CLB
2-27-79
1

This paper was written in January, 1957. At that time I was Chairman Pro Tem of the SHARE system design Committee; this committee had been given the charter to design & produce an operating system for the forthcoming IBM 709 at the previous SHARE meeting (fall, 1956). My job was to organize the committee, find a permanent chairman, and start the ball rolling. The first meeting took place at RAND on 2 Feb., 1957, at which time I submitted these thoughts. As you know, the result - several years later - was the SHARE Operating System (SOS) for the 709/90 which was widely used (in competition to IBSYS - the mfgr's. attempt.

Throughout, the thread of "structuring" is to be found, but more subliminally than explicitly. I was personally advocating a system that would by the very nature of its restrictions force some structuring of the programs. (Note the absence of references to higher level languages - FORTRAN - 1956 was fairly early and IBM's Fortran very restricted in capability compared to machine language (I/O, alpha, diagnostics, etc. all were primitive) and "efficiency still very much in vogue. We had yet to learn to hide our inefficiencies in the operating system and its access methods where it was less noticable and more respectable than in an individual application pgm!)

Getting any system designer to agree that restrictions (even conventions) on programming techniques were good, was a damn near impossibility back then. For an example, witness the reception accorded "PACT" in mid '56. (See ACM Conf. Proc. (Philadelphia)). (PACT was a cooperatively-produced "compiler" for the 701(!) which had to overcome lack of floating-pt. hardware - it pre-dated fortran, and was not interpretive - and which was rather highly constrained.)

My comments surface on p. 18, but apparently didn't cut much ice. SOS reflects most of the ideas in this paper, but, alas, none relevant to structuring.

SOME THOUGHTS ON A PROPOSED ASSEMBLY-LOADER

The present symbolic assembly schemes are rather inefficient for a number of reasons; this proposal is an attempt to eliminate these deficiencies. First, the usual procedure consists of working with at least two decks simultaneously: The original symbolic deck with one instruction per card, and the resulting binary deck. Extensive changes are usually made by going back to the original symbolic deck, while minor changes are effected by the use of binary or octal correctors. Thus it usually happens that there are at least three versions of a problem at once: The symbolic deck which was not kept up to date, the binary deck with its change cards, and the listing produced by the assembly program.

The difficulty arises because, with present input-output equipment, there is a tendency to avoid reading the symbolic deck whenever possible--it seems reasonable to assume that if there was no reading or listing time associated with an assembly, it would be perfectly feasible to assemble and run for each pass of the problem. There is still the objection made by some people, however, that there are really too many symbolic cards to conveniently handle each time it is desired to run a problem. The binary deck takes care of this problem by compressing the instructions to (on the 704) 22 per card, but, unfortunately, destroys the symbolic information originally written down, and requires cross-reference to be made via consultation of the assembly listing. It is felt that a number of these problems are eliminated by the proposed scheme. Note also that a high percentage of runs on a machine are "first time" runs in that at least a small change has been made in the code since the previous run. The time consumed in assembly--both machine time and elapsed time--is a deterrent to the to-be-preferred method of making all changes and corrections symbolically and assembling for each pass.

It might be mentioned at this point that the trend (at least of the more sophisticated programmers) is away from any knowledge of actual machine locations and storage assignments, even during the debugging stage of the coding. BACAIC, of Boeing, has been doing this for at least a year, and it is to be presumed that few people will actually make reference to either the SAP format coding produced by FORTRAN or the assembled equivalent thereof. In PACT, for example, the usefulness of this information lies principally in the fact that this is the only way that we can tell what still isn't working in the compiler itself. (Provision should be made in every system, it is felt, for obtaining machine language equivalents of a problem, principally for the purpose of diagnosing machine failures.)

