Assembly conventions for the EDSAC.

In this note I shall give a survey of the assembly conventions as developed for the EDSAC --later called "EDSAC 1"-- in order to subject them to a later analysis. The choice of the EDSAC conventions has been inspired by three considerations:

1) they have been well-documented (if we mean by "well-documented": fairly completely; by today's standards the book [1] is pretty unreadable!)
2) the EDSAC in Cambridge, England, was --and shall remain forever!-- the world's first program controlled automatic digital computer (EDSAC standing for Electronic Delay Storage Automatic Calculator)
3) the subtitle of [1] is: "With special reference to the EDSAC and the use of a library of subroutines", and the second part of this subtitle explains why this machine deserves our special attention. In retrospect I think it quite impressive that the group in Cambridge realized, right from the start --note that this was in the late forties!-- that there was no point in designing and building a machine without designing a discipline for its usage as well.

The store consists in principle of consecutively numbered locations of 17 bits --usually the text refers to "the address of a location" but the term "serial number of a location" is also used occasionally-- . In the true von Neumann style, such a 17-bit location contents admits two interpretations, viz. that of a "short number" and that of an "instruction" or "order".

Historical aside. "A special feature of the EDSAC is the possibility of combining any two consecutive storage locations (provided the first one has an even serial number) into a single long storage location capable of holding a number with 35 digits. [...] It is possible to accommodate 35 digits in a long storage location, and not 34 only, since in the ultrasonic store of the EDSAC the digits of successive numbers are stored end to end and one digital position between each is left unused; when two storage locations are combined this position can be used to contain an extra digit (sometimes called the "sandwich" digit)." The existence of such long locations explains the constraint for some subroutines that "the first order

must have an even address".

In the section "Input and output subroutines" --when dealing with the input of long constants-- the sandwich digit is responsible for the paragraph: "When a subroutine contains only one or two long constants, an alternative to the use of an input subroutine [for the conversion from decimal to binary representation] is to put the constant in as two short numbers. A difficulty arises, however, because there is an unused digit between each two short numbers (the "sandwich digit"). The method can be used, therefore, to put in a number which has a zero in the 17th position after the binary point. This is not a serious limitation, since either the number itself or its complement is of this form unless it is an odd multiple of 2^{-17} ." (End of historical aside.)

In store, each instruction occupies 17 bits, from left to right:
5 bits for the functional part
11 bits for the address part
1 bit for the indication short/long (for some instructions, such as the jump instruction, not meaningful).

When stored, a subroutine always occupies a number of consecutive locations. Because an instruction of a subroutine may have to refer to one of the words of the subroutine itself, the bit pattern representing the subroutine in store may depend on the actual place the bit pattern occupies in store. Such reference to one of the own words of the subroutine arises
1) in internal jump instructions --used as a tool for the implementation of alternative and repetitive constructs--
2) in the use of (numerical) constants occuring in the bit pattern
3) in the use of variables allocated within the bit pattern (independent of the question whether these variable words are also interpreted as instructions or not)
4) in supplying in one of the registers of the arithmetic unit return information: the calling sequence of a (closed) subroutine contains an instruction that places itself in a register.

Note. Observe that allocating variable information within the bit pattern

is not the only cause for the need within a subroutine to refer to itself. (End of note.)

Note. If instructions could address relative to the current value of the instruction counter or, better still, relative to the address of the first order of the subroutine currently being executed --a convention that would probably require a special register reserved for that purpose-- the bit pattern representing a subroutine in store would no longer need to be dependent on its position in store. (End of note.)

Besides being dependent on its own position in store, the bit pattern representing a subroutine in store may also depend on the positions of other subroutine or data areas the subroutine needs to refer to.

All that has been said above about subroutines is also applicable to the main program. The only potentially different role --about which more later-- of the main program is that it need not be placed in the subroutine library.

By today's standards the store of the EDSAC was very small. In the instruction format we have 11 bit available for the address part, the machine was designed to have a store of 1024 short words only, and I am not sure whether more than 512 short words have ever been built. (The more mercury tanks, the more severe the technical problem of keeping them with sufficient accuracy at the same temperature.)

As a result it was understood, right from the start, that the total subroutine library would be many times larger than the total storage capacity of the machine. As a result the subroutine library must contain for each subroutine a representation that is independent of its actual position in store, when actually used.

Each subroutine in the library was punched on a separate piece of five-hole paper tape. On the tape each instruction again consisted of function part and address part. The function part was represented by a letter --supposedly of some mnemonic value-- , the address part was repre-

sented by a (decimal) integer, followed by a so-called "closing letter"; of the 15 closing letters available, 3 had a constant meaning, the remaining 12 were available to control, which bit pattern would be derived from the punched subroutine text, in which a different closing letter would be used for each subroutine or data area referred to.

