On not duplicating volatile information.

In EWD718 I have described the assembly conventions for the EDSAC. With one aspect of the problem they dealt quite satisfactorily, viz. with the problem of extracting a sublibrary from the total library of subroutines. A characteristic of the organization was the introduction of a closed nomenclature for the elements of the extracted sublibrary, this in contrast to the open nomenclature for the elements of the total library. Less satisfactory was the free duplication (or the uncontrolled dissipation) of all sorts of allocation results formed in the various stages of the assembly process.

In the EDSAC organization one could get away with the crude solution of duplication thanks to a number of simplifying circumstances. I mention the following ones:

a) Always the whole machine was allocated to the execution of a single program; as a result it sufficed to have an organization that could embed the assembled program in that fixed and constant environment as provided by "the machine".

b) The machine was "simple" in that it had only a one-level store in which each location was as good as any other.

c) The main components of the assembly --master routine and library subroutines-- were "simple" in the sense that they were components of a constant size that was well-known in advance.

In short: the problem of duplicated volatile information was avoided by not introducing volatility. In this note we shall attempt a more systematic study of the ways in which duplicated volatile information can be avoided: we can try to avoid either volatility, or duplication!

A crucial observation is that some duplications are unavoidable, at least if we accept as dogma the purpose of creating an environment from which the open nomenclature that covers the whole library has disappeared. A program may call explicitly the library subroutines A and B , both of which may need library subroutine C ; in that case the preset parameters of A as well as those of B will contain a reference to C . As the dogma excludes the use of the library name of C in those preset parameters, a local terminology has to be duplicated.

Because we want to avoid the duplication of volatile information, these multiple references to C must be in a nomenclature that we can afford to keep constant. In the EDSAC conventions we have seen two techniques. Without the assembly routine M1 the programmer had to decide the storage layout once and for all, and the references to C in the preset parameters of A and B would be in terms of the address of C's "first instruction". With the assembly routine M1 the preset parameters of A and B could refer to C in terms of its "ordinal number". The latter has the advantage that then the references are independent of the lengths of subroutines (as implied by their allocation in store).

If we take a program as the unit of assembly, we are allowed to take each subroutine's ordinal number in that program as providing a sufficiently constant nomenclature that we may duplicate in the preset parameters.

Having thus avoided duplication of first addresses of subroutines on the input tape we should explore what would be involved in keeping them volatile. The answer is simple: don't copy them, neither during program assembly, nor during program execution. Still confining ourselves to the situation in which the complete machine is at a single program's disposal, we come to the following machinery during program execution.

To start with we would need --with apologies for the acronym-- a program routine table PRT with an entry for each ordinal number used in the program; each entry will "lead to" --and here I have intentionally used a vague terminology-- the corresponding routine's current "address of first instruction".

The program routine table PRT is needed to call a routine: given the routine's ordinal number it will provide the current address of the routine's first instruction, an address that is needed for the selection of the next instruction to be executed. The access to PRT in order to process the ordinal number of the called routine introduces some overhead on the procedure call; as soon as the routine has been called, however, its successive instructions can be selected with full speed as before, if we maintain the EDSAC's "instruction counter" as the register containing the address of the location containing the next instruction to be executed.

Note, however, that maintaining such an instruction counter is a dangerous thing to do: upon subroutine call it is filled with a copy of "the address of the routine's first instruction", information that we wanted to keep volatile! We must keep such an undesirable copy under strict control; I suggest that we do not allow further copies of it to be made. This implies that we must provide an alternative for the EDSAC way of supplying at subroutine call the return information. Instead of copying the contents of the instruction counter --i.e. supplying the current address of the call-- we must supply a non-volatile characterization of the place of the call. Within the context of the program such a non-volatile characterization is provided by the ordinal number of the calling routine, together with the call's position relative to the location of the calling routine's first instruction. From the need to provide such non-volatile return information we deduce that in addition to PRT the machinery requires a further status variable, CRN say, for the "Current Routine Number". In the subroutine call CRN is redefined and set to the ordinal number of the routine called, while its old value is saved in the return information; upon return from the subroutine it is reset to the ordinal number of the calling routine, whose execution is continued. (The status variable CRN can be viewed as a non-volatile extension of the instruction counter.)

So much for not duplicating addresses of first instructions. What about the ordinal numbers? As long as each single program has the whole machine at its disposal and can be regarded as a unit of assembly, the situation is relatively simple.

On the final tape each library subroutine is punched in terms of closing letters and preceded by control combinations that define the preset parameters in terms of ordinal numbers that are substitued for the closing letters during the input of the program. As a result the subroutine call as stored contains, just as the return jump generated during program execution, the ordinal number of the target routine. The entries of the PRT contain for each routine the address of its first instruction.