Let's start out with a brief description of how the proposed system might operate. A problem has been coded (say in SHARE language) and the symbolic cards are sent to the machine for processing. The programmer receives back a binary deck, much smaller than the original symbolic deck, which, however, contains ALL of the information of the symbolic deck, including the comments on the right hand side of the page. We hope that the symbolic deck gets thrown away--perhaps this is a duty of the operator. The new deck is called the Alpha deck, and is all that the programmer works with from this point on. (Undoubtedly someone would eventually code a routine to reverse this process; whether or not it would be desirable is doubtful.) The exact amount of compression in this deck may be estimated as follows. First of all, if a card consisted of eighty random Hollerith characters, at the minimum it could be represented in 13 1/3 words of BCD information or about a little more than half the original space. On the other hand, if the card contained an instruction such as LBT, not much more than eight bits of information (assuming a machine such as the 704 with about 100 operations in the extended order list). This is a saving of about 100 to one; that is, 100 such orders could be punched on one card. In the actual case, on the average, it is felt that more than 22 instructions, in symbolic form, could be punched on one card, assuming no comments have been written. This is more efficient than punching the instructions in assembled binary form!! When comments are added, we might expect that an average of 12 Hollerith characters per instruction would suffice, and that about eight symbolic instructions might be punched on one card.

The above conclusions are a direct result of the fact that the information punched on one symbolic SHARE instruction card is a very small amount of the possible information which could be put on one card, and that, in addition, the information written down is highly redundant. Encoding schemes have been proposed many times for information which have these characteristics--we are just making one more. Later paragraphs will talk about this in more detail. The only comment to be made at this point is that probably the instruction part of the deck and the comments part of the deck would be physically separable, so that comments need not be loaded when they aren't going to be read.

Back to the process. The coder now desires a code check run. He sends the ALPHA deck to the machine room, along with his data, and a "Load and go"assembler is added in front of his deck. The card reading time is relatively small--it could turn out to be less than required at present with absolute binary decks--and the assembly process is likewise rather insignificant. (Unless, of course, we make the encoding scheme so complicated that the time necessary to decode gets to be long--in which case we don't try to be so efficient of space and start thinking of time.) The fact that the assembly is made at load time doesn't restrict the machine available to the program to be run in any way, except perhaps in the rare case when there are no drums available on the machine and all tape units are taken up by permanent file tapes. (Someone will always be around to dream up a case in which the system will fall flat on its face.)

His program having been assembled and executed, the programmer starts to look for his mistakes. (Admittedly, if it runs right the first time, and never needs any revisions, we probably have wasted our time.) He finds a few of them, makes the corrections he feels necessary in symbolic form, and punches them up. These new cards, in addition to having a regular SHARE symbolic instruction, would contain the following information, for example: "This is a change to the instruction at ABQ+4", "This is the first of a series of instructions to be inserted following SUM-3", "This is the last of the series", "Delete the instruction at PQR", "Delete a block of N instructions starting at ERROR+7". Abbreviated notation would be used to designate what the change card means, probably requiring about nine or ten card columns at the most. In addition, comments could be put on the cards.

The changed cards are now put in front of the ALPHA deck, and shipped back to the machine room. The "Load and go" process reads and makes all changes in the deck as requested, looking, or course, for obvious errors in the process, and does the problem. This procedure is repeated until the problem is checked out.

There is one other frill to be added to the process, however, and that is that the coder may receive, at any code check pass, a new, up-to-date ALPHA deck with all of his changes incorporated. This is probably an absolute necessity, as few of us can resist the temptation to make changes on changes on changes, etc., until we are rather confused as to what we really <u>do</u> have. At the same time, for those who feel that it is absolutely necessary to know where and what everything turns out to be inside the machine, a new listing of the assembled code could be produced. Peripheral equipment could be used for making the listing, giving the programmer time to see that the code didn't work, the ALPHA deck, and hence the listing would be no good, and it would never have to be made.

Some other benefits accrue under this system. It is relatively easy, and we feel highly desirable, to set aside part of the memory for information <u>about</u> the code to be used by any of several debugging programs which could then produce their output also in the symbolic language of the programmer. Certainly any problem which comes close to using <u>all</u> of the high speed memory available will certainly overflow the memory about three or four code check passes later.

We have, in the above paragraphs, proposed a system which, we feel, has a number of advantages over our present method of operation. At any one time there is only one deck to contend with, and, likewise, only one listing. If at any time the coder is in doubt as to what his deck contains, he can call for a new listing and get this in his original symbolic form, with comments. If, as so often happens, the original program becomes a source for another, the "source" deck may be reproduced on the 519, and this will evolve into the new program.

