Integrated system design leads to the inclusion of certain features in the assembly language for the convenience of compilers, and others for the convenience of program segmentation.

This paper discusses motivation for the inclusion of these features, and traces their influence upon the internal structure of the assembly program.

Design of an integrated programming and operating system

Part II: The assembly program and its language

by R. B. Talmadge

modular design of programs

In the past few years, programming methods in general use have tended to emphasize modularity of program design; not only for large, complex programs, but even for those of relatively modest size. A common method for achieving modularity has been to divide the program into (relocatable) segments, because:

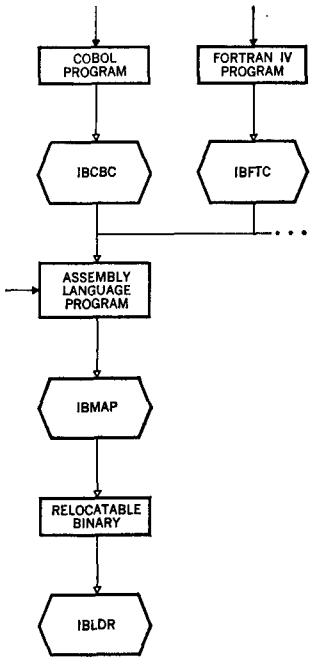
- Coding and checkout of the individual segments can proceed in parallel, thereby considerably reducing the time between analysis of the problem and availability of a running program.
- Modifications to the program, which are usually confined to a few segments, can be accomplished without the necessity of reassembling the entire program.
- Duplication of effort can be minimized by taking advantage of libraries of previously checked-out subroutines.
- Different programming techniques can be used for the various segments, thus allowing the exploitation of the strong points of a particular language or processor.

However, it has generally not been possible to realize all these advantages. Programs created by FORTRAN II and COMMERCIAL TRANSLATOR, for example, are so incompatible in deck format,

interprogram reference facilities, and loading technique that a combination of the two is operationally impractical. Even compatible programs, such as are produced by FORTRAN II and FAP, have had rather limited reference facilities. Moreover, methods have not been available to treat programs as independent during checkout, and dependent during normal execution, without requiring substantive changes.

To overcome these difficulties, the 7090/94 IBJOB processor has assumed program segmentation as a fundamental operating principle. Basic compatability is obtained by using IBMAP as the common assembler, so that the source language origin of any segment is indistinguishable to the loader (Figure 1). Consesequently, the assembler receives a substantial part of its input from non-human programmers; that is, from the IBFTC and IBCBC compilers. Its output is binary-symbolic information which is to be loaded by IBLDR, employing rather sophisticated relocation and reference techniques. The assembly language, and the structure of the assembler itself, have been decidedly influenced by these circumstances.

Figure 1 Common assembly



The first part of this paper is a discussion of some of the more distinctive features of the assembly language. Motivation for their inclusion is covered, and a few examples of usage are given. The second part is an examination of the influence of these features on the mechanization for the assembler.

content of paper

In addition, there are two appendices. The first supplies some details of the assembly language, in order to help the reader unfamiliar with IBMAP understand the examples. And, for the reader interested in technique, the second appendix exhibits some details of the mechanization by means of an example.

The assembly language

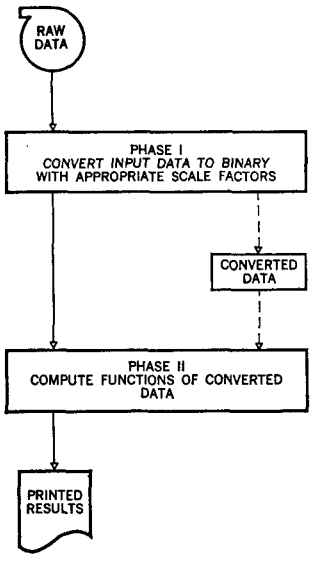
control sections

Designation of a program, or part of a program, as a "control section" permits the loader to identify the section with one in another program, giving the effect of independent relocatability to the section, and providing the mechanism by which references are made between separately assembled segments of the same program.

