الجمهورية العربية المتحدة

معهد التخطيط القومي

Memo. No. 569

TIME SHARING AND INCREMENTAL COMPUTATION

May, 1965

LIONELLO A. LOMBARDI

Operations Research Center

القاهرة

Summary

This paper contains

. Proposals for the design of bulk storage devices suitable for multiple-access computers (sections 3.1 and 3.2). The same kind of bulk storage would also 1) allow for implementing effective extensions of memory by means of "page turning" techniques (section 3.3) and 2) on-line, real-time management information systems roughly as inexpensive as batch processing systems (section 5).

. Extensions of the above concept to a general point of view in computer design (sections 2 and 4) which stresses considering interfaces between functional elements as independent functional elements ("buses"). This point of view might be of some use mainly in connection with the design of future computer utilities.

1. Introduction and outline

So far experimental multiple access computers have been built by adapting and patching commercially available pieces of equipment which had been designed with batch processing in mind. Preliminary experience with them is already able to indicate which type of building blocks of computer can be utilized without major changes and which ones need redesign.

This report is aimed at submitting to computer designer and product planners some design patters of some computer elements that they might consider when facing the problem of designing computers primarily suited for a multiple-access usage with real-time response. However, in preparing this report, multiple access has not been the only advance in computer technology that we had in mind. Multiple access aims at giving direct, real-time access to a central computer with a large number of programs to users located at terminals easily accessible to them. But here we consider also the natural step beyong this, namely allowing independent computers to communicate with each other on a real time basis in order to pool their respective power excesses, specialized performance and library programs. This concept of computer network is certainly not new. Its main interest perhaps consists of leading to the development of computer utilities (as opposed to computer centers), analogous to the electric power or telephone utilities. A second interest consist of the fact that multi-location computer networks are amenable to survivability measures against total breakdown which are more difficult to implement in computer centers.

The reason why we are considering such two advances together is that there is a considerable overlap in their peculiar requirements on hardware design. The main element of these overlapping requirements, which contrasts them with the ones of batch-processors, is the need of a variable, easily controllable and potentially very high rate of communications between

- The computer and the user at the terminal
- The computer memory and the bulk storage
- Independent communicating computers

In synthesis, they require special design features in the interfaces between modules, no matter whether those modules are central processing units, terminals, memories or bulk storages.

Let us consider a concrete problem which is hampering the development of our multiple access facility at M. I. T., namely the one of the communications between core memory and bulk storage. In a time-shared computer frequent swaps are required between them. However, available bulk storage devices have a single (or few) interfaces (racks of heads, arms, combs, etc.) which allow to retrieve or bring information from or into bulk storage. This slows down the operation considerably. If the quanta of time allotted to each user are short and the number of users is large, the frequency of required swaps increases to the point of becoming a critical factor. The main problem is not the one of the rate of transmission, but rather the one of the access time for searches. More specifically, searches can be performed only serially by a single interface (in the case of the M. I. T. facility, at a rate of the order of 10 per second at most). What is really

needed is an interface element which is in some way independent of the bulk information support itself, so that further interface elements can be easily added in order to share the heavy load of the searches. This is possible only if the usage envisaged contemplates searches which are originated and executed independently and simultaneously as opposed to being cascaded. This condition is fully met in the type of usage proposed in the examples of sections 3.2, 3.1 and 5.

So sections 3.1 and 3.2 of this paper will be devoted to discussing how bulk storages and their interfaces should be designed in order to meet the above goals. The same kind of storage also will allow to solve the problem of page turning, as discussed in section 3.3. But more important of all, it will allow to overcome the main obstacle to the large scale adoption of on-line real-time management information systems: for this see section 5.

The above outline of a concrete problem and its solution leads to the thinking of interfaces no longer as appendixes to other functional units, but rather as actual, full fledged, independently designed, functional units (modules, if one wants to call them so) of systems. Now, we ask, is it profitable to extend this philosophy from bulk storage interfaces to all interfaces in a system? In most cases we do not at present have a pressing need to do so. But such need might arise when the time of thinking of computer utility networks in concrete terms will come. So we devote section 2 of this report to a rather general and idealized, but unified, preliminary discussion of concepts along these lines. Such discussion is based on the following assumption:

These are two features common to the design of most computer systems which we should consider changing if a leap forward is to be made. The first, or chartered interface structure, is the reliance on a predetermined, frozen system of communications, or interfaces, between operational elements, which can only accommodate functional elements up to a certain prescribed number and in certain prescribed configuration types: this allows for the availability of reduced and inexpensive versions of potentially large system (i.e., it allows to start small), but does not provide a good vehicle to grow either big or along unpredicted patterns. The second, or hierarchization, is the necessity of some kind of central supervisory control (hardware and/or software) which assigns and monitors the tasks of the different elements: besides making survivability of the system contingent upon the constant flawlessness of the supervisor, such features limits the growth of the system to the size or configurations that the supervisor can handle. Both such features are a left-over of the time when designers were focusing on the batch processing performance of accentrated (packaged) computers, aimed at batch processing rather than on computers which should primarily interact on a real-time basis with their (human or automatized) environment.

The concept that we propose as replacement of the first feature is the one of independent interface, here called bus. This discussion is reinforced by examples of design of accesses to bulk storages (section 3.1) and terminal links (section 4). However, the second example is substantially less specific and concrete than the first, due to the present lack of an appropriate basis for experimentation.

The proposed replacement to the second feature of batch processors (hierarchization) consists of providing for all coordination decisions, such as those pertaining to priority or storage allocation, to be made locally, with reference limited to the part of the system which is directl contiguous to the particular point where the decision is made. This is n to say that central supervision is always a bad feature. Simply, supervi is a matter of installation policy and not one of system design. So sup visory features, when desired, should be built as an overstructure of systems as opposed to being intimately, and hence rigidly, imbedded in their basic structure.

2. Three types of modules

Here we will present a rather idealized way of looking at the modules of an information system. The purpose is to provide a frame of references for the following sections.

We will consider three types of modules as being parts of a system, namely buses, transducers and memory areas.

2.1 Buses

The purpose of a bus is to transfer information between one set of modules to another set of modules.

Examples of buses in conventional computers are: Buffer registers for core memory input-output. Buffers of input-output units. Staticizers or buffers of input-output terminals. Logical (one-message-at-a-time) channels in multiplexed transmission systems. Arms, combs, traps, racks of read-write heads or junctions for input-output to drum, disk, magnetic card, tape drum (e.g., FACIT Carousel or IBM 2321 Data Cell) or delay line storage.