An alternative would have been not to process the preset parameters during program input by substituting them for the closing letters, but to store them in little tables, one for each subroutine. We might call them SCLT's, short for Subroutine Closing Letter Tables. The SCLT belonging to a subroutine would contain an entry for each closing letter used in the subroutine; each entry would mention the ordinal number of the subroutine corresponding to the closing letter in question. Each PRT-entry would lead, besides to the address of the subroutine's first instruction, to its SCLT, e.g. by mentioning the latter's starting address. The bit patterns representing the library bodies are now allowed to contain their original closing letters; during program execution each closing letter can be translated into the corresponding ordinal number because the current SCLT can be accessed because its starting address can be found in the PRT under control of the current routine number CRN . (This would be a second use of the status variable CRN .)

The second alternative has been mentioned, not because it is particularly attractive, but because the first alternative --substituting the ordinal numbers all through the binary texts-- does not work as soon as we would like to store several programs simultaneously, but in such a way that they can share the bit patterns of library subroutines used by several of them. The ordinal numbers supply <u>per program</u> a closed terminology for the library subroutines used in that very same program. The sublibraries extracted for the different programs will, in general, overlap only partly; it is, in general, therefore unavoidable that the same library subroutine will be denoted in different programs by different ordinal numbers. In the case of simultaneously stored programs sharing the bit patterns of common library subroutines the first alternative therefore doesn't work.

In order not to complicate matters still further we shall assume from now on that the mutual dependence of library subroutines as to be catered for by program assembly is fixed by (the current state of) the library, i.e. we now disregard the form of programmer control over the assembly process as mentioned in the Note on pg. EWD718 - 7. Under that assumption the mutual dependence of the library subroutines is fully determined by the union of the subroutines possibly needed by each of the programs. For the sake of the argument I assume that that union is still a small fraction of the total library, small enough, in any case, to justify extraction and introduction of a "union nomenclature" that only covers the union of the library routines possibly needed by the currently loaded programs.

The fact that we have to play a new sort of game may dawn upon us as soon as we realize that, without renumbering, it is impossible to keep the union nomenclature fully closed. When only the first program is loaded the union nomenclature can be kept closed. When the second program is loaded in addition to the first, we can still keep the union nomenclature closed by assigning the next numbers to the further library subroutines needed by the second program. Because at the moment of the loading of the first program it was unknown which library procedures it would share with the second program, gaps in the union nomenclature must be expected when the execution of the first program terminates before the execution of the second program does so. By always assigning in the case of extension the first currently free number --"trying to fill the left-most gap"-- we seem to do the best we can do when we wish to keep the nomenclature as well closed as possible: thus the maximum value assigned never exceeds the union's maximum size in the past. In this case I think that the first free number strategy is quite acceptable and I assume in the sequel that it has been adopted for the union nomenclature.

It stands to reason to expect, analogously to the PRT for each program, for the installation as a whole a URT , short for Union Routine Table, with an entry for each routine of the library that may be needed by at least one of the programs currently loaded. In contrast to the PRT's, the URT need not be fully closed: in the middle of it we may have entries marked:

"currently free". While a member of the union of selected subroutines, each subroutine has a constant "union number", which acts as the selector for its entry in the URT .

Again we have essentially two choices for the bit pattern to be stored. In the one case we regard the union nomenclature sufficiently constant to allow its duplication. The role of the PRT in uniprogramming is fully taken over by the URT , and all programs and all subroutines refer to the subroutines in terms of union numbers. The machine needs the status variable CRN as before, the only difference being that its contents is now interpreted as union number, the ordinal numbers having disappeared from the scene. Also all return information is represented in terms of union numbers. A consequence is that when the same program is loaded at two different occasions, the bit patterns representing the program at those two occasions will differ from each other, the union numbers used being all different. Such free duplication of union numbers makes the internal representation of each program dependent on the environment in which it happens to be embedded, and I do not regard that as very attractive: such an organization really forces us to treat union numbers used by a program as non-volatile during its complete execution.

Suppose now that in our multiprogramming environment we would like to be able to interrupt a program execution, but in such a way that an interrupted computation can be resumed at a later stage, say a week or a month later. Suppose, furthermore, that during the interruption the installation proper must be able to shed the interrupted computation completely, i.e. all information pertaining to the interrupted computation will be dumped on an external medium --a magnetic tape, say-- and the installation proper must be allowed to continue operating fully independently of the fact that there exists an interrupted computation to be resumed at some later moment. Well, that implies that we cannot allow the interrupted program to "reserve" union numbers: upon resumption it has to refer to the library subroutines in a different union nomenclature. In other words, we now have to consider the union nomenclature as volatile and, therefore, have to ask ourselves how its free duplication can be avoided.