Because the control needed is purely additive, the following arrangement works. During the reading of (a copy of) a library procedure each next word to be stored is formed by building up (the binary representation of) the instruction as represented by function letter and following decimal integer, and by then increasing the result by a "preset parameter" that depends on the closing letter. On the final tape each copy of each library subroutine is preceded by a series of "control combinations", defining for each of the (variable) closing letters used in the following library subroutine the value of the corresponding preset parameter.

Historical note. I quote section 6-2 "Storage of library subroutines." from [1]. "Subroutines in the library are punched on colored tape so that they can easily be distinguished from program tapes, which should be white. Several copies of each subroutine are provided and when not in use each copy is rolled in a small cardboard box. The boxes are filed in serial order in a steel cabinet. The master copy of each subroutine is kept under lock and key and is used only when all existing copies of the subroutine are damaged. The master tape is then used to prepare further copies by means of a duplicator. All copies must be checked against the master, by means of a comparator, before being put into the library for general use." (End of historical note.)

The mechanisms sketched above are used in two ways. In the one way the programmer decides explicitly the complete storage layout of the final program, i.e. the addresses of the first orders of all subroutines and of the first numbers of all data areas. In terms of these chosen addresses he can formulate all the control combinations defining the preset parameters and the main program. The authors add: "A few spare locations can, however, be left between the subroutines if desired. This reduces the possibility of error arising because of a miscalculation of the locations required by

a subroutine and enables corrections involving a slight increase in length to be made to a subroutine without renumbering. Such corrections often must be made to subroutines which have been specially constructed for the program."

The other way is introduced as follows: "[In the examples shown] the programmer has to decide where the master routine and each subroutine are to go in the store and to insert the correct addresses in the orders in the master routine which call in the subroutines. The object of an assembly routine is to relieve the programmer of these and other mechanical tasks." (In view of the modest size of the store, and hence of the largest program that could be composed for the machine, the argument in favour of the assembly routine doesn't sound too convincing!)

Data areas are numbered in the order in which they occur at the beginning of the tape; they are followed by master routine and the subroutines, which are also numbered in the order in which they occur on the tape. References to subroutines or to data areas --either in master routine or in the control combinations defining the preset parameters-- are essentially expressed in terms of these ordinal numbers. An assembly routine named "M1" places these areas and routines consecutively in store, at the same time filling in the starting addresses as they become known. The fifth note deserves to be quoted: "If storage space is short, M1 may be placed where it will be overwritten by the last subroutine of the program."

Note. On the one hand it is a feat of ingenuity that the same input program --the so-called "initial orders"-- could be used with and without the assembly routine M1 . I also think it telling that the use of the assembly routine M1 was optional; not using the assembly routine saved a word per routine and that may have been the reason. Another reason may have been that, when all was said and told, the service provided by the assembly routine wasn't that impressive. (End of note.)

Let us now try to analyze the EDSAC assembly conventions a bit further. The purpose of this analysis is the isolation of its separate components and a more clear understanding of their roles.

We begin by remarking that, when assembly routine M1 is used, library subroutines are referred to in four different ways

1) by their "library name", as in the catalogue --in the following is shall use the term "library name" to refer to the name of a subroutine, viz. the name it has in the library--
2) by a "closing letter", as on the library tapes
3) by their "ordinal number" as they appear on the final tape
4) by their "address of first instruction" in store.

All four --library names, closing letters, ordinal numbers and addresses of first instructions-- can be represented by bit sequences and, as such, all could occur on the final tape. The first question to be answered seems to be: why do the library tapes refer to each other by means of closing letters, and not by means of library names?

In the case of the EDSAC the answer is simple: the processing of a library name, in particular the associated selection --of the contents of the appropriate cardboard box from the steel cabinet-- was deemed beyond automation and, hence, kept outside the machine. Hence no library names on the final tape, hence the need for other nomenclature.

Because this answer is so simple, it might be instructive to think for a moment about the situation in which such automatic selection of a standard subroutine text, given its library name, would be possible, but only "after a fashion". With "after a fashion" I mean that such automatic selection, though possible, is a most painful process: we may think about the whole subroutine library --the contents of the steel cabinet, so to speak-- being punched on one huge reel of paper tape. In that case the processing of a library name is sufficiently painful to make it obvious that a major purpose of the game is to represent the library subroutines selected in such a way

that thereafter library names need not to be processed anymore.

As soon as we remember our target, i.e. the representation of the selected library routines that is free of library names, it is clear that our most attractive source is one in which the bulk is free of library names. Well, this is exactly what the closing letters do for us: on the huge reel we could have each subroutine body exactly as it is punched now on the tape in the cardboard box, but preceded by the analogue of the control combinations for the preset parameters; on the huge reel these control combinations would define the meaning of the closing letters that are used in the subsequent body in terms of library names.

The closing letters appear as a local terminology --local to the tape representation of the subroutine-- for reference to its environment, and we are free to choose the most convenient terminology.