Each bus may include

i) A bit which in on if and only if the bus is busy (busy bit).

ii) A register (source register) devoted to carrying the address of the unit (and maybe the address within such unit) from which the information to be transmitted comes.

iii) A register (destination register) devoted to carrying the address of the unit (and maybe the address within such unit) to which the information to be transmitted should go.

iv) A register (continuation register) to carry the address or indication of the address from which to resume execution of the

program module which has been interrupted in order to execute the transmission. (in some cases, namely when either of the units between which information is transmitted is a memory area, this item can be replaced by a string of bits of such areas, which is addressable through i) or ii) above).

v. A signaling feature (interruptor) which triggers the resumption of execution of the interrupted program.

vi. The equipment to actually carry out the transmission.

Quantitative parameters which measure the performance of a bus are

i) Its source range, or size (in bits) of the part of the system (contiguous modules) from which information can be transmitted through it. Sources which are storage areas should be measured by the sum of their capacities in bits. Sources of other kinds (transducers or other buses, as well as memory areas which are considered independently of their capacity) should account for one bit each.

ii) Its destination range, analogous to the above, but in the opposite direction.

iii) Its bit rate (measured in number of bits per second) which a bus transmits.

iv) Its message rate (measured in number of messages, each of one bit, per second) which a bus can transmit. The message rate, which indicates how many accesses per second a bus can execute, is much more significant than the bit rate for buses such as bulk random access storage accessing tools. The message rate is a stochastic variable which depends on the source and destination of the messages: it should be noted by its mean value, and perhaps variance (the code and parameters of a probability

distribution may be informative in some cases).

A particular kind of bus is a priority scheduler, which assigns available modules (of any of the three types) to a program module which request some. The source range of such a bus is the number of modules which can request its services. Its destination range is the number of modules that it can interrogate for availability. Its bit rate is uninformative since such bus transmits only the address of a module per message. Its message rate is the number of assignments that it can execute per second.

The design of priority schedulers can vary extensively. The field is quite well known due to half a century of research by engineers and mathematicians analyzing telephone networks. What is common to all priority schedulers is that they should assign a module of a particular category whose busy signal is off to one of the modules which requested it. So a priority scheduler should be able to

i) Evaluate whether a module belongs to a given category

ii) Sense its busy bit

iii) Assign modules and initiate operation.

There should be a criterion for the assignment of modules, which should take into account both the priority of the requestor and the variable load on the requested modules (e.g., a multiplier should be assigned to a program module asking for the execution of a sum only conditional upon all adders being busy and the requester having good priority level). All we can say in general is that design simplicity rather than priority optimality should be the criterion in designing priority schedulers.

2.2 Transducers

The purpose of a transducer is to transform information upon request. If such transformation depends only on the information which is being supplied the transducer is usually called combinatorial. If the transformation depends only on the information which is being supplied and on the contents of internal memories of the computer the transducer is called sequential. In general, such transformation may also depend on exogenous variables such as real time gauges human intervention initiated independently or upon request of the transducer, load to which the transducer is subject, or input of digital information from other sources.

Examples of transducers are: Adders, multipliers, dividers, etc. Complete arithmetic units. Decoders and transcoders. Control units. Input-output terminals. On-line terminals with human operators. Complete autonomous computers.

Exactly like a bus, a transducer may include a busy bit, a source register, a destination register, a continuation register, an interruptor, and the equipment to carry out the transformation. Again not all of these parts need always to be present (in some cases some may be borrowed from contiguous buses).

Among the quantitative parameters which measure the performance of a transducer we can again list the source range and destination range, defined as the number of (contiguous) buses (i.e., buses which can reach it and bring information to or retrieve information from it); and also the input message rate, input bit rate, output message rate and output bit rate, all obviously defined (not all of the latter four parameters are very informative for some types of transducers).

We already see a considerable overlap between buses and transducers. In fact the distinction is more a question of emphasis than a dicotomy. One can easily discuss buses as particular transducers. We have chosen to do otherwise solely in order to stress the modularity of the system of interfaces, which is composed of buses.

Other quantitative parameters can only be given for particular transducers. For example, for an adder we can consider a performance parameter such as its speed, or $n*k$, where k is the number of couples of numbers, of which the largest has n bits, that it can add in a second (then the speed is significant only to the extent to which $n*k$ is independent of n). For a multiplier, we can define as its speed the number $m*n*k$, where k is the number of multiplications of a number of m bits times one of n bits that it can execute in a second (significant only to the extent to which $m*n*k$ is independent of m and n).

In general it is not reasonable to try to assign aggregate quantitative measurements of performance to complex, diversified transducers: this is possible only for simple ones. However, from the standpoint of this discussion we may take one kind of aggregate measurement of performance, namely one referring to transducers which perform analogous function (i.e., they are analogous). More precisely, we can say that if all transducers a_i of a set A of analogous transducers have the same source and the same destination (i.e., the sets of their contiguous buses coincide), then, if $P(a_i)$ is any performance parameter of a_i,

$$P(A) = {}_iP(a_i) \qquad (2.2.1)$$

The above sentence (a useful general consideration and attitude, rather

than a theorem) means that for most intents and purposes the performance parameters of sets of analogous buses which can always replace each other by having identical means of access can be cumulated. In many cases, this means that they can cumulate without interface overheads their capacity of carrying loads, in the sense that, for example, if a adequate for the needs of a program module z, then three transducers identical to a with sources and destination common to a are adequate for the needs of three independent time-shared program modules identical to z. (Of course, we refer here only to performance of the transducers, not of their interfaces: such time-sharing would require threefold message and bit rate, and thus increasing the number of buses. The point here is to disentangle bottlenecks created by the performance of transducers from the ones generated by interfaces, which should be considered separately).

It should be stressed that such cumulation affects external performance only conditional upon the clause concerning the common sources and destination being satisfied: for trivial that this point might appear, the fact of having systematically disregarded it at design stage is one of the main reasons for the unbalance and rigidity of some information systems.

2.3 Memory areas

The purposes of memory areas are to store modules of operating program, temporary data on a stand-by basis, and extensive data bases.

Examples of memory areas are segments of core, microfilm or any kind of read-only memory, tracks or sectors of tracks in disks or drums, bites of delay lines, physical blocks of magnetic tape, tape strips on Data Cells, or magnetic cards.

Again, memory areas should in some cases be provided with a busy bit and source and destination registers. Continuation registers and interruptors are needed only in particular cases, namely when memory areas are to play a control role.

In addition, a memory register should be provided with a program register which, when the memory area is busy, carries the identifier of the program module or user which is using it: program registers are useful mainly for memory protection. They can be shared by close sets of memory areas in some cases.

Significant parameters of a memory area are its capacity in bits, its source and destination ranges (i.e., number of contiguous buses) and the bit rate at which information can be written into or read from it.

The assignment of memory areas to modules of programs should be handled by memory assigners. A memory assigner should keep a standing list of available (free) memory areas and a list of the program modules to which busy memory areas belong. It should assign them much the same way as a priority scheduler assigns transducers, switch requests by converting relative addresses, and protect from attempts of accessing memory areas by foreign program modules by mistake.

The important point here is that memory assigners are themselves separate modules, rather than appendixes of memory areas. It is convenient to consider them as buses, because this allows unified discussion of the operational range in terms of the above defined source and destination range of buses and memory areas (this is the reason why we also like to view priority schedulers as buses, while others might prefer to consider them as transducers, due to the fact that they actually transform information).

When we measure an entity, we generally measure bottlenecks, averages, and interfaces. In the case of an information system, the measurement of the work of buses can be informative in determining the picture of the data flow. So we might want to have meters, or modules which measure the work of buses and keep appropriate accounting. To be consistent with the above distinctions, such meters should be viewed as memory areas, although they don't look that way. (A meter can also be viewed designed as a small memory area where cumulated data are stored, which communicates through ad hoc buses with an ad hoc gauging transducer).

In the case of an information utility, meters used for billing and reporting should be placed both contiguous to the buses which transmit information between the customer and the facility used and contiguous to the priority schedulers and memory assigners within the facility.

2.4 Survivability and supe sion

Survivability of an information system should be based on the fact that priority schedulers and memory assigners should keep standing information (either exogenous or originated by internal automatic verification) of all modules falling within their scope which are temporarily or permanently out of order or permanently assigned to high priority functions. So survivability can be graduated in terms of partial overlap between the scope of priority schedulers and memory assigners.

Needless to say, also the assignment of priority schedulers and memory assigners to program modules should be handled by autonomous modules whenever there is overlap.

Survivability is one of the reasons (not the only one) for considering designing complex information systems which can work without a central supervisor. This is not to say they cannot accommodate a supervisor, but rather that the desirability and features of a supervisor is a matter of policy and not one of system design, so that decisions on this issue should be made by the computer user, not by the supplier. The latter should only be able to allow any system to accept supervision, if so desired.

> Supervision should be based on collection of information from the periphery, making decisions centrally and dispatching them to the periphery. Information for decisions should be collected by meters as specified above and should be routed to wherever it is needed by elements of the network (buses,

chains of buses, etc.). Supervisory decisions, like all decisions, are made by parts of the network, which differ from the remainder of the netwoek not primarily for the way they are designed, but rather for the particular way they are connected. Then, decisions based on such information should be routed to the periphery in the same way. In many cases it seems that a good way of executing decisions is by controlling the behaviour of memory assigners and priority schedules (although we have no experience at this stage to substantiate this statement).

3. Problems of stand-by information

We will discuss now how a random access (disk, drum, delay line or systems of magnetic cards or tape strips) storage able to meet the requirements of multiple access computers should be designed and how two well identified problems which hamper the development of currently working system can be easily solved.

3.1 Stand-by storage

A disk, drum, delay line, magnetic card or Data Cell Drive is a system of memory areas. The arms, combs, junctions, traps or heads which transfer information to and from it are buses. So, in principle, it is of some importance not to tie a priori these two different kinds of modules together.

If the stand-by storage relies on mechanically moving arms, traps, combs, or racks of heads, then each of such arms or combs, along with the logical items discussed in 2.1, is a separate bus, and their assignment should be handled by priority schedulers. The disk stack, drum, tape strip or card system should be designed to accommodate a variable and potentially very large number of arms, combs, racks or traps.

More promising is the case of stand-by storages without moving parts, where each track is under at least one steady reading or read-write unit. (Incidentally, this is always the case for delay line storage). Then such units should be viewed as parts of the memory areas, while the access buses are strictly electronic. More precisely, buses consist only of their logical parts (busy bit, source, destination and [illegible] registers, interruptor and transmitting line). The message

rate can be regulated by the spinning speed of the disks or drum or by the frequency of the delay line, by the number of reading units per track or line or by the number of buses. The advantage of solutions without moving parts are low cost per access bus and wide boundaries within which the number and type of buses can be chosen and expanded.

Such a stand-by storage requires asyncrouous time-shared operation among program modules (belonging to the same or different programs). Coordination should be handled by providing program modules with the capacity of effectuating some elementary operations that we will see exemplified in section 4 and 5. Such operations can be simple and decentralized under the assumption that hardware modules are provided with the prescribed logical items (busy bits, source, destination and continuation registers, interruptors). So effective random access storage should be designed by keeping well in mind at least this aspect of the functional relationship between hardware and software.

Some commercially available stand-by storages (disks or drums especially) have multiple arms or combs which are organized in a way that they always look simultaneously for the same memory area. Such a set of arms or combs constitutes a unique bus. Such systems have been developed for a purpose different from the one that we have in in mind, namely the one of maximizing access speed. However, in general, their message rate is rigid, so that they do not provide a good basis for multi-access or multi-computer systems. In fact, the message rate in this kind of stand-by memories can be increased only by multiplying the number of complete disk or drum storages

units. But this may involve scattering data in a memory space wider than necessary. In addition, the limited range of each bus with respect to the total memory available (there is no overlap of scope what so ever) causes bottlenecks whenever two memory areas on the same unit need to be searched. On top of that, software supports for such approaches are quite involved and generally not provided (indeed they cannot even be designed efficiently unless provision for them has been made at the level of hardware planning). So this short cut, which, incidentally, is also constrasts with the efficient usage or removable disk packs, is to be discarded.

3.2 Bulk storage for multiple-access, time-shared computers.

Experience with multiple access, time shared computers shows that the main bottleneck in such systems is the swapping of programs in and out of core memory. In a typical stand by memory system (drum or disk) designed to accommodate only one accessing bus, the following typical situation arises: when an execution quantum of the program module A_1 has been performed, the accessing bus gets charged with the function of moving A_1 to stand-by storage and replace it with another program module A_3, which is in stand-by storage. In the meantime, the computer proceeds to execute a quantum of another program A_2. The execution of the latter quantum is generally over much before the above swap is accomplished. At this time A_2 should be swapped out and replaced, which is impossible because the unique bus is busy: so the computer must wait. If the number programs in simultaneous execution is large (e.g., 100 or so) then, even if there are two or four stand-by storage units each with one bus, the computer is idle most of the time. The criticality of this bottleneck can be reduced only by increasing the length of the execution quanta (thus decreasing the immediacy of man-computer interaction and barring out some possibilities of real time control) or by increasing the core memory size in order to keep many programs in core at the same time (which is most expensive) or by severly limiting (e.g., to a small multiple of ten at most in the case of an IBM 7094) the number of simultaneous programs, which contradicts the very purpose of multiple access computation facilities.

On the contrary, by using a stand-by storage as described in 3.1, the problem would be easily solved simply by installing a number of

inexpensive access buses such that the sum of their message rate and bit rate match the anticipated swap frequency and the related memory input-output, respectively.

3.3 Page turning

The problem of 3.2 is formally similar and functionally identical to the one of one-level storages, (towards whose solution design of the Ferranti ATIAS computer is a step forward but not yet a satisfactory answer). Upon simplification, this new problem can be described as follows: the computer has only limited (expensive) core or film storage space, which, however, is largely extended by drums or disks. The problem is to have the hardware-supervisor system designed in a way to allow both programmer and compiler to think in terms of having a homogeneous (one level) core storage of the (large) size of the disks or drums, and letting the supervisor take care, at execution time, of always placing into the real (small) core storage the particular section (page) of the program which is being instantaneously utilized, and removing it after utilization (page turning).

Again, if one uses a conventional stand-by storage limited to n buses (n small, and $n=1$ in case of the simplest version of ATIAS!) these will be immediately tied up for swaps relative to page A_1, A_2 ..., A_n, respectively. At the end of the execution of page A_{n+1} the computer must wait until a bus is free. Unlike the problem of 3.2, this one does not allow a shortcut patch solution such as lengthening the execution quanta, because the execution time for a page is generally outside the control of either the programmer or the computer, and is in most cases by many orders of magnitude shorter than the time required by a bulk storage access.

Here again there would be no problem if the stand-by storage were designed independently of its interfaces as outlined in section 3.1, because in this case the message rate could be inexpensively matched to the anticipated page turning frequency.

3.4 Parametrization of random access storage.

There follows now a preliminary study for the parametrization and measure of data processing capacity of random access data systems designed as outlined in section 3.1.

Given a drum or disk storage drive $\underline{i}$ of capacity V(i) bits, let A(i) be the sum of the average access time plus the time necessary for transmitting an average length record, i.e.,

$$A(i) ::= \frac{1}{\text{message rate}} \quad \frac{\text{mean number of bits per record}}{\text{bit rate}}$$

$A(\underline{i})$ is important in artificial intelligence where decisions are logically cascaded (serial), but by and large of lesser relevance in general data processing.

Let $N(\underline{i})$ be the maximum number of independent accessing buses which can be installed, and let $C(\underline{i})$ be the cost of each of them. Then the measure of the top performance (or power) of the storage system is

$$P(\underline{i}) ::= \frac{N(\underline{i})}{A(\underline{i})}$$

while the measure of its dollar payoff is

$$R(\underline{i}) ::= \frac{1}{A(\underline{i}) \cdot C(\underline{i})}$$

These are the important factors, not only $A(\underline{i})$.

One must be careful in considering the case of a system with a set $\underline{i}$ of several disk storage drives, whose total capacity is

$$V(\underline{i}) = (\underline{i})$$

One should not a priori believe that

$$P(\underline{i}) = P(\underline{i}) \qquad (3.4.1)$$

Formula (3.4.1) is approximately correct only for applications where the principal file to be searched has a volume much greater than V(i). In any other case, to make it true, that is, to make all buses of all drives work in parallel, one should scatter the records of the file among all the i forming $\bigcup i$, which has the drawbacks discussed in section 3.1. For small files a better formula is

$$P(\bigcup i) = P(i) \qquad (3.4.2)$$

In conclusion, random access devices suitable for time-sharing and random access processes in general should have the following features:

a) Removable disk packs or drums.

b) Allowing for an arbitrary number of independently operating accessing buses.

c) High values of R(i) for a wide range of P(i).

4. Terminal links for multiple access computers

In order to make a multiple access computer serve effectively as automated extension of the human intellect for creative work or decision making in research, education, engineering designing or management, the computer should be made easily, continuously and independently accessible to a large and unboundedly expandible number of people without decreasing its performance/cost ratio. Some terminals may be simple, inexpensive typewriters; others advanced real-time handwriting recognizers. To be able to distribute terminals adequately, one must be willing to pay the price consisting of the fact that most of them will be used infrequently and on a rather unpredictable time schedule. The only way to make this price approachable on a large scale production basis is to untie as far as possible from the terminals all functional units which can conceivably be shared by several of them, namely their buffers and links to the computer network.

A terminal consists of two buses: an input bus (keyboard, A/D converter, light pencil, pencil tracker, etc.) and an output bus (typing part of a typewriter, D/A converter, CRT tube, incremental plotter, etc.). These are the only modules that should be permanently tied to a terminal station, and thus the only ones for which scant usage is to be contemplated.

The typical operation is as follows: when an input bus $\underline{i}_1$ is activated, a priority scheduler P_1, contiguous to $\underline{i}_1$ as well as to other input buses, should assign to $\underline{i}_1$ an available link bus $\underline{i}_1$ out of the set $\underline{L}_1$ of all link buses contiguous to $\underline{i}_1$. (The bus $\underline{i}_1$ will typically be a buffer register). Though its interruptor, $\underline{i}_1$ will request the assignment by an available priority scheduler of a transducer to perform the next step of processing.

For example, if $\underline{i}_1$ is a keyboard, $\underline{\underline{i}}_1$ will carry an addressed string consisting of the quantum of information just keyed in, together with the identifier of the terminal to which $\underline{i}_1$ belongs; then the requested transducer will be a decoder or control unit (out of a contiguous set) which will determine what to do with the addressed string (which may carry a signal, an instruction, or data), perhaps asking other transducers for help by means of a priority scheduler or appropriate software bus (supervisor). The next steps might be typically a request to a priority schedulers for buses and requests to another priority scheduler for an arithmetic transducer or/and to a memory assigner for memory areas.