A few comments now about the mechanization of the encoding and assembly process. Some sort of "optimum" (?) encoding scheme would require as few bits as possible per symbolic instruction card. What follows has not been thought out entirely, as will become obvious. Let's treat the symbols themselves first. On one pass through the original deck, each of the symbols is extracted, and placed in a table in an order which depends upon the number of characters in that symbol. In addition, a count would be kept of the number of symbols with each number of characters, and this information would precede the symbol table itself. Thereafter, each symbol would be replaced by the relative location within the table at which it would be found. Since few codes have more than a few hundred symbols, about eight bits would suffice. The following technique here would save quite a bit of this eight bits on the average. Let the lack of any symbol be denoted by a single zero bit--then the space normally required by the eight bits would not need to be used at all. (It should have been remarked that a problem with 200 symbols of an average length of three characters would require about five binary cards.)

As was suggested above, the eighteen BCD bits of the operation part of the instruction would be replaced by a maximum of seven bits representing one of the 100 or so possibilities, and even this number might be reduced, on the average, by assigning a shorter representation to the more probable instructions--say add--and a longer representation to the less probable ones--say store left quotient. The address, tag, and decrements could be broken up into their symbolic and absolute parts, and represented in a similar fashion. Very likely each instruction should carry a tag telling which of the five parts were present and which were absent; there are sixteen possibilities (every instruction, we hope, has an operation part) and the tags for 9 such encoded instructions would fit in one word. No information need thus be carried in an instruction's final encoding for any part which did not appear.

It is proposed to have a single bit in each instruction signifying whether or not that instruction card contained a comment. The comments themselves would be punched as a separate deck, and could be thrown away if the coder was convinced that his memory was good enough--or if he didn't want to give any clues to the poor fellow who has to decode the problem. Then, too, these need not be loaded when they would be of no use--such as a pass which did not produce a listing or a new ALPHA deck.

It looks as if the above scheme--apart from the possibility that the "Loader-and goer" might be a dilly to code, and apart from the fact that this whole thing is new and unfamiliar and hence probably unattractive--could form the basis for a system of using a machine. (It is probable that the gains from compacting the code into the ALPHA deck would be less marked on a machine other than the 704, which is rather wasteful of instruction space.) Although most of the above discussion has been more or less along the lines of a modification of the assembly <u>process</u> as applied to the existing SHARE language, the work involved would be of such magnitude as to require a careful scrutiny of language of the system as a whole. Some thoughts on this will follow.

The admitted purpose of an assembly-compiler-what-have-you system is to overcome some of the deficiencies inherent in the modern high-speed digital computer. The assembly programs of the past have adequately handled many of these, but a large area of difficulty--namely that associated with the debugging process, for example--has been completely neglected. Let us review what seem to be some of these unattacked deficiencies.

Early in our experience in coding for a stored program computer, we find that there are many sequences of instructions which are repeated over and over again throughout the course of a problem, and indeed, are common to many different problems. At this point we look for a means of using the code that we wrote yesterday to do the same job today, and in the process come up with the idea of constructing a closed subroutine. By closed is meant the property which the routine has which allows us to transfer to it from various parts of our own code and have it do our job for us, after which the sequence of control comes back to our code at the proper point.

It is easy to fall into the trap of thinking that this is the only advantage of the closed subroutine. If this were the case, the efficiency of many of the problems now being run could be improved markedly by actually incorporating the subroutine as an open routine at the point at which it was called for, eliminate the bookwork associated with the "calling sequence decipherment", and go on doing so till we ran out of memory. (This is of course based on the assumption that there is some memory space left over when the job is done, and is probably truthful a good part of the time.) A more important feature which is a property of the closed subroutine lies in the fact that we use it to process varying pieces of data in some standard way, and go on to give the subroutine a name by which we can easily refer to a rather complex process. Originally the device by which this was accomplished was a programming "trick" (xxx Reset add xxx), but, in the 704, at least, this is built into the hardware (TSX YYY,4). Once we have debugged the subroutine (which may be none too easy) we can more or less forget about the means by which it does its job, and only have to make sure that we have supplied it with the right data and used the output properly. (This is a big help in debugging the program which uses the subroutine, as is soon discovered by the neophyte programmer.) Conversely, if at some later stage in the evolution of our final code we discover an error in the subroutine, we can go ahead and change it knowing that so long as the task the routine does is unchanged (that is, what we thought it was supposed to do--not what it actually ended up doing) we will not have to worry about changing any other parts of the code. This in itself should be enough to make us look more closely at the subroutine concept as a device for doing not only all of our standard functions, but all of the computing which we have to do.

Unfortunately, there is a large amount of inertia on the part of the coder which keeps him from constructing his entire problem on the above scheme, and this will continue to be the case until the system in which he writes his code allows him to do this with little, if any, additional effort.

If it is done, the process of coding boils down, in the main, to assembling the data for a process, jumping to the routine which does the process, and getting rid of the data produced--which probably means sending it on as input to another routine. The code for the process itself is one or two orders which make up a calling sequence. Of course it will eventually be necessary to code the process routines themselves, but if at any time a task looks too difficult to code, or if it will interrupt our train of thought to code it as it arises, then it can again be relegated to a subroutine. By the time we get down to the bottom "level" of the process, the routines probably are so simple that the individual time spent coding them will be small. There might be an awful lot of them to code, but past experience seems to dictate that the time spent in checking out a long sequence of orders goes up far faster than at a linear rate, so that in the long run there may be a tremendous gain in overall time spent. In fact, the debugging process can be explained fairly simply if the code is constructed in this fashion. First, put in the data for the code, and see if the right answers come out. No? Check the data into the first subroutine and the answers it gives out. Still no go? Check the data into each of its subroutines in turn, and continue the process until we get down to the lowest level, at which point there will presumably be a simple set of orders to debug. (The temptation for the coder to "cheat" and do something rather complex just to relieve the monotony will be great, but should be resisted at all costs.) Checkout, if not a "snap", at least is orderly and logical.

In the above discussion fundamental assumptions were made. First, that each routine is self contained and does not clobber any other portion of the code, and that, in fact, it can be written without any knowledge of the details of the routines either above it or below it in the hierarchy. This is where present-day machines fall down, since once inside the machine one routine cannot be told from another, and instructions of the "clobbering" type are executed along with all the rest. It may be possible to construct hardware that isn't restrictive as to length of program, etc., to do this, but until it is done we have to rely on our assembly-compiler to do it for us.

How are references made within such a routine? There are only a few types of these necessary, as follows. One, references to other instructions within the routine, as for example transfers. Two, references to the input data. Three, references to the output data. Four, references to constant data which is probably carried along with the subroutine. (If it isn't there right with the routine, chances are it will be forgotten at some time or another.) Five, the references to the common or working storage that the routine requires. Finally, the references to the other routines which it uses. (Only references to the routines themselves, and not to the input and output of these is meant, as these latter two classes show up as requirements of the lower routine.)

Since we don't like to use machine language locations or addresses (when we're coding we don't know them and when the problem is running we couldn't care less, except for possible machine difficulties. Let's not discuss debugging right now), we use symbols to represent the addresses of the instructions to which we are referring. We should be able to use any symbol within the allowed set for this purpose, and with no knowledge of the symbols used by either higher or lower level routines. That is, what is desired is a class of symbols which carry different meanings from routine to routine. The present 704 assembly routine makes this sort of thing fairly difficult at best (the coders who can use the HED operation with facility on several levels seem to be few and far between.) Any number of devices might be used to accomplish this end, but whichever is finally employed should be an automatic function of the assembler or of writing down the code, and should not rely on separate indications which can be forgotten.

Also needed in the way of symbols for orders are the symbols which stand for the whole group of instructions within a code. These again should be separately distinguishable from other symbols, and this process should be automatic.

(This one isn't too difficult--if it shows up as the address of a TSX order, it is one of these. Let's not get to a routine without going through a TSX).

The symbols used for the transmission of data must again be separate from either of the above classes, and in addition, must have some sort of "relative" property associated with them; that is, a symbol for a block of data is used with another symbol or number to identify the particular piece of data within the block. Just how the data itself is transmitted from block to block may be accomplished in various fashions, and all of them should probably be allowed. First, we can send the data itself, by means of some standard cell within the machine. The Accummulator is most often used for this purpose, but some schemes require standard locations such as 0000 or some specified part of the temporary storage. Alternatively, we can transmit the location at which the data can be found, and if there is lots of it, how much there is there. This can be carried to extreme by transmitting, for example, L(L(L(L(L(X))))), but confusion quickly sets in. In the process of transmitting data from one level routine to the next, we must make sure that we don't affect any of the data sent to or from it by another level routine. Right at this point we come to one of the objections to using the sort of scheme which we have been describing. A large amount of time, it is argued, is spent shuffling data around, and finding out what we want to do, rather than actually doing it. The situation really isn't as bad as it seems, however, as the machine is actually spending most of its time executing the bottom level of subroutines, and at this level it is not the case that most of the work is shuffling data. It seems fairly obvious that a five or ten percent reduction in efficiency is more than compensated for by the gains made by knowing what is going on.

When we get to the problem of the working (temporary) storage required by the routines at various levels, we find a great disparity of prejudices. Some systems put all the intermediate working storage in one great big pit, to which the name "COMMON" is given. This has the advantage of reducing the storage required, but many has been the times that something thrown into COMMON just wasn't there when it was supposed to be, and the process of finding the offending instruction, which can be at any of numerous levels, can be tedious at best. Other schemes use one level of working storage for the lowest level of code, and another level of working storage for all the other routines. This helps the situation some, but if the number of levels is large, trouble can still arise in a hurry. Perhaps the best scheme would be to have each routine have its own private store of working storage, and, as with the references made to instructions within a routine, the symbol used for this working storage should be interpreted separately from routine to routine.

All through the past few pages the trend has been to ignore two of the basic tenets of "Good Coding" which have been adhered to in the past; i.e., that the code should be as short as possible and as efficient as possible. This was probably a fairly good idea when the machines were relatively slow and storage was limited, but as problems grow in complexity and machines in capacity and speed it is rapidly becoming outmoded. It is even possible to get a good argument going as to what extent these properties should be included in the basic subroutines that are going to be used again and again. Perhaps someday someone will do a study to compare the coding and checkout costs of eliminating one instruction or a few microseconds from a square root routine against the actual cost of the machine time saved. The results would probably be enlightening.

While the actual details of an assembly-compiler notation and format have been completely put aside, the system as a whole should allow the routine-subroutine concept described above to be used in a problem with great facility. For those die-hards who insist on large complicated codes which are not broken down into manageable sections, it would be certainly possible to treat the whole thing as one big subroutine, and even go as far as to code the whole thing in actual octal--or even binary. The fact that a system permits sophistication need not restrict it in any way.

The routine-subroutine concept, if <u>recognized</u> (and not merely handled) by the assembler, allows a great number of debugging aids to be incorporated easily and automatically at any time during the checkout process. The various ramifications of this have not been explored to a great extent, but the fact that information <u>about</u> the code is carried along at all times with the code itself (which has been assumed to be in symbolic form right up to the time it is loaded in for execution) permits us to carry on the debugging process at a much higher level than ever before possible with non-interpretive schemes.

While the foregoing remarks have purposely been directed away from the idea of a restrictive system, a purely personal belief is that such a system should indeed be rather restrictive. It should not, however, be impossible to "beat the system"; but certainly the path of least resistance for the coder to take in preparing his problem should be along the lines outlined above. Perhaps the ingenuity and inventive talents of the imaginative coders would be applied to the tasks of fitting some seemingly "impossible" functions into the system as it stands.

An assembly system, such as the one proposed above, should have some means on incorporating "standard" subroutines into our problems with a minimum of effort on the part of the coder.

(What we call a standard subroutine is to a large extent, at present, dependent upon the secondary storage capabilities of the machine, and the amount of paper one must look through to find the specifications for the desired routine. The main objection raised because of the secondary storage access time; a tape unit probably can hold at least as many routines as have been generated by SHARE, but the access time to get to these may be high. The question of the trouble it takes to find the write-up for the routine in question can be solved by the use of standard indexing schemes, or the like. This may, in fact, require the full time services on the part of a "librarian", and some installations have just this sort of person to take care of the paperwork.) Regardless of the size of our subroutine library, however, there are three methods currently in use for incorporating these into the problem. The first method is just to pull these from a file of symbolic cards and read them along with our own symbolics. The second method has the assembler incorporate them from a library tape and produce them in binary along with the rest of the code. The third scheme, in its most refined form (which has never completely been done) would have the assembly (or the coder) insert a small routine which would pull these in just prior to the actual execution of the code as a whole. (A fourth method comes to mind, which, while not being efficient (probably), will be mentioned. Assume an interpretive mode of operation in which a subroutine is placed in memory the first time it is actually called for--the code may have been running for some time. Thereafter it is assumed to be in memory. Other subroutines are called in as needed, but the first time only. When space runs out, one of the least used subroutines would be wiped out in deference to the newly desired routine. The scheme could undoubtedly be applied to a non-interpretive mode also, but it would be easy to conceive of times when the program would spend all of its time shuffling subroutines in and out.)

The first-mentioned method will probably always be with us, as there are a great number of routines which would be used so seldom that it would scarcely be worthwhile to carry them around in the system. Also, new routines would be put into a program in this way at least as long as their originator isn't confident enough of their correct operation to commit them to the "black box." But it doesn't seem to require any effort on our part to have the assembler do this for us. (This, by the way, is one of the weakest points in our present-day compilers such as PACT and FORTRAN, since in this instance we are dealing, while coding, with two mutually incompatible languages.)

The main difference between the second and third method is twofold. First, we may be unwilling to read in the sub-routine from some secondary storage device (which is then lost, presumably, to the coder) because of the time involved. Secondly, we are confronted with the matter of changes to subroutines. This is really a pertinent point to consider, as almost all "checked out" subroutines develop bugs at some stage in their career. If we incorporate them at assembly time and hence make them a part of our program which will not be changed when the library is changed, we are reasonably assured that we know what our problem is doing. However, we may have to go through all of our active or potentially active programs to put in a corrected version when one shows up. It would be hard to say how much the time lost here is compensated for by other considerations. Also, it causes no problem at all when we remove a routine from the library, if we are careful about telling our coders.

If, on the other hand, we incorporate the subroutines into all programs at loading time, any change made in a library routine will automatically show up in its corrected version in all our programs. But, unless we have designed around it, any lengthening of the routine may give us trouble, and if the new calling sequence does not include the old one as recognizable, we again face the difficulty of having to change all of our programs. The space situation isn't too difficult unless we are beginning to run out of memory; the routine which reads them could take care of the problem of fixing up the cross references. The calling sequence question is just a rephrasal of the old problem of "Just when is a new subroutine really new and when is it a modification?" It may be that with careful attention to the problem of a general format for acalling sequence format that this wouldn't be as bad as it first seems. Finally, we would have to be careful about removing a routine from the library. This might cause the library to get out of hand rapidly.

In the assembler-loader described above, it is hard to determine which we should do. Either one is possible, though the second method first comes to mind. If we wished to incorporate the subroutines "once and for all", this could be done during the initial pass to convert out symbolics to the ALPHA deck, putting in the subroutines in symbolic form. If there really is a dilemma, both methods could show up as options, leaving it up to the coder or individual installation to use his own discretion.

Another question not altogether alien to the first concerns the convention of references to standard subroutines by other subroutines. Here we can either disallow it completely, make references to other subroutines through the calling sequence, or allow complete freedom of references. (In the latter case, of course, the assembly would take care not to put the same subroutine into our program twice. Here too, somebody can louse up the situation by making modifications to the subroutine at execution time, presuming he knows what it looks like on the inside. The same questions arise here with regard to changes and deletions of subroutines referred to by others, and one decision will affect the other one too.

One last word about subroutines concerns the method of incorporating them into the code. Few of the presently used assemblers automatically insert them for us. In SAP, for example, the LIB operation must be inserted in the code, and what is worse, it must be inserted at a point remote from the point in coding at which we TSX to the subroutine. Most compilers, on the other hand, recognize that when a subroutine is transferred to it had better be somewhere in the machine, and the compiler takes appropriate steps to see that this is done. This is really such a trivial task for the assembly program that it should always be a feature, if only an optional one.

Completely aside from the subroutine question, this last point brings to light a principle which should always be followed if we want to cut down errors in coding. This is: There should be as little remote indication of what we want to do as is possible. Numerous examples of this are scattered in all our codes. The LIB operation is one; the LXD situation is another (on the 709 this has been taken care of by the AXT instruction). Any effort which we spend to eliminate the necessity of our referring to some other instruction, to another piece of paper, to eliminate the need for remembering to do something later on, is all time well spent, and should be a guiding principle in constructing any system of coding for a digital computer.

C. L. Baker
2 February 1957