We can solve the problem by a proper combination of the machinery considered previously, extended with one status variable (which is really a consequence of the transition from uni- to multiprogramming); I shall call this statusvariable CPR , short for "Current Program Reference". In the order of increasing multiplicity, the machinery consists of the following components.

0. For the installation proper as a whole:

0.0 a Union Routine Table URT

selector: union number

entry : address of subroutine's first instruction

Remark. Duplication of union numbers occurs, but is restricted: a routine's union number occurs in one PRT-entry of each program that needs the routine, directly, or indirectly. (End of Remark.)

0.1 a status variable CPR equal to the starting address of the PRT of the program currently under execution

0.2 a status variable CRN equal to ordinal number of the currently executed routine, as assigned to it in the program currently under execution.

1. Per program loaded:

1.0 a Program Routine Table PRT

selector: ordinal number

entry : routine's union number (see Remark under 0.0)
starting address of routine's SCLT in this program.

Remark. Duplication of ordinal numbers occurs, but is restricted: a routine's ordinal number in this program occurs in one SCLT-entry for each subroutine selected for this program that refers to it; furthermore ordinal numbers are duplicated in the return information. (End of Remark.)

2. Per program loaded per subroutine selected for it:

2.0 a Subroutine Closing Letter Table

selector: closing letter

entry: ordinal number in this program of the subroutine the closing letter refers to (see Remark under 1.0)

The bit patterns representing the library routine bodies contain their original closing letters. The closing letters provide a terminology local to the binary text, which is translated via an SCLT in the terminology local to the program in which the routine is invoked; via a PRT this terminology local to the current program is translated into the union nomenclature. (Note how the PRT is determined by the current program, and the SCLT by the current program/routine combination.)

It first sight it seems a tortuous way of doing. Consider a library routine A with union number UA , calling a library routine C with union number UC . Let A be called from two programs in which it has the ordinal numbers OA1 and OA2 respectively, and in which the library routine C has the ordinal numbers OC1 and OC2 respectively. The closing letter under which the binary text of A refers to C has always to lead to UC, but the — due to technical trouble in the typewriter continued in handwriting— path along which depends on the identity of the current program: with OAi selecting in PRTi the SCLT is found that contains the entry OCi; with OCi selecting in PRTi we get UC . The point is that besides UC, which may not be duplicated, we need OCi as well, namely for the return information and for the setting of CRN.

Remark. In the meantime the flexibility mentioned in the Note on pg. EWD718-7 is back again. (End of Remark.)

Historical Note. The last organization described is much more ambitious than I dared to consider while designing the THE Multiprogramming System in the mid-sixties. The problem of sublibrary selection was avoided by always storing the complete library, which wasn't very big, in the installation proper, which, thanks to its drum, had a store that, at the time, was considered large.

Information to be stored was subdivided into so-called "pages" of 512 consecutively numbered words; during its lifetime each page was identified by a unique bit pattern, thus providing a "global" nomenclature, which was freely duplicated all through the system. As a result the library was very hard to change; even extending it was not simple.

Furthermore it had to be a so-called "load-and-go" system: during program compilation one or more page frames in core would be allocated to the program for the course of its entire execution and the numbers of these frames would be freely duplicated all of the object text! Later we had many occasions to regret our loss of flexibility. (End of Historical Note.)

Remark. In the last arrangement the duplication of union numbers is so modest that renumbering can be considered. As a result the could remain a closed terminology. (End of Remark.)

* * *

While carrying out the above analysis I was pleasantly surprised when observing how clear the guidance was that came from the combination of the two design principles — avoid the frequent processing of open nomenclatures and avoid the duplication of volatile information — . The principles themselves are not very surprising, but their combination is surprisingly effective.

A final remark should be made. What should be regarded as "volatile" or as "non-volatile" greatly depends on the envisaged way of using the installation: remember that union numbers should be regarded as volatile when we want to remove interrupted computations from the installation proper in such a way that they can be resumed later. But observe the influence on the proposed machinery! This was an influence at quite a detailed level. It gives an indication why machine design is so hard. I think I would be willing to defend the last arrangement, even if it turns out that no one wishes to interrupt computations. Just to be safe!

Plataanstraat 5
5671 AL NUENEN
The Netherlands

29 October 1979
prof.dr. Edsger W. Dijkstra
Burroughs Research Fellow