In case $\underline{i}_1$ is a pencil tracker, $\underline{\underline{i}}_1$ will carry a 1-bit signal, and it will ask an available priority scheduler to assign to it an available transducer (specifically, an A/D converter or special purpose computer) out of a contiguous also to $\underline{i}_1$, able to transmit the tracking signals. Then the decoded tracking signals will be further processed as in the previous case, where $\underline{i}_1$ was a keyboard.

Similarly for the output bus $\underline{o}_1$: to perform output, if $\underline{o}_1$ requires buffering (as in the case of visual displays), the neighboring part of the network will summon through a memory assigner a memory area (buffer) out of a set. In any case there will be a request for a link bus $\underline{\underline{o}}_1$ contiguous to $\underline{o}_1$, which in the case of buffered output will transmit a single signal to promote the request by $\underline{o}_1$ of a bus contiguous to both the buffer and $\underline{o}_1$ (alternatively, such request can be issued by the buffer). In case of non-buffered output, $\underline{\underline{o}}_1$ will be used to directly transmit an increment of output to $\underline{o}_1$.

Besides allowing the maximum utilization of expensive terminal

processing and storage equipment possible under the circumstances, the main advantages of this approach is that it does not require any kind of general, operationally expensive supervision which would limit expansion and freeze the design pattern against unanticipated evolution. Instead, with this approach all decisions are made locally (at interface level).

However, not having a supervisor, increments of information flowing through the modules of the network are in a sense loose. If there were a supervisor, this would always keep track of what happens where. Not having one, it is necessary to associate to increments of information identifying references to their nature (i.e., instruction, signal, datum), their origin and/or destination. Thus, by and large, information should circulate in the form of addressed strings above introduced by an example. The basic concept of addressed string can be used and extended in many ways to suit specific needs, so it would inappropriate to establish guidelines at this stage due to the lack of a concrete testing ground. We will just mention the fact that some transducers should be enabled to perform the task of changing, augmenting or abolishing parts of an addressed string other than the quantitative contents (i.e., the part of it which would be the only necessary in a supervised, hierarchized computer netwoek). They should also operate on the addresses or other parts: thus, routing decision and message switching can be made by transducers, perhaps with the aid of other contiguous modules.

> It would be a mistake to view the addressed strings as redundant packages of information, and to think that their use is a price that one should pay in order to dispose of supervised computers. It looks like, but it is not true, that a supervised computer can proceed with the bare quantitative contents of the information increments. In fact

at any instant of the operation of a supervised computer, the information other than the quantitative contents (routing, descriptive information, etc.) relative to all information increments present anywhere in the computer, is actually available to the supervisor. So the difference between a supervised computer and the one that we are investigating is that the former disassociates, but does not reduce the size of, addressable strings, and often keeps all information but the quantitative contents in awkward places and disorganized configuration, thus requiring and tying up expensive hardware and slow software. Instead, the point of view taken here contemplates storing all information which is needed in a uniform, systematic and inexpensive fashion by using the addressed string configuration. In a system designed from this standpoint, expensive pieces of hardware devoted to processing information other than quantitative contents are summoned (on a time sharing basis) only when and for the exact length of time that they are needed. So, against appearances, addressed strings are a competitive advantage, rather than a disadvantage, of this new point of view vs. the one of supervised systems.

Besides the quantitative contents, actual, relative or symbolic addressers, and descriptive information, often addressed strings should carry the identifier of the program module or user from which they have been directly or indirectly originated. This feature allows for a handy way of allowing memory schedulers to provide for effective protection of busy memory areas from mistaking invasion attempts by unrelated, simultaneous program modules. They could do this by

comparing, before erasing information, such identifier of the addressed string which is requesting erasure with the contents of the program register (see 2.3) of the memory area affected.

5. On-line Real-time Management Information Systems

During the last three years or so it has become frequent in the trade literature to predict the advent and discuss the advantages of on-line, real-time information systems. Such quite evident advantages to management would be in terms of selective supply of information for decision directly upon request, instantaneous evaluation of the implications of proposed policies, immediate execution of decisions, reactive reporting on the consequences of such decisions and maintainance of the data base continuously posted with new transactions. Their advent was predicted exclusively on the basis of the desirability of such advantages and on the expansion of computer usage that they imply.

However there is an obstacle to the actual taking place of their advent on a large scale basis. Let us discuss such obstacles.

In such literature writers were maintaining that on-line real-time system with a reaction time of the order of seconds (as opposed to hours or days of batch processing systems) are inherently expensive. And they could easily document such statements on the basis of available experimental systems.

But let us analyze why they are expensive. Typically, the arrival of an exogenous message (transaction or request of information) into an on-line real-time system causes a hierarchized and often fanning out chain of request for records of the data base. The data base is typically on disks or drums, (in the future, perhaps, on delay lines). If such disks or drums have only one accessing bus each, retrieval of such records has to be serial, Using today's commercial equipment, the retrieval of data base records for a single transaction requires typically a time of the order of one second

or more, while the related internal processing, which is carried out at electronic speed, generally takes no more than a few milliseconds, even in case that it is quite subtle. So most of the time the computer is idle waiting for the drum or disk access buses to retrieve information. There is clearly no point, while waiting, in entering a new transaction, because this one would do nothing but add to the current load of the accessing bus its own requests of data base records. On the other hand, adding power to the computer will increase the cost without enhancing performance: the bottleneck is not in the internal power, but in the message rate of the access buses of drums or disks. Increasing the number of disk or drum units would be a poor approach, as discussed in section 3. (This last remedy has been compared to the following one: Assume that Boston has too few taxicabs, so that people are delayed when they want to go to the airport: then, as a remedy, let us build a second airport).

Besides cost considerations, this situation also imposes a limit of performance of the order of one exogenous message per second to on-line real-time systems. This implies that a common, valuable, integrated data base cannot be used too actively for supporting decisions made at different points and different level of an organization.

It is clear from the preceeding discussion that, however, high cost and low performance ceiling are not inherent in on-line, real-time systems, but rather depend on the fact that the random access data base supports currently available on the market, which have been designed primarily with batch processing in mind, turn out to be not suitable for on line real time usage. More precisely, manufacturers today supply packages consisting of a disk or drum with one or a small number of access buses. Instead, what is

needed is independence between the disk or drum and its access buses. This relatively simple system advance is perfectly within reach of present engineering design. Its implementation would probably make on-line, real-time management systems available at the cost and power of conventional batch processing systems.

From the standpoint of the user's system designer, the data base support should be measured as follows: The total capacity in bits of the aggregate A of all drums or disks should match the maximum predicted size of the data base. Independently from this, the interface should be sized as follows: let f be the peak frequency of exogenous messages (transactions or requests for information) that the system should cope with. Let n be the mean number of accesses to the data base implied by a transaction and m the message rate of each accessing bus (supposed two-way for simplicity). Then the total number M of necessary buses is

$$M = \frac{f*n}{m} \qquad (5.1)$$

Notice that M does not depend on the size of the data base. In designing the interface, one should try to minimize the cost for given M*m by choosing m appropriately.

Formula (5.1) is true if and only if every accessing bus can reach all of A. If the range of each accessing bus is comparatively large and there is extensive overlap of range between buses, then (5.1) is still approximately correct for all intents and purposes of design.

A problem arises when A is divided into sections for which the probability of an exogenous message causing an access varies. In this case buses must be allocated in a way such that their ranges overlap more densely where the probability of access is higher. More precisely, let p(a) be the

probability of a transaction requesting an access to the memory area $\underline{a}$, and $\underline{p}(\underline{a})$ the mean of $\underline{p}(\underline{a})$ over A. Let further $\Phi_1(\underline{a})$ and $\Phi_2(\underline{a})$ be the range of $\underline{a}$ (i.e., the number of access buses contiguous to $\underline{a}$), and $\overline{\Phi}(\underline{a})$ its mean over A. Then an ideal distribution of accessing buses would have to satisfy

$$\frac{\Phi_1(\underline{a})}{\overline{\Phi}(\underline{a})} = \frac{\underline{p}(\underline{a})}{\overline{p}(\underline{a})} \qquad (5.2)$$

for every $\underline{a}$ in A. Clearly, due to the inherently discrete nature of modular system s in concrete systems (5.2) can only be approximated.

Let us briefly describe the typical operation of an on-line, real-time management information system. Each incoming exogenous message $\underline{r}_1$ is assigned an internal (e.g., core) memory area $\underline{a}_1$ by a memory assigner. Then it is processed until it is determined which $\underline{k}_{r1}$ records of the data base are relevant to its further processing. At this point $\underline{k}_{r_1}$ available accessing buses, $\underline{b}_1^{r1}, \underline{b}_2^{r1}, \ldots, \underline{b}_{k_{r_1}}$ are put in charge of accessing such records and memory areas $\underline{a}_1^{r1}, \underline{a}_2^{r1}, \ldots, \underline{a}_{kr1}^{r1}$ are assigned and saved for receiving them. The busy bits of such memory areas are put into the busy state. In the meantime the next transaction $\underline{r}_2$ is entered on processed like $\underline{r}_1$, and so on for $\underline{r}_3$, $\underline{r}_4$, etc. As soon as one accessing bus, say $\underline{b}_i^{rj}$ has performed and its access, it interrupts operation (as soon as possible) and transfers a record into $\underline{a}_i^{rj}$. Then $\underline{b}_i^{rj}$ is made available for further transfers, while the processing of $\underline{r}_i$ is resumed (notice that the data base record newly entered might summon further data base records). The processing of any $\underline{r}_i$ might involve several outputs into the environment, summoning of transducer (e.g., adders) and augmenting, reducing or modifying the data base. This last operation is performed by putting to the "not busy" state its busy bit and/or posting all of its contiguous memory assigners accordingly.

In the design there is a lot of alternative as to where to place busy bits and continuation registers. For example, busy bits of internal memory areas can be replaced by internal memory of all the memory assigners contiguous to them. Continuation registers of disk or drum access buses can be replaced by those of the internal memory areas from/to which they transfer, and so on.

The total number q of internal memory areas for given f is approximated by the formula

$$q = \frac{f*(n+1)}{m} \qquad (5.3)$$

whenever internal processing can be overlapped to input-output of data base records. Otherwise, if internal processing requires a mean of $1/i$ seconds per transaction, then

$$q = \frac{f*(n+1)}{m+i} \qquad (5.4)$$

By comparing (5.1) with (5.4), for large n and i we can draw the following

Balancing rule: In an on-line, real-time system drawing from an extensive data base the number of internal memory areas devoted to temporary data storage equals the number of accessing buses of the random access storage.

The cumulative size v_1 (in bits) of the necessary internal memory is given by

$$v_1 = q * s \qquad (5.5)$$

where s is the mean size (in bits) of the records (transactions, requests or data base items) weighted with respect to their relative frequencies of occurrence.

Finally, if the active programs which should be permanently in memory require a total of p bits, the total memory size should be

$$v + p$$

LIST as a language for an incremental computer.

1. General

The following two characteristics are commonly found in information system for the command and control of complex, diversified military systems, for the supply of information input for quantitative analysis and managerial decision making, and for the complementation of computer and scientist in creative thinking ("synnoesis") [10].

1) the input and output information flows from and to a large, continuous, on-going, evolutionary data base;

2) the algorithms of the process undergo permanent evolution along lines which cannot be predicted in advance.

Most present day information systems are designed along ideas proposed by Turing and von Neumann. The intent of those authors was to automate the execution of procedures, once the procedures were completely determined. Their basic contributions were the concepts of "executable instructions", "program" and "stored program computer". Information systems based on this conventional philosophy of computation handle effectively only an information process which 1) is "self-contained", in the sense that its data have a completely predetermined structure, and 2) can be reduced to an algorithm in "final" form, after which no changes can be accommodated but those for which provision was made in advance. Consequently, the current role of automatic information systems in defense, business and research is mainly confined to simple routine functions such as data reduction, accounting, and lengthy arithmetic computations. Such systems cannot act as evolutionary extensions of human minds in complex, changing environments.

List-processing computer languages [7] have introduced a flexible and dynamically-changeable computer memory organization. While this feature permits

the manipulation of new classes of data, it does not solve the basic communication problems of an evolutionary system. Each program must still "know" the form of its data; and before any processing takes place, a complete data set containing a predetermined amount of data must be supplied.

Multiple-access, time-shared, interactive computers [8] cannot completely make up for the inadequacies of conventional and list-processing systems. With time-sharing, changes in systems being developed can be made only by interrupting working programs, altering them, and then resuming computation; no evolutionary characteristics are inherent in the underlying system of a multiple-access, time-shared computer. Thus, as preliminary usage confirms, multiple-access, time-shared computer. Thus, as preliminary usage confirms, multiple-access time sharing of conventional computers is useful mainly in facilitating debugging of programs. While such physical means for close man-computer interaction are necessary for progress in information systems, they are not sufficient alone to produce any substantial expansion in the type of continuous, evolutionary, automatic computer service with which this paper is concerned.

2. The problem

A new basic philosophy is under development for designing automatic information systems to deal with information processes taking place in a changing, evolutionary environment. [1,5,6]. This new approach requires departing from the ideas of Turing and von Neumann. Now the problem is not "executing determined procedures", but rather "determining procedures". Open-endedness, which was virtually absent from the Turing-von Neumann machine concept, must lie in the very foundations of the new philosophy.

The basis of the new approach is an "incremental computer" which,

instead of executing frozen commands, evaluates expressions under the control of the available information context. Such evaluation mainly consists of replacing blanks (or unknowns) with data, and performing arithmetic or relational reductions. The key requirements for the incremental computer are:

1) The extent to which an expression is evaluated is controlled by the currently available information context. The result of the evaluation is a new expression, open to accommodate new increments of pertinent information by simply evaluating it again within a new information context.

2) Algorithms, data and the operation of the computer itself are all represented by "expressions" of the same kind. Since the form of implementation of an expression which describes an evaluation procedure is irrelevant, decision of hardware vs. software can be made case by case.

3) The common language used in designing machines, writing programs, and encoding data is directly understandable by untrained humans.

While the Turing-von Neumann computer is computation-oriented, the incremental computer is interface-oriented. Its main function is to catalyze the open-ended growth of information structures along unpredictable guidelines. Its main operation is an incremental data assimilation from a variable environment composed of information from humans and/or other processors. (Still, the incremental computer is a universal Turing machine, and can perform arithmetic computations quite efficiently).

Current research on the incremental computer is aimed at designing it with enough ingenuity to make the new principles as fruitful as the ones of Turing and von Neumann (see [1] and [5]). Some of the main study areas are: the design of the language; the class of external recursive functions and a mechanism called a discharge stack [2] for their fast evaluation; the design of

a suitable memory and memory addressing scheme (the latter problem is being attacked by means of higher order association lists); saving on transfers of information in memory and the use of cyclic lists; avoidance or repetition of identical strings within different expressions through the use of short-hands, and related problems of maintenance of free storage.

The following will present a quite elementary, restricted and perhaps inefficient version of the incremental computer based on LISP. LISP is the currently available computer language which most closely satisfies the requirements of an incremental computer system. The purpose of this presentation is to demonstrate some of the concepts of incremental data assimilation to scientists who are familiar with LISP. Features of a preliminary LISP implementation can be used as a guide in the development of the ultimate language for the incremental computer.

3. Aspects of the proposed solution

Various structures have been proposed for the language of the incremental computer, mainly stressing closeness to natural language (for preliminary studies see [3] and [4]). Here, however, we will consider the case in which this language is patterned on LISP. In this case a simplified version of the incremental computer will be represented by an extension of the normal LISP "EVALQUOTE" operator. This operator, itself programmed in LISP, will evaluate LISP expressions in a manner consistent with the principles of the incremental computer which are presented below. The LISP representations and programs for implementing these principles will be discussed in section 4 of this paper. The LISP meta-language will be used for all examples in the following sections.

i) Omitted arguments:

Suppose func is defined to be a function of m arguments. Consider the problem of evaluating

$$\text{func}\ [x_1; x_2; \ldots; x_n] \qquad (n<m) \qquad (1)$$

Regular LISP would be unable to assign a value to (1). However, for the incremental computer (1) has a value which is itself a function of (m-n) arguments. This latter function is obtained from (1) by replacing the appropriate n arguments in the definition of func by the specified values $x_1, x_2, \ldots, x_n$.

For example, consider the function

$$\text{list 3} = \lambda[[x;y;z];\text{cons}\ [x;\text{cons}\ [y;\text{cons}\ [z;\text{NIL}]]]]$$

If A and (B,C) are somehow specified to correspond to the first and third arguments in the list 3 definition, then the incremental computer should find the value of list 3 [A;(B,C)] to be

$$\lambda[[u];\text{cons}\ A;\text{cons}\ [u;((B,C))]]$$

ii) Indefinite arguments:

In regular LISP a function can be meaningfully evaluated only if each supplied argument is of the same kind -- such as S-expression, functional argument, or number -- as its corresponding variable in the definition of the function. In contrast, the incremental computer permits any argument of a function to be itself a function of further, undetermined arguments. (If these latter arguments were known, then the inner function could be evaluated before the main function, as LISP normally does.) The value of a function with such indefinite arguments should be a new function, all of whose unspecified arguments are at the top level.

For example, consider again the function list 3 defined above. In the incremental computer,

list 3 [D;λ[[u];cons[E;u]];car[u]]

should evaluate to

λ[[r;s][cons[D;cons[cons[E;r];cons[car[s];NIL]]]]]

iii) Threshold conditions

Consider for example the function sum = [[x;y];x + y]. We say that the threshold condition for evaluating a sum isthat both arguments of sum be supplied and that they both be numerical atoms. In general, a threshold condition is a necessary and sufficient condition for completing, in some sense, the evaluation of a function. In regular LISP, it is considered a programming error to request the evaluation of an expression involving a function whose threshold condition cannot be satisfied. In the incremental computer, on the other hand, expressions may be evaluated even though they involve indefinite or omitted arguments (as in (i) and (ii) above). In these cases the evaluation is not complete in the sense that the values are themselves functions which will require additional evaluation whenever the appropriate missing data are supplied.

Occasionally the threshold condition for a function does not equire the presence of all the arguments. For example, the threshold condition associated with the logical funcrion and is, "either all arguments are present and are truth-valued atoms, or at least one argument is present and it is the truthvalued atom representing falsity."

The incremental computer must know the threshold conditions for carrying out its various levels of evaluation. One of the most challenging problems in the theoretical design of the new incremental computer is that of determining

The illustrative program described in the next section employs only the most obvious threshold conditions.

4. The program

Let us consider some of the problems of representation and organization which must be faced in the course of implementing a LISP version of the incremental computer.

i) Omitted arguments:

Since LISP functions are defined by means of the lambda-notation [9], the role of an argument of a function is determined solely by its relative position in the list of arguments. If an argument is omitted, the omission must not change the order number of any of the supplied arguments. This can be accomplished only if each omitted argument is replaced by some kind of marker to occupy its position. Therefore in this LISP formalism for the incremental computer each function must always be supplied the same number of arguments as appear in its definition; however, some of these arguments may be the special atomic symbol "NIL*" which indicates that the corresponding argument is not available for the current evaluation.

The evaluation of a function, some of whose arguments are NIL*'s, is approximately as follows: Each supplied argument (i.e., each argument which is not NIL*) is evaluated, the value substituted into the appropriate places in the definition of the function, and the corresponding variable deleted fro the list of bound variables in the definition of the function. What remains is just the definition of a function of the omitted variables.

ii) Indefinite arguments:

An indefinite argument, as discussed in section 3 above, is an argument which is itself a function of new unknown arguments. The present program assumes that any argument which is a list whose first element is the atom "LAMBDA" is an indefinite argument. This convention does not cause any difficulty in the use of functional arguments, since they would be prefixed, as S-expressions, by the symbol "FUNCTION". However, there is an ambiguity between indefinite arguments and functional arguments in the meta-language. Also, it is illegal to have an actual supplied argument be a list starting with a "LAMBDA". A more sophisticated version of this program should have some unique way to identify indefinite arguments (perhaps by consing a NIL* in front of them).

The treatment of indefinite arguments is straightforward if one remembers that a main function and an indefinite argument are both λ-expressions, each consisting of a list of variables and a form containing those variables. The process of evaluating a function fn of an indefinite argument arg involves, then, identifying the variable v in the variable-list of fn which corresponds to arg; replacing v by the string of variables in the variable-list of arg; and substituting the entire form in arg for each occurrence of v in the form in fn. The treatment of a conditional function containing an indefinite argument is similar although somewhat more complicated.

iii) Conflicts of variables:

The same bound variables used in different λ-expressions which appear one within another "conflict" in the sense that they make the meaning of the overall expression ambiguous. The use of indefinite arguments frequently leads to such conflicts. This problem is avoided in the present system by replacing every bound variable, as soon as it is encountered, by a braid -

new atomic symbol generated by the LISP function gensym.

iv) Threshold conditions:

Certain program simplifications can be made automatically by the incremental computer, if corresponding threshold conditions are satisfied. In particular, if every argument of a function is the symbol NIL*, then the function of those arguments is replaced by the function itself.

The incremental computer is represented by the LISP function evalquote 1. This function is similar to the normal evalquote operator except that evalquote 1 first checks to see if any incremental data processing, of the kinds discussed above, is called for. If so, evalquote 1 performs the appropriate partial evaluations. If the given input is a normal LISP function of specified arguments, on the other hand, the effects of evalquote 1 and evalquote are identical.

A listing of the complete deck for a test run, which includes the definitions of evalquote 1 and all its subsidiary functions is available at the MIT computation centre.

5. Conclusions

We can now make the following observations concerning the use of LISP as the language for the incremental computer:

i) Although perhaps too inefficient to be a final solution, LISP is still a very useful language with which to illustrate the features of a new concept of algorithm representation. It is especially easy to use LISP to design an interpreter for a language similar to, but different in significant ways from, LISP itself.

ii) The program described in this paper is quite limited with regard to its implementation of both LISP and the incremental computer. If a more

complete experimental system were desired, the present system could easily be extended in any of several directions. For example, in LISP, allowance could be made for the use of functions defined by machine-language subroutines, and the use of special forms; in the incremental computer, threshold conditions could be inserted to allow partial evaluation and simplification of conditional expressions.

iii) Replacing all bound variables by new symbols is too brutal a solution to the "conflict" problem; the resulting expressions become quite unreadable. Bound variables frequently have mnemonic significance, and therefore should not be changed unless absolutely necessary. A more sophisticated program would identify those symbols which actually caused a conflict, and then perhaps replace each offending symbol with one whose spelling is different but similar.

iv) When a function of an indefinite argument is evaluated, the form in the argument is substituted for each occurrence of a variable in the form in the function definition. Similarly, when a function has omitted arguments, those arguments which were not omitted are each evaluated and substituted for each occurrence of variables in the form in the function definition. In the interest of saving computer space, we must be sure that what is substituted is a reference to an expression, not a copy of the expression. In the interest of readibility, perhaps the print-outs should similarly contain references to repeated sub-expressions, e.g. in the form of λ-expressions, rather than fully expanded expressions.

BIBLIOGRAPHY

[1] L.A. Lombardi and B. Raphael: Man-computer information systems, lecture notes of a two week course UCIA Physical Sciences Extension, July 20-30, 1964.

[2] L.A. Lombardi: Zwei Beitrage zur Morphologie and Syntax deklarativer Systemsprachen, Akten der 1962 Jahrestagung der Gesellschaft fur angewandte Mathematik Mechanik (GAMM), Bonn (1962); Zeitschr, angew. Math. Mech. (42) Sonderheft, T27-T29.

[3] _____: On the Control of the Data Flow by Means of Recursive Functions, Proc. Symp. "Symoblic Languages in Data Processing", International Computation Center, Roma, Gordon and Breach, 1962, 173-186.

[4] _____: On Table Operating Algorithms, Proc. 2nd IFIP Congress, Munchen (1962), section 14.

[5] _____: Prospettive per il calcolo automatico, Scientia (in Italian and French) Series IV (57) 2 and 3 (1963).

[6] _____: Incremental data assimilation in man-computer systems, Proc.1st Congress of Associazione Italiana Calcolo Automatico (AICA), Bologna, May 20-22, 1963 (in press).

[7] D.G. Bobrow and B. Raphael, A Comparison of List-processing Computer Languages, Comm. ACM, expected publication April or May, 1964.

[8] M. I. T. Computation Center, The Compatible Time-Sharing Systems: A Programmer's Guide, M. I. T. Press, Cambridge, Mass., 1963.

[9] A. Church, The Calculi of Lambda-Conversion, Princeton University Press, Princeton, New Jersey, 1941.

[10] L. Fein: The computer-related science (synnoetics) at a University in the year 1975, American Scientist (49) (1961), 149-168; DATAMATION (7) 9 (1961), 34-41.