMΔMC6Δ
INDIREKT:Δ:IND.2:ΔBEDΔJONΔ
ΔΔΔΔ
LOWΔNRΔ 0
YES = 400000
NO  = 0
          ISA+10
          IOF
          LAW -1
          .IFZER 400000-ΔBEDΔ
          TAD* ΔSEMΔ
          .ENDC
          .IFZER 0-ΔBEDΔ
          TAD ΔSEMΔ
          .ENDC
          LAC (400000
          ISA
          ION
          JMP* LOWΔNRΔ
ΔΔΔΔ
ΔΔΔΔ
ΔΔ
AND-INTERCONNECTION
MACRO-NUMBER
NRΔ
NRΔ
DZ2Δ
JUMP ADDRESS
TRANSFERVEC
PΔ
MC6Δ
INITIAL STATE BOTH INPUTS ACTIVATED
INITIAL STATE, INPUTS ACTIVATED
CONΔ
JONΔ
ΔΔΔΔ
          A         ΔΔNRΔA,ΔΔNRΔB,ΔPΔ,,,,ΔCONΔ
ΔΔΔΔ
ΔΔΔΔ
GΔ
TOR
MACRONR.Δ
NRΔ
NRΔ
DZ2Δ
AUSSPRUNGADRESSEΔ
SPRUNGADRESSE Δ
PΔ
MC6Δ
INITIAL CONDITION THROUGHPUT
ΔΔΔΔ
          G         GΔNRΔ,GΔNRΔF,GΔNRΔT,ΔPΔ,,ΔBEDΔ,
```

Fig. 13

```
MACRO-LIBRARY STATE 16.7.70
ÄÄÄÄ
ÄÄÄÄ
MACDEFÄ
ÄÄÄÄ
          .DEFIN S,SM,SN,SF,ST,M,N,BS,2NL,2NM
          .IFZER (SM-SN
SF        0
          DZM       BS
          JMP*      SF
ST        0
          CLA!CMA
          DAC       BS
          JMP*      ST
BS        0
          .ENDC
SN        0
          CLA
          SAD BS
          JMP NL
          JMS N
          JMP NM
NL=.
          JMS M
NM=.
          JMP* SN
          .ENDM
          .DEFIN BUF,BUFNAM,2BUFAD,LENGTH
BUFNAM    BUFAD
          LENGTH
          .DEC
BUFAD     .BLOCK LENGTH;  .OCT
          .ENDM
          .DEFIN SAV,SAVNR,MARK,N,ADR1,ADR2,ADR3,ADR4,ADR5,ADR6,ADR7
          .ETC 2ZEIGER,2RUAD,PARNR,SET
SAVNR     0
          LAC MARK
          TAD (PARNR+1
          DAC INDICATOR
          LAC RUAD
          DAC* INDICATOR
          LAC MARK
          DAC RUAD
          .IFPOZ N-1
          LAC ADR1
          ISZ INDICATOR
          DAC* INDICATOR
          .ENDC
          .IFPOZ N-2
          LAC ADR2
          ISZ INDICATOR
          DAC* INDICATOR
          .ENDC
          .IFPOZ N-3
          LAC ADR3
          ISZ INDICATOR
          DAC* INDICATOR
```

Fig. 14