CGTM 54, August 29, 1968
John R. Ehrman

REPORT of the SHARE-GUIDE ASSEMBLER PROJECT

This report is a summary of ideas and suggestions for correcting and extending the current System/360 Assembler Language, with the intention of encouraging IBM to undertake the development of a large, high-speed, and powerful macro-assembler embodying as many of these ideas as possible.

The material is organized in rather crude fashion: while some of the suggestions are listed in one area, it will be seen that they apply in many others. The report should therefore be read not as a coherent outline for the organization of a macro-assembler, but simply as a collection of facilities that we think should be considered in eventual discussions of implementation. No assumptions are made concerning feasibility; some of the ideas would be trivial to implement, while others are perhaps best characterized as "blue sky".

While the material in this report is due to many people, I take full responsibility for any confusion that may arise from poor presentation. Any suggestions for additions or corrections should be addressed to me at the address below.

John R. Ehrman
Computation Group
Stanford Linear Accelerator Center
Stanford, California 94305
Phone: (415) 854-3300, x307

CGTM 54, August 29, 1968
John R. Ehrman

REPORT of the SHARE-GUIDE ASSEMBLER PROJECT

This report is a summary of ideas and suggestions for correcting and extending the current System/360 Assembler Language, with the intention of encouraging IBM to undertake the development of a large, high-speed, and powerful macro-assembler embodying as many of these ideas as possible.

The material is organized in rather crude fashion: while some of the suggestions are listed in one area, it will be seen that they apply in many others. The report should therefore be read not as a coherent outline for the organization of a macro-assembler, but simply as a collection of facilities that we think should be considered in eventual discussions of implementation. No assumptions are made concerning feasibility; some of the ideas would be trivial to implement, while others are perhaps best characterized as "blue sky".

While the material in this report is due to many people, I take full responsibility for any confusion that may arise from poor presentation. Any suggestions for additions or corrections should be addressed to me at the address below.

John R. Ehrman
Computation Group
Stanford Linear Accelerator Center
Stanford, California 94305
Phone: (415) 854-3300, x307

Contents

A.1.1. It should be possible to determine the value of all attributes of a given symbol (such as type, length, relocatability, and value), and whether each has been defined yet.

A.1.2. It should be possible to override any Assembler-defined attribute of a defined symbol by explicit declaration.

A.1.3. The programmer should be able to declare additional programmer-defined attributes for any symbol by explicit declaration, and then be able to assign and manipulate those attributes.

A.1.4. It should be possible to determine whether a symbol has been multiply defined; which of the attributes in multiple definitions are in conflict; and to specify that one or another of the various current attributes are to be used as the correct ones for the symbol in question.

A.1.5. All the standard attributes of a symbol should be available to the TESTRAN interpreter under the TEST option.

A.1.6. A part of the relocatability attribute of a symbol should be the name and type of the CSECT or DSECT within which the symbol is defined. Otherwise, given the relocatibility attribute, one should be able to determine the corresponding section type and name.

A.1.7. It would be useful to have an "alignment" attribute for a symbol.

A.1.8. If named location counters are used, it should be possible to determine the name of the location counter under which a symbol is defined.

A.1.9. The rules for the use of attributes are somewhat inconsistent at present. For example, if one writes "A EQU 5" then the length attribute of A is found to be 1; this implies that the length attribute of a self-defining term should be 1 also. If however one has the misfortune to write or generate a term like "L'5" in an expression, the statement is treated as erroneous. Furthermore, since the length attribute of something is arithmetically equivalent to a decimal self-defining term, there should be no objection to writing a term like "L'L'L'A" where the result would be 1.

A.1.10. There is no good reason why the integer and scale attributes of a symbol should not be available at both macro-expansion time and at statement assembly time; one could generate code for scaled fixed-point arithmetic using these attributes in the same way as is now done with the length attribute in character-moving and decimal instructions.

A.2.1. Since the length attribute of an expression is defined to be that of the leftmost term, the same should apply at macro-scan time, so that a macro argument &A with value "3+X" will not generate an error when something like "L'&A" is coded.

A.2.2. There is no special reason why the attributes of the symbol X in the statement "X DC (3-2)F'5' " should be undefined at macro time.

A.2.3. It would be useful to be able to find out what would be the attributes of a symbol defining a constamt that was generated from a given character string. For example, suppose the value of the variable symbol &X is "AB&&CD", and we want to know what would happen if we used it to generate a character-type constant: that is, what would be the attributes of the symbol XX in the constant definition "XX DC C'&X' ". It can be seen that the count attribute of &X is 6, whereas the length attribute of XX would be 5.