When members of a set are essentially identified by (an equal number of) successive integers, this is sometimes called "a closed nomenclature". Closed nomenclatures have the advantages of being compact and being easily processed, viz. as subscripts in linear arrays, this in contrast to "open nomenclatures", the processing of which requires searching, hashing or other "associative" techniques. (Names of people --assumed to be all different-- form an open nomenclature: an arbitrary sequence of letters is almost never the name of an existing person; telephone numbers in a town, however, form an almost closed nomenclature. Quite wisely the telephone exchange only accepts the closed nomenclature, leaving to us the searching in the directory in order to discover what to dial!)

Note. When elements of a set are identified by means of a closed terminology, the membership test is trivial: the number must be at least equal to the lower bound and at most equal to the upper bound. The fact that the cost of this test is independent of the size of the set has been cleverly exploited in a number of surprisingly efficient graph-theoretical algorithms designed in the 70's by R.E.Tarjan and others; the algorithms could be arranged in such a way that all subsets to be considered would

always contain consecutively numbered vertices. Run-time legitimacy tests of addresses as encountered or generated during program execution form another application. (End of note.)

The successive closing letters are no more than a coding for the successive natural numbers: the local terminology in which the tape representation of a subroutine refers to its environment is closed. The vital importance of the local terminology per subroutine is not that it is closed --that is only very nice-- but that it allows a representation that is independent of the actual environment: the same representation of the bodies is as meaningful on the huge reel of paper tape that contains the whole library, as it is on the final tape that contains only the subroutines selected.

Note. For the sake of completeness I should mention that in the EDSAC assembly conventions the protocol of the preset parameters and the closing letters is not only used for the extraction of the "sublibrary" consisting of the library subroutines actually used in the program: it also gives the programmer the possibility of additional control. We may have two library subroutines, B1 and B2 say, semantically equivalent, but the one faster and the other shorter. Another library routine may need either B1 or B2, and the choice can be left to its user, who can express his preference in a preset parameter. (End of note.)

The dependence on the actual environment is concentrated in the definitions of the preset parameters. On the huge reel it had to be in terms of library names, these forming the only terminology that covers the whole library. But we wanted a final program tape free from library names, hence a local terminology for the set of selected routines has to be introduced. Having seen the advantages of a closed terminology, the introduction of the "ordinal numbers on the final tape" is no longer a surprise.

In the final stage, i.e. during the input of the program, assembly routine M1 allocates all routines and substitutes (or adds) "addresses of first instructions" all over the place, as required by the EDSAC hardware

that transmits the bits of the address part of the instruction under execution directly to the selection register of the store (or to the instruction counter if it is a jump instruction). When no assembly routine is used, this distribution of position dependent information all through the program text takes place in two stages: the programmer, who composes the final tape, distributes "addresses of first instructions" up to the preset parameters --thus bypassing the terminology of the ordinal numbers-- , and the input program takes care of their final distribution, from the preset parameters into the bit patterns representing the subroutine bodies in store.

Note that the final stage, as sketched in the previous paragraph, whether done with M1 or without, combines two distinct elements we had better distinguish between: the routines are allocated and the outcome of this allocation decision is distributed all through the program text. Such allocation decisions are in a very special way typical: they are both irrelevant and unavoidable. When we design a library text, the number of independent references to the environment decides the number of preset parameters, and when we make it a rule to use them in order, this number determines which closing letters we are going to use. But with N independent references we can pair the closing letters and the objects referred to in N! different ways! The way in which the routines get their ordinal number on the final tape is equally arbitrary. And so is the allocation of store regions to the different routines. Arbitrary, and therefore irrelevant, as they are, yet those decisions have to be taken. (It does not suffice to develop the theory that collisions can be avoided if every verhicle keeps the same side of the road: in practice a country has to choose between "keep left" and "keep right".) It is very appropriate to leave such irrelevant but unavoidable decisions to the machine.

The distribution of the outcome of the allocation decisions all through the program text is, however, a completely different matter. No further decisions are taken, the only thing that happens is that the values decided upon are copied in all sorts of places.

In the case of the EDSAC this duplication, all through the program

text, of such allocation information was dictated by the structure of the hardware, which was kept as simple as possible. We should, however, always remember its consequence: information that has been duplicated all over the place is very hard to change. As I hope to explain later, allocation decisions are very often typically the kind of decisions one would like to revoke. Under those circumstances a machine structure that, for supposed reasons of efficiency, forces extensive duplication of information that, for other reasons, one would like to keep volatile, can become a great nuisance.

The supposed benefits of duplication are always easily expressed and quantified: simpler hardware performing more instructions per second. The ill effects of the ensuing loss of flexibility in machine usage are more subtle and less easily quantified. They require more foresight than many a machine designer has shown to possess; as a sad result the penalties of operating system overhead can be impressive.

[1] Wilkes, Maurice V., Wheeler, David J. and Gill, Stanley, The Preparation of Programs for an Electronic Digital Computer. (With special reference to the EDSAC and the use of a library of subroutines.) Addison-Wesley Press, Inc., Cambridge, USA, 1951.

Plataanstraat 5
5671 AL Nuenen
The Netherlands

18th October 1979

prof.dr.Edsger W.Dijkstra
Burroughs Research Fellow