By definition, a *control section* consists of any combination of instructions or data occupying a contiguous block of memory in the assembled code, which is declared to be a control section by means of the CONTRL or ENTRY operations. (In the latter case, the length of the block is zero.) For such a block, the assembler places an external reference label, chosen by the programmer, into a *control dictionary*, together with information enabling the loader to determine the base and length of the section. All references in the program to symbols within the section are given a special relocation code in terms of this control dictionary entry.

Now, when the program is loaded, either by itself or as a segment of a larger program, the loader examines the control

Figure 2



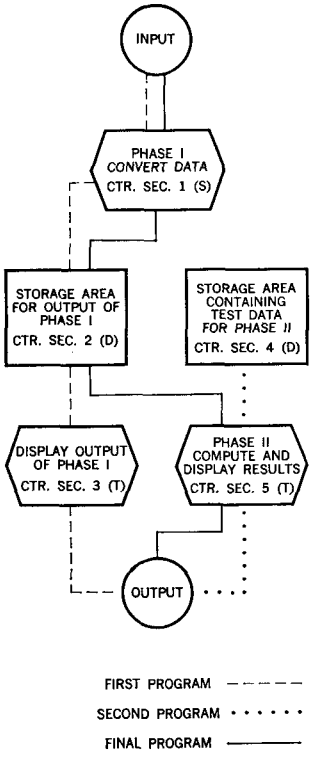
dictionaries of all segments for duplicate external reference labels. (Control cards permit any label to be changed prior to this examination.) If duplicate labels are found, one of the sections, as indicated by the programmer, is selected to appear in the final program, and the others are deleted. Whatever the choice, all references in the body of every segment to a given section are translated by the loader to the final location of the section, while other references are adjusted to account for any deletions.

This technique introduces a uniform relocatable method of handling all interprogram communication. The provision of direct references by the loader is particularly useful because:

- The program may be written without taking into account the final disposition of any control section. In effect, the assembly need only allow for the *possibility* of external reference, without forcing the actuality.[1]
- Data requires no special treatment. The COMMON statements of FORTRAN, for example, translate directly into control sections, irrespective of the number or order of the blocks in any segment.

Furthermore, since sections which are to be identified need not have the same internal structure, and so may perform quite different functions, control sections offer a convenient method of attacking the general problem of parallel coding and checkout. To illustrate, suppose we have a simple program structured as in Figure 2. The program is written in two parts, corresponding to Phase I and Phase II—possibly in different source languages—which are separately assembled.

Figure 3



The first part is written with three control sections:

Section 1 (labelled S), which does the input conversion (Phase I).

Section 2 (labelled D), which is the storage area for the converted data.

Section 3 (labelled T), which causes display of the converted data.

The second part is written with two control sections:

Section 4 (labelled D), within which is assembled test data for Section 5.

Section 5 (labelled T), which does the actual computing (Phase II) and displays results.

Sections 2 and 4 have been given the same label, as have Sections 3 and 5 (recall that in the final program, the loader will accept only one control section with a given label).

The two assemblies may be loaded and checked out as the separate self-contained programs indicated in Figure 3. Section 3 supplies the required references as well as a means for checking the basic Phase I program. Similarly, Section 4 provides references to the output of Phase I and a predetermined input sufficient to check the Phase II computations. When each program is fully operational, simple control-card instructions to the loader will

cause decks to be loaded and combined as a single program, consisting of Sections 1, 2, and 5. And, since direct references are supplied by the loader, not a single instruction need be changed in either assembly.

virtual references

If, as is often the case, it is known that a section of code has been previously assembled, proper referencing can be obtained without including a real control section (that is, an explicitly defined one) in the assembly. Reference to a symbol which is not contained in the body of a program results in the creation of a *virtual* control section by the assembler. For such a section, the loader will expect to find another program with a real control section corresponding to the virtual reference. If no such section is found in any program on the input file, the system library (IBLIB) will be automatically searched for the required reference. If still not found, the symbol cannot be defined for this load, and execution is suppressed.

Thus use of a library subroutine, a routine on the input file, or alternate use of both types, to perform some function presents no extra difficulties. For instance, suppose that Section 3 of the previous example existed on the subroutine library. If the section were left out of the first assembly, and no special instructions given to the loader, execution of the Phase I program by itself would use the library version of Section 3; but execution of the combined program would use Section 5.

qualification

Physical segmentation is not the only method of achieving modularity, nor is it always the most desirable. Often, a better approach is to write the segments separately, but assemble them together. An example of this is a COBOL program in which the segments are to process common data. Because the compiler requires data description in order to generate correct instructions, and because these descriptions are likely to be voluminous, separate compilations might well require a substantial amount of extra processing time, both for the programmer and the compiler.

In order to distinguish between sections of code generated by different programmers, assemblers have, in the past, used various naming conventions. In one method, called *heading*, a single character is prefixed to a symbol to distinguish it from an otherwise identical one elsewhere in the program. The special nature of this device is unsuited to complicated program structure. Consequently, commercial languages have introduced a much more powerful method.

If we consider commercial applications, with their complicated, repetitive, data descriptions, we see that the reflection of the data structure by means of symbols is a powerful mnemonic aid. For example, in a program which processes master and detail records, whose structure (symbolized by indentation) might well be

```
MASTER
  MAN
    NUMBER
    RATE
```

and

```
DETAIL
  MAN
    NUMBER
    RATE,
```

references to employee number in the form

```
MASTER  MAN  NUMBER
DETAIL  MAN  NUMBER
```

prove extremely convenient.

It is clear that this same type of symbol *qualification* (so-called by analogy with adjectival qualification of nouns) is useful in the assembly language. Accordingly, by means of the QUAL and ENDQ operations, any section of code may be declared to be qualified by any symbol; further, the qualifiers may be nested to any depth. It is possible, then, to use meaningfully the symbol

```
MASTER$MAN$NUMBER
```

in IBMAP. (The $ is used to connect qualifiers since blanks are not allowed in the variable field.)

deferred symbol definition

In most assembly programs, the appearance of a symbol in the location field immediately defines the value of the symbol. Expressions used to define symbols at location counter discontinuities, or in terms of other symbols, can consist only of constants and previously defined symbols. Consequently, care must be exercised in the physical sequence of the program statement to insure that:

- Storage allocation and data definition statements are placed so as not to interfere with the flow of the program.
- Dependent statements which arise from distinct parts of the program fall in the correct order.

These restrictions are handled without serious difficulty by a human programmer: storage allocation and symbol definition statements, wherever they arise logically in the program, are merely collected on separate coding sheets, arranged in the proper order, and manually inserted at some convenient point. The equivalent process in a compiler is not so simple. A substantial amount of compiling effort is required to build tables, analyze the storage assignment situation, and save the generated instructions for insertion into the proper place in the program. However, with IBMAP, all restrictions on the order of symbol definition have been removed. And this, coupled with the use of independent location counters to collect data and instructions which are logically sequential but physically remote, permits simpler and faster compiling procedures. In a FORTRAN program, for example, which contained the scattered statements

```
DIMENSION A(10)
EQUIVALENCE (A(5),B)
DIMENSION B(5)
```

in any order, the instructions

```
    USE     A
A   BSS     10
    USE     PREVIOUS
```

at the first dimension statement,

```
    USE     B
B   BSS     5
    USE     PREVIOUS
```

at the second, and

```
    BEGIN   B,A+4
```

at the equivalence statement, insure proper storage allocation without requiring much deliberation by the compiler.

immediate symbol definition

There are several applications for which a choice based on the value of a symbol must be made in the first pass. For example, since any operation is allowed within the scope of a DUP, the duplicate sequence must be expanded immediately in order to determine the actual number of instructions generated. Hence, a symbol which appears in the variable field expression determining the scope of the operation must be assigned some value in the first pass.

In a conventional assembler, this presents no problem, since the function representing this normal location counter or equivalence definition is available. In IBMAP, however, with deferred definition, it is necessary to provide another function. The method adopted was to create a special class of symbols which are assigned an immediate value by a pseudo-operation called SET.

The distinction between symbols defined in SET terms (immediate symbols) and ordinary symbols can be summarized:

Normal definition (DFN)	SET *Definition* (S-*value*)
• Value depends on location counter symbols or equivalences (EQU) involving such symbols and constants.	• Value independent of location counter.
• Is treated as constant, relocatable, or complex according to structure.	• Always a constant.
• Is global, depending upon entire complex of symbol relationships. May not be altered.	• Is local, and may be altered at any time.
• Has no value until after definition pass.	• Has same (local) value in all passes.
• May be formed from any legitimate combination of symbols, qualified or not.	• Qualification not permitted in defining expressions.

Thus, although the sequence

```
A  EQU  5
   DUP  A,3
```

is not effective, the intended result can be obtained with

```
A  SET  5
   DUP  A,3.
```

While the function defined by SET is completely independent of this location definition function, and properly applies only to immediate symbols, its usefulness has been increased by extending its interpretation to include ordinary symbols. By definition, the S-value of an ordinary symbol is zero if it has not yet appeared in the location field, and one if it has. The effect of this interpretation is to provide the ability to test the physical sequence of ordinary symbols.

The evaluation procedure, then, is uniform and unambiguous. In any situation involving a decision in the first pass, the S-value of a symbol is used. In the second pass, the S-value is used for immediate symbols; and definition, in the usual sense, for ordinary symbols.

The assembler

conventional two-pass assembler

The customary procedure of a traditional two-pass assembler can be briefly summarized as follows:

- The first pass defines locations to be assigned to the symbols used by the programmer. Definition is accomplished by keeping a counter, called the *location counter*, which is increased by one for every instruction encountered or generated, or by more than one for certain pseudo-operations. Information is retained in a dictionary which has two parts: The *name table*, in which is kept the external form of the symbol; and the *internal dictionary*, in which the definition is recorded, as well as other information of interest, such as relocation structure. The dictionary is built up linearly: an entry is made only when a symbol is encountered in the location field. Except for equivalence operations, the definition is the current value of the location counter.
- The only instructions fully processed in the first pass are those pseudo-operations which affect the location counter. Since there are usually few of these, the linear ordering of the dictionary is tolerable. In the second pass, however, every symbol must be found; hence between passes the dictionary is sorted alphabetically so that the relatively efficient binary search technique may be used.
- The second pass reprocesses all instructions and accomplishes the actual assembly. The identical BCD card images are used, except that, in order to avoid backspacing the input tape, they are taken from an intermediate tape (or tapes) created

during the first pass. Almost all the real work of the assembly occurs in this second pass.

pseudo-operation dictionary

An assembler with deferred symbol definition cannot use the same technique. Observe that the information contained in the sequence:

```
   ORG  A
A  EQU  B
B  EQU  50
```

must be scanned twice, in the order given, before it can be determined that the initial origin is 50. If, then, the assembler is to operate at a reasonable speed, the original source cannot be used for this scan. Instead, it is necessary to construct a *pseudo-operation dictionary* during the first pass which contains all the essential information in the variable field of any pseudo-operation which may affect a location counter. Since the size of this table is necessarily limited, and it may be scanned many times, the information is encoded in binary form (internal text), compact yet amenable to rapid scanning.

use of internal text

The decision to replace all external information by the same text follows naturally. Since a substantial portion of the assembler input is compiler generated for which assembly without listing is the normal mode, use of internal text will produce a substantially faster second pass for two reasons:

- There is a drastic reduction in the length of the intermediate tapes.
- Less time is necessary to process a given instruction.

On the other hand, any instruction which is to be listed must carry along the original form as well as the internal text (since this is stripped of all commentary). However, the increased length of the intermediate tape is counterbalanced by the increase in processing speed; so that the second pass in the list mode is about the same speed as a conventional assembler.

dictionary structure

The decision to use internal text has non-trivial implications for the dictionary structure. Text production in the first pass requires the immediate replacement of a BCD symbol by an internal identifier at every appearance of the symbol, whether in the location field or in the variable field. Hence, to avoid time-consuming searches, the name table is formed non-sequentially by a simple scattering rule which permits rapid placement and retrieval of any symbol.

Contrariwise, there are strong reasons for building the internal dictionary in linear order:

- Consider the control section defined by

  ```
  X  CONTRL  A,B
  ```

  By definition, all symbols processed between A and B lie in the control section X. Now, since the final loading location of the section may be determined by another program, the

relocatable text for

```
CLA     C+2
```

where C in the section (say A+5) is to be represented as

```
CLA     C(X)+7
```

where C(X) is the control dictionary reference for the section X. It is thus necessary to reproduce the actual sequence in which symbols occur in the program. If the internal dictionary is scattered, a "physical order" chain must be kept in each entry.

- Similarly, the exact structure of qualification nesting must be reproducible at all points. A scattered internal dictionary would require a "qualification order" chain in every entry. However, if entries are made sequentially, a simple test for the limits of a qualification section suffices.

Thus, a sequential internal dictionary conserves space in core. Of course, not all is pure gain. Because qualification cannot be determined during the first pass, the internal text must reference name table entries. Hence, some space is required for a *reference table* which correlates name table entries with their internal dictionary correspondent.

basis of structural differences

We see, therefore, that the significant differences between the structure of the IBMAP assembler and other assemblers in the 704/709/7094 family with almost identical languages[2] can be traced precisely to those differences in symbol definition and symbol reference facilities discussed earlier: deferred symbol definition, control sections, and qualification.

Appendix I: Some details of the IBMAP language

Every statement in the IBMAP language is of the form

```
SYMBOL OPN VARIABLE.FIELD.
```

The usage of the symbol, SYMBOL, in the first field (the location field), and the meaning of the variable field, depend upon the operation code, OPN. The VARIABLE.FIELD may be composed of subfields, separated by commas, and is always terminated by a blank.

For machine operations, SYMBOL is the location symbol attached to the instruction; VARIABLE.FIELD is of the general form

```
ADDRESS,TAG,DECREMENT.
```

For pseudo-operations, there are a variety of constructions. The following table lists all that are used in this paper.

The operation	*In IBMAP language means*
`X  CONTRL  A,B`	The control section whose external label is X begins at the location assigned to the symbol A, and extends up to (but not including) the location assigned to the symbol B.

The operation			*In IBMAP language means*
X	ENTRY	A	The location assigned to the symbol A is an external reference point governed by the control section whose external label is X. (This is equivalent to a CONTRL A,A.)
	QUAL	Q	Begin name qualification under the symbol Q.
	ENDQ	Q	End name qualification under the symbol Q.
	USE	A	Switch to location counter A. If the word PREVIOUS appears, the switch is to the location counter in use just prior to the current one.
S	BSS	E	Reserve E cells; that is increase the current location counter by the definition value of the expression E. The symbol S refers to the first of these cells.
S	BES	E	Same as BSS, except that the symbol S refers to the first cell following the reserved group.
	BEGIN	A,E	The initial value of location counter A is the same as the definition value of the expression E.
S	EQU	E	The symbol S is equivalent to the expression E with respect to substitution in any context.
	DUP	E1,E2	Duplicate the next S(E1) statements S(E2) times, where S(E) is the S-value of the expression E.
A	SET	E	Set the S-value of the symbol A equal to the S-value of the expression E. No equivalence between A and E is implied.
S	DEC	L1, L2, $\cdots$, LN	Convert the literal subfields L1, L2, $\cdots$, LN from decimal to binary and store in successive locations. The symbol S refers to the first of these locations.

Appendix II: Symbol definition in IBMAP

We describe here an unusual feature of the assembler mechanization: the process used to define symbols. Details will be illustrated with reference to the sample program of Tables A1 and A2.

Table A1 shows a program written for an assembler without multiple location counters or deferred symbol definition. Table A2 exhibits the same program written in IBMAP language. In the program of Table A2, the three groups of instructions which are indicated in Table A1 have been assigned to different location counters, and the physical sequence has been altered, so that group two appears before group one. Note that this alteration of sequence has caused the variable field of the BES to be un-

Table A1 Sequential form of sample program

Definition	*Location*	*Instruction*		*Remarks*
				Group 1:
0	X	DEC	1,2	Assembled under blank
2		CLA	X	location counter, initial
3	W	STO	Z−1	value of 0.
3	N	EQU	W−X	Equivalence type definition
				Group 2:
4		TRA	Y	Assembled under location
10	Z	BES	N	counter L, whose initial value is determined by final value of blank counter.
				Group 3:
10	Y	HTR	0	Assembled under location counter M, whose initial value is assigned by the BEGIN as equal to that of the symbol Z.

defined when first encountered, thus forcing deferred definition.

Nevertheless, the two programs assemble the same for the following reasons:

1. The blank location counter, which controls Group 1, is considered primary by the assembler. Since it has no BEGIN, its initial value is taken to be zero.
2. Location counter L, which controls Group 2, also has no BEGIN, hence its initial value starts immediately after that of the previous counter. Since counter L is mentioned before counter M, this is the blank counter. Hence, the first instruction of Group 2

   TRA Y

   is assembled at the location immediately following

   STO Z−1

   the last instruction of Group 1.
3. Group 3, under control of location counter M, is initialized by the BEGIN to start at the same location as symbol Z.

The method by which the symbols of the program of Table A2 are assigned the same values as those of Table A1 (as shown in the column marked *Definition*) depends upon the construction in the first pass of the dictionaries illustrated in Tables A3 and A4. For convenience of written representation, the variable field nature of these table entries is shown as a form of VFD (without writing the VFD itself) according to the usual conventions: the number to the left of a slash represents the length of the subfield in bits; the expression to the right of a slash indicates the content of the field.

Table A2 Sample program in IBMAP form

Definition	*Location*	*Instruction*		*Remarks*
4		USE	L	Start with Group 2 under
4		TRA	Y	control of location counter L.
10	Z	BES	N	Note that definition of BES length, N, and of attached symbol Z is deferred.
		BEGIN	M,Z	Initial definition for third location counter (could appear anywhere in program).
0		USE		Switch to main (blank) location
0	X	DEC	1,2	counter and assemble in-
2		CLA	X	structions of Group 1.
3	W	STO	Z−1	
		USE	M	Third location counter now used.
10	Y	HTR	0	
3	N	EQU	W−X	Defines N, which is length of previous BES (could appear anywhere in program).

Table A3 shows the internal dictionary. Except for the blank counter entry, which is initialized, entries are made sequentially for each symbol encountered in the location field, and for each location counter. Symbol entries are one word of the form

6/A,12/SC,3/0,15/NEXT

where A is an adjective code describing the type of entry (1 for an ordinary symbol), SC is the number of instructions between this entry and the previous one (the separation count), and NEXT is the location of the next entry in the location counter chain. Location counter entries are two words: the first corresponds to an ordinary entry for the initial value of the counter (its adjective code is 4); the second, in slightly different format, corresponds to the final value.

Thus, in Table A3, observe that the entry for W at location D+6 has a separation count of 3, since there are three instructions between it and the previous entry for the symbol X, and that it chains to the entry at location D+1. The latter, which is the second entry for the blank counter, has a separation count of 1, representing the last instruction of its scope

STO Z−1

and is chained, in turn, to the first word of location counter L.

Entries in the pseudo-operation dictionary (Table A4) are always two words, plus the internal text necessary to describe the variable field. Although in slightly different format, the first word is similar to an internal dictionary entry; that is, it con-

tains the separation count and the location counter chain. The second word contains the actual pseudo-operation code, split between prefix and tag, and a chain address to the next pseudo-operation. As we shall see, this second chain is essential to the definition process.

Observe that Tables A3 and A4 reproduce the definition structure of the program. Starting at the first word of the entry for the blank location counter and tracing the location counter chains (the separation count at the first word of a location counter entry is always zero), we arrive at the following:

Chain I	*Entry location*	*Corresponds to*	*Separation count*	*Next entry*
	D	Start of blank ctr	0	D+5
	D+5	X	0	D+6
	D+6	W	3	D+1
	D+1	End of blank ctr	1	D+2
	D+2	Start of counter L	0	P
	P	BES N	1	D+4
	D+4	Z	0	D+3
	D+3	End of counter L, and of chain consisting of blank ctr followed by ctr L	0	

Table A3 Internal dictionary after first pass

Entry for	*Table location*	*Content*	*Explanation*	
Blank location counter.	D	6/4,15/D+2,15/D+5	6/4 15/D+2 15/D+5	This is a location counter entry Next location counter at D+2. First entry in this chain at D+5.
	D+1	3/0,15/1,18/D+2	3/0 15/1 18/D+2	No BEGIN for this counter. Last separation count is 1. This counter hooks to counter L
Location counter L.	D+2	6/4,15/D+7,15/P	Similar to blank location counter	
	D+3	3/0,15/0,18/0		
Z	D+4	6/1,12/0,18/D+3	6/1 12/0 18/D+3	This is an ordinary entry. Separation count is zero. Next entry in chain at D+3.
X	D+5	6/1,12/0,18/D+6	Similar to entry for Z.	
W	D+6	6/1,12/3,18/D+1	Similar to entry for Z.	
Location counter M.	D+7	6/4,15/0,15/D+9	6/4 15/0 15/D+9	This is location counter entry. It is the last counter. First entry in chain at D+9.
	D+8	3/1,15/1,18/0	3/1 15/1 18/0	There is a BEGIN for this counter. Last separation count is 1. The chain ends here.
Y	D+9	6/1,12/0,18/D+8	Similar to entry for Z	
N	D+10	6/1,30/0	6/1 30/1	Ordinary entry. No chain enters or leaves.

Chain II	*Entry location*	*Corresponds to*	*Separation count*	*Next entry*
	P+3	BEGIN M, Z	0	D+7
	D+7	Start of ctr M	0	D+9
	D+9	Y	0	D+8
	D+8	End of counter M	1	

Note that:

- Entries under the blank counter and the L counter form a continuous chain. This is a consequence of the sequence rules prescribed for initializing counters without a BEGIN.
- Counter M, which has a BEGIN, forms a separate chain.
- The entry at location D+10 for the symbol N is isolated from the location counter chains, as is the EQU entry at location P+6. However, the pseudo-operation chain word of the EQU entry (location P+7) references N.

Also note that if we consider Chain II as an extension of Chain I, then the sequence is precisely the same as that of the unscrambled program of Table A1.

Table A4 Pseudo-operation dictionary after first pass

Entry for	*Table Location*	*Content*	*Explanation*	
BES	P	3/1,15/1,3/0,15/D+4	3/1	This is first word of entry.
			15/1	Separation count is 1.
			3/0	Counter chain has not been followed.
			15/D+4	Next chain entry at D+4.
	P+1	3/4,15/0,3/1,15/P+4	3/4,3/1	This is a BES.
			15/0	Not used.
			15/P+4	Second word of next pseudo-op at location P+4.
	P+2	Text word describing variable field (N).		
BEGIN	P+3	3/1,15/0,3/0,15/D+7	3/1	This is first word of entry.
			15/0	No separation count.
			3/0	Counter chain has not been followed.
			15/D+7	The counter is at D+7.
	P+4	3/3,15/0,3/0,15/P+7	3/3,3/0	This is a BEGIN.
			15/0	Not used.
			15/P+7	Second word of next pseudo-op at P+7.
	P+5	Text word describing variable field (Z).		
EQU	P+6	3/1,15/0,3/0,15/0	3/1	This is first word of entry.
			15/0,3/0	Not used for this entry.
			15/0	No chain enters or leaves.
	P+7	3/5,15/D+10,3/0,15/P+11	3/5,3/0	This is an EQU.
			15/D+10	Attached symbol is at D+10.
			15/P+11	Second word of next pseudo-op at P+11.
	P+8, P+9	Text words describing variable field (W−Z).		
	P+10	36/0	End of pseudo-operation dictionary.	
	P+11	36/0		

If we add to the pseudo-operation dictionary a hypothetical first entry

```
BEGIN     , 0
```

for the blank counter (since it does not have a BEGIN), we see that all chains start in the pseudo-operation dictionary at some BEGIN entry. Then, if we start at any defined pseudo-operation entry and trace the location counter chain, definition can be lost only at another pseudo-operation. These simple observations form the basis for the following *definition algorithm*:

