The variable size machine.

Classical programming is done for a fixed size machine. The fixed size machine consists of a constant number of components -processors, peripherals, storage locations- all with a permanent existence, an undeniable identity. The identity is reflected in the addressing structure, which is unique in both ways: each address points to a uniqu component and, conversely, each component is identified by a unique address. Later on, I think, I shall call such an address " an identity index".

The classical uniprogrammer has the complete machine at his disposal; he has to decide when comoonents will be used for what purpose, decisions that materialize via the addresses as they occur in and are generated by his program.

In multiprogramming the situation is drastically different. If a number of programs co-reside within the machine, it must be known at any instant of time which part of the machine is at the disposal of which program, in order to avoid a clash of claims (to put it mildly).

For combinatorial reasons it is unacceptable that the programmer decides during program construction which part of the machine will be claimed by his program. (If so, two independently conceived programs could both,say, claim the storage locations from 0 through 1000, thus prohibiting concurrent execution.)

To postpone the decision until the moment of program loading is slighly better, but only slightly. It means that the program text indicates over which areas consecutivity is needed, more general: what amount of freedom is given to the loading program. If addresses -as they are in classical programming- are quantities upon which arithmetic operationsa are performed, the allocating algorithm will need consecutive areas in core store and the ability to use gaps may be absent.

Slightly better still is the technique, in which the programmer claims a consecutive area of store but will address the locations with indices from 0 onwards, indices to which at run time a base address will be added to transform the index into the machine address. As long as the program only manipulates indices, this technique allows reallocations of store by shifting, an operation that has to be accompanied by adjustment of the appropriate base address.

The abone change is significant: the program itself has a local terminology in which to identify storage locations, viz. the identity indices, thar are among the subject matter of the process, viz, the abstraction from the actual storage area used is performed so rigidly that indeed reallocation can be done in a safe and clear way. In practice, this solution has a number of disadvantaqes, due to a lack of refinement.

For one thing: the program is still executed by a fixed size machine, be it that the size is set at the moment of program execution.

For another thing: apart from a private terminology in terms of which a program can identify its private objects, there are common objects, such as library procedures that should be identified as well. The mixture of such terminologies is a problem for which, as far as I know, no satisfactory solution has been found.

Finally, the identification by consecutive index values has two serious disadvantages. On the one hand it forces the program to do it that way, on the other hand, the integers are so "neutral" that the way in which the program asks for its information (by just stating the integer index value) is most unhelpful in the case that the environment has to implement it all with multilevel storage.

What we are looking for is a process structure that defines (and modifies) the size of the machine as the process continues. Furthermore we require that it presents its "needs" -i.e. in the implementation of the variable size machine- in a useful manner. Above aims have a clear technical aspect. Simultaneously we desire a logically satisfactory way in which local and global terminology can be mixed.

If you wish you can regard the variable size machine as a "logical interface" between programs to be executed and implementing susyems. At present, hardware extensions of a configuration tend to give rise to severe adaptations problems for the software. It is urgently hoped (and to some extent even expected) that the notion of the variable size machine will contribute to the solution of this vital problem.

Part of the consequences of the notion of the variable size machine is that it redefines not only its size, but also its structure, in the hope that the implementation can make good use of it. (We shaal have to give a sharper definition of the notion "structure", but loosely speaking the obvious way in which to increase implementation efficiency can always be regarded as the systematic exploitation of structure rather than the ad hoc exploitation of discovered combinatorial luck.) One of the ways in which I hope to structure can be described as "desequencing" and I plan to apply this both to data and processing.

I intend to apply it to date, i.e. instead of specifying storage as a linear sequence -a vector of consecutive elements- the variable size machine must be able to comprise a number of vectors. There is no question about it, that the programmer will benefit from this - he has a kind of "elastic memory" that solves (for him at least!) a number of allocation problems. Also the implementation should be able to benefit from it, certainly as seon as multiple storage of different levels is considere (In the multiprogramming system considered we saw that first the programmer has to fit his variables in a linear store and that thereafter the implementation has to find a long, now consecutive storage area! That seems funny!)

A next point of desequencing is, that I intend the variability of the size also to apply to the number of processes conceptually going on in parallel. If the program "goes in parallel" -at present we must be content with a loose terminology, I am sorry!- this should be regarded as an invitation to allocate more processors, if available.

In the previously considered example (single vector and implicit base address per program) each variable of a program has an identity index and the identity indices are the quantities that have a semantic function within this program. The actual address associated with it has no meaning within this program, it is an "external" value, "only" a consequence of the embedding (as given by the base addresses) and it has only a meaning with respect to the embedding. This is very satisfactory.

As far as the system is concerned the values of the vector elements are non-interpreted: the only thing the system does with these values is moving them around when the emebdding is changed. Also this is very satisfactory.

Remark. Twice I have used the term "satisfactory" without stating explicitly why. I think that I shall return to this question in a later letter.