A.2.4. The allowable range of the type attribute is rather restricted, since one cannot determine many of the characteristics of macro operands that are quite useful in conditional assembly. For example, it would be useful to be able to determine that an operand is a valid symbol, an expression, a quoted string, a macro name, and so on. This would require the extension of the type attribute to more than a single letter; a decimal value is suggested. Similarly, one might wish to make use of some of the properties of an operand treated simply as a character string: that is, to ask whether a given character is a member of a given set of characters, or more specifically, whether it is a letter, digit, or operator.

B.1.1. It would be useful to be able to generate sequence symbols dynamically: for example, the statement "AGO .A&B" could be used to speed the process of branching through a multi-directional switch.

Similarly, it should be possible to define and construct sequence symbols by concatenation when they are encountered during macro definition.

B.1.2. It would be useful to expand the available macro-time branches to include the following:

1. AIF (boolean-expression)true-exit,false-exit
2. AIF (arithmetic-expression)neg-exit,zero-exit,pos-exit
3. AGO (exit-1,exit-2,...,exit-n)seta-expression

where the derivations from the similar Fortran statements are obvious.

B.1.3. A useful statement for controlling loops and counters during conditional assembly would be

ADO sequence-symbol,set-symbol,from-expr,to-expr,increment-expr

where the Fortran DO statement parentage is clear. Such statements should be nestable.

B.1.4. It should be possible to define an iteration control which is similar to the WHILE and UNTIL constructs of some ALGOL languages.

B.1.5. The IRP pseudo-op considerable simplifies the coding of meaningful macros. It would also help to be able to define a variable symbol to be used as an argument counter during the expansion of the code controlled by the IRP; allowing the specification of a start value and increment for such symbols would also be helpful. There should of course be a means for the termination of the IRP processing.

C.1.1. Expression evaluation should be consistent: (a) operations on a variable should produce the same result as would be obtained from the same operation on any expression yielding the value of the variable; (b) any operation that applies to certain data types should apply regardless of where the expressions occur.

C.1.2. The present rules for evaluating expressions are in some sense inconsistent between the macro passes and the assembly passes: the value of a SETA expression or variable is a 32-bit two's complement integer, while an assembly-phase expression is a 24-bit integer. There seems to be no good reason to retain this difference in treatments.

C.1.3. The restriction of values of self-defining terms to 24 bits seems arbitrary; but even if there is a good reason in the current implementation, it should be removed in any extensions. For example, it should be possible to write "DC A(X'12345678')" with intelligible results so long as the restrictions implied by the use of an A-type address constant are satisfied. This means that one could write "DC A(X'03000000'+NAME)" rather than having to write "DS 0F" followed by "DC X'03',AL3(NAME)". This will introduce no incompatibilities with the linkage editor.

C.1.4. Unary operators should be allowed.

C.1.5. Logical operators should be allowed in arithmetic expressions, as well as shifts.

C.1.6. If an intermediate expression has a value attribute of 0 or 1 and a relocatability attribute of 0, then it should be allowed to be combined with relocatable terms through the use of the * and / operators.

C.1.7. Anywhere a self-defining term is normally required, it should be possible to write an expression whose value will be used (if it is not relocatable). Thus one should be able to write "A EQU 3" followed by "SPACE A".

C.1.8. The rules for the resolution of expressions into base-displacement form are carefully laid out in the Assembler Language manual, so that it is

clear in writing something like "L 9,8(3)" that the Assembler must resolve the expression (in normal circumstances) with a base of 0, a displacement of 8, and an index of 3. On smaller models of the System/360, this common programming error can lead to reduced performance because the user must pay for an extra indexing cycle for each RX instruction written in this careless fashion. It should be possible to require that if (1) there is no USING in effect that specifies an absolute expression, and (2) the basc digit of the addressing expression is zero, and (3) an implied address is used, then the index digit should replace the base digit.

C.1.9. There is currently no means for specifying the base in which a conversion from one type of symbol to another is to take place. For example, one might want to know the value to be associated with a given character, so that one could write "&A SETX ('8',10)" and have the byte value '8' converted to the decimal value 248. Another example: "&B SETX ('A',16)" would yield the value C1 for &B. This would greatly simplify the generation of constants at macro time.

D.1.1. It should be possible to use a macro to define other macros.
It should be possible to define a macro anywhere before it is used.
It should be possible to redefine macro definitions.

D.1.2. There should be some means to determine the scope of a macro definition. There should be various forms of redefinition, which allow pushing the current definition onto a list, popping previous definitions from the list, and using a definition from an arbitrary position in the middle of the list. It should be possible to redefine or update any definition in the list, as well as retain definitions that have been popped.

D.1.3. There should be a facility for function-type macros accessible from any field in an instruction or macro call.

D.1.4. It should be possible to construct a comment field in a generated statement by concatenation.

D.2.1. The instruction set to be used during an assembly should be under the control of the programmer, so that portions of the table of operations can be deleted, replaced, or overridden as desired. That is, the entire set of operation codes should be organized in list form, so that new members can be added to, and old members taken from, the table or portion of the table of interest.

D.2.2. It should be possible to make use of many of the assembly-phase operations such as base-displacement resolution, length attribute determination, etc. for arbitrary operands, not just for those used in certain classes of machine instructions.

E.1.1. The current restrictions on the macro-time uses of attributes to quantities in the argument list undoubtedly derive from the necessity of limiting the size of the local dictionaries for each macro. There should be no need for this if the programmer is willing to pay the price for having the full symbol table available when he wants it; this would allow the determination of attributes of generated symbols, which is not possible at present. This would also mean that there would be no need to distinguish between inner and outer macros.

E.1.2. The types of list structure available for macro operands should be expanded to include arbitrary lengths and nestings.

E.1.3. It should be possible to build arbitrary-length argument lists by concatenation. At present, the comma delimiter structure is determined before substitution of variable symbols, which might themselves contain commas. This would allow the insertion of sublists into the middle of an argument list.

E.1.4. If it is possible to write the operands of a macro-instruction as arbitrary lists, and make provision for their treatment internally, then it would be possible to handle through macro definitions the processing of the normal instruction set, since constructions such as "3(7,REGB)" could be analyzed very simply. It is suggested that the expression preceding the first left parenthesis be accessed by a notation such as &A(0) or &SYSLIST(n,0), where the normal sublist notation would apply for the other terms in the operand. This facility would provide the equivalent of an "OPVFD" instruction.

E.1.5. It would be useful to treat the input parameters of a macro-instruction as though they were locally-defined character variables with initial values defined by the call. That is, one should be able to write

```
        MACRO
        ANY   &X
        ...
&X      SETC  something
```

without having to break down the input string into little pieces contained in a bunch of local variables.

E.1.6. Substring notation can presently be used only in SETC statements; it should be allowed anywhere that a normal character expression is allowed. This will simplify the treatment of macro arguments which are treated as

character strings.

E.1.7. The count attribute of a macro-instruction operand or SETC variable should be useable anywhere that a self-defining term may appear, rather than merely in SETA expressions or relations.

E.1.8. It would be useful to be able to pass a set symbol or list as a macro operand (in the sense of call-by-name), rather than pass the value of the operand. This would allow the changing of the value of an item from an outer macro by operations in an inner macro.

E.1.9. There seems to be no good reason why keyword operands should not be allowed in the name field of a macro-instruction.

F.1.1. There should be some means of pushing variable symbols onto stacks or lists, as well as a complete set of operations shich allow manipulation of the elements of the appropriate list in use.

F.1.2. It should be possible to define a set of variable symbols which have the characteristics of a bit string, so the individual bits can be manipulated by use of a set of bit-string operations such as is provided by PL/1.

F.1.3. There seems to be no good reason to restrict the size of a SETC variable to 8 characters; 255 might be a more natural and useful maximum.

F.1.4. The dimension of a variable symbol should be allowed to be an expression, which is evaluated when the symbol is declared.

F.1.5. It should be possible to construct the name of a local or global variable symbol through concatenation. This also implies that it should be possible to have "executable" macro statements appearing before all variable symbols have been declared.

F.2.1. There should be System Variable symbols as follows:

&SYSDATE current date

&SYSTIME current time

F.2.2. It would be useful to know whether &SYSECT is in a CSECT or DSECT.

F.2.3. It would be useful to have system variable symbols that specified the settings of various quantities such as print switches, ICTL settings, ISEQ settings, current USING information, and so forth.

F.2.4. It should be possible to use &SYSLIST even with keyword macros; simply skip over keyword operands.

&SYSLIST(0) should refer to the name-field symbol.

The use of &SYSLIST without a subscript should refer to the entire operand list (excepting keyword operands) strung together. This would make it possible for example to transfer an entire argument list to an inner macro.

F.2.5. There should be some menas of specifying a return code from a macro expansion. This would allow a macro coder to control the expansion of outer macros from the results of the expansion of the inner ones. It is suggested that there be a system variable symbol &SYSCODE associated with each macro expansion (in much the same way as &SYSNDX) which is a programmer-defined return code; the code could be defined, for example, as the value of the operand of an MEXIT instruction. The value returned could be formed by ORing the new value to the current value of &SYSCODE, thus allowing multiple return indications from various levels.

F.2.6. There is at present no way to communicate with the inside of an assembly except through the medium of source statements, such as copy code or macro definitions. It would be most helpful to have a system variable symbol &SYSPARM which is set to the value of an appropriate parameter from the PARM field of the invocation of the assembler. Thus, one could write a number of debugging statements into a program, and when an assembly is wanted without them the part of the PARM field that specifies the value of &SYSPARM could be changed, and the assembly would then skip the debugging statements.

G.1.1. Anything at all should be allowed inside a DSECT for purposes of symbol definition. Thus, ENTRY and EXTRN in a DSECT should simply be ignored except to define the symbols.

G.1.2. It is awkward not to be able to define DSECTs within existing DSECTs without having to use different relocatability attributes. For example, one might want to describe a structured area within a larger structured area, but there will be only a single base register available to point to the larger area. (The Fortran H "STRUCTURE" statement provides such a facility.)

G.2.1. There should be some means to qualify symbols so that the programmer can specify a structure for his program more detailed than is allowed by the current System/360 CSECT structure.

G.2.2. If qualified symbols are allowed, it should be possible to determine all active qualifiers at the point of symbol definition.

G.2.3. It should be possible to push and pop segments of the symbol table. This would greatly simplify the coding of block-structured material, and would provide a simple means of symbol qualification.

G.3.1. It should be possible to determine the length of a CSECT or DSECT, both at assembly time and at link-edit time.

It should be possible to determine the length of any number of virtual dummy sontrol sections, as can now be done for a single one through the use of the CXD instruction.

G.3.2. It is presently possible to structure only a single virtual dummy control section through the use of the DXD instruction and the Q-type address constant. It would be helpful if it were possible to define for link-editor resolution several different and distinct virtual dummy areas, in much the same way as the DSECT instruction can be used to define many virtual areas whose values are resolved (for base-displacement purposes) at assembly time.

G.3.3. It should be possible to specify addends for the operands of Q-type address constants. At present, one must either name all the items in the virtual area unless they are in a DSECT; it should be possible to prescribe one's own levels of structure by writing something like Q(A+4), so as to obtain something between the DSECT technique and the current DXD provisions.

G.4.1. A major omission in the present assemblers is any facility for specifying remote assembly of code or other statements. The "RMT" pseudo-operation of past assemblers was a crude method for performing such remote assembly; it should now be possible to specify several ways in which text being saved for later processing is to be retrieved. That is, one should be able to specify that blocks of statements are to be placed either in named queues or stacks (FIFO or LIFO lists) for later retrieval.

G.4.2. It should be possible to have many named location counters within a given CSECT or DSECT. A system variable symbol such as &SYSLOC could then be used to determine the name of the location counter currently in use.

G.4.3. If multiple location counters are available, the operand of a CSECT or DSECT instruction could be taken as the setting of the pertinent location counter.

G.4.4. There should be some way to generate named COMMONs.

H.1.1. The system interface should be cleaned up considerably, so that it is possible to change default parameters at Sysgen time, for example. This would also allow the use of standard DD names.

H.1.2. There is no good reason that the END card should be logically equivalent to an end-of-data-set signal on SYSIN. The Assembler should re-initialize itself for another assembly, much as is done by the Fortran compilers and the PL/1 compiler with the "Process" statement. This will eliminate the need for separate job steps for each assembly.

H.1.3. It should be possible not only to specify that a batch of assemblies is to be done, but that a given set of items such as macro definitions, DSECTS, EQUs, and declarations be considered global to all the assemblies of a given job step, so that they are retained throughout.

H.1.4. There should be some types of constants similar to the CXD whereby the linkage editor could fill in the length of control sections or common areas.

H.1.5. It should be possible to have the linkage editor combine control sections in a variety of ways, so that they are concatenated, chained, ANDed, ORed, and so forth. This would simplify the coding necessary to initialize tables and data areas, particularly if the tables are coded in portions of several assemblies.

J.1.1. It would help to be able to say "DROP ALL" to get rid of all currently active base registers, so that one won't forget to drop one in a multiple-CSECT assembly, nor drop one not in use with an accompanying diagnostic.

J.1.2. It is an inconvenience in the current language that each specification of a register in a USING statement overrides the previous definition of the value to be assumed in the register for resolution purposes. There should be some means to save and restore the contents of the USING table both for individual registers and for the table as a whole, for an arbitrary number of levels of saving.

J.1.3. It should be possible to determine all currently active base registers with a fixed relationship to a given expression (that is, so that it makes sense to calculate the difference of the given expression and that in the USING list).

K.1.1. The PRINT conditions should be save-able and restore-able from a pushdown, so that they can be changed during a macro expansion and then put back to the original setting.

K.1.2. Allow from 1 to 8 characters for the object module ID, rather than precisely 4 as at present. If the numeric field overflows, just restart at zero.

K.1.3. It would be useful to be able to obtain cross-reference listings of the following:
(a) macro names used; (b) registers used; (c) instructions used;
(d) pseudo-operations used; (e) literals.

K.1.4. There should be some means to control the collection of symbol table and cross-reference information. For example, if there are many standard DSECTs at the start of a program, but only a few of the synbols defined in them are actually used in the following program, it might be useful to the programmer to be able to turn off the collections of cross-reference information for the duration of the definitions.

K.1.5. One should be able to control all parts of the assembly listing. In particular, there should be a much broader choice of listing and debugging facilities available for macro-programming, such as traces of all conditional assembly instrctions executed, macro level numbers, inner macro parameters, and so forth.

K.1.6. Comments should be allowed on CSECT and DSECT statements.

K.1.7. There should be some means of specifying that the first line (or selected lines)of generated code and the location counter should be printed in macro-expansions.

K.1.8. It is a nuisance not to be able to generate attractive output from a macro expansion by outputting an operand at a time on separate lines with continuation characters at the end of the line. That is, there is no way to generate continuation characters under programmer control; they are "meta-characters" in the current assembler language.

L.1.1. There is no need to limit the use of literals to single operands that can appear in DC statements. For example, one might like to write "L 2,=(AL1(3),AL3(NAME))" or "EX 5,=(MVC 0(0,2),0(3))" with the programmer being responsible for addressability problems.

L.1.2. It would be useful to be able to specify an alignment for literals.

L.1.3. It should be possible to write literals in address constants, such as A(=F'4') or =A(=F'4') or even =A(=A(=F'4')).

L.1.4. It should be possible to specify multiple constants separates by commas in B- and X-type constants.

L.1.5. It should be possible to allow scale factors in individual constants in exactly the same manner as is done for exponent modifiers and internal decimal exponents.

L.1.6. It would be useful to define another level of structuring for constant definitions, as suggested by the following:

```
ARRAY   DC   (2,25,6)E'0'
```

which defines an array of 2 rows, 25 columns, and 6 planes, and 300 elements in all. One could then inquire through the use of an attribute notation as to the number of individual dimensions, their extents, and so on, so as to generate indexing code fo array manipulation at assembly time without having to recalculate the indexing parameters either dynamically or by hand before reassembling.

In this scheme, the dimensionality information could be kept in a list in such a way that the total replication factor is the product of all the list elements, and the present use of the duplication factor is represented by a list with a single element.

M.1.1. The pseudo-operations OPD and OPSYN are invaluable.

M.1.2. The table of extended mnemonics for the branch instructions should be extended to cover the RR forms of the BC instruction.

M.1.3. Forward references should be allowed in EQU statements.

M.1.4. It should be possible to expand the forms of allowable expressions permitted in the operand field of an EQU statement to include such things as "symbol EQU expr (expr)" or "symbol EQU expr(expr,expr)".

M.1.5. It would be useful to be able to selectively choose which of the attributes of the given operand are to be assigned to the symbol, so that one could pre-define some of the attributes of a symbol and leave the others (such as value) until later.

M.1.6. The restrictions on the use of ICTL should be loosened.

M.1.7. There should be a NULL pseudo-op which is a means of attaching a label to the next bit of code (not a macro call but maybe the first byte of the macro expansion when it expands into code) in the same section. Attributes are to come from the next item. In case the NULL is last in the section, the value should be the byte beyond the end of the code in the section, and have default length 1 and undefined type.

J.1.8. Name-field symbols should be allowed on CNOP instructions whether or not NOPR instructions are generated. The name should be assigned the desired value, and the length and type attributes should be 2 and I respectively.

M.1.9. There doesn't seem to be any compelling reason to treat statements such as "L 8,2(0,3)" as implying alignment errors; if the programmer has specified nothing about the uses of the registers, the Assembler should make no assumptions about what will be in them.

M.1.10. The programmer should be able to modify the severity codes assigned to error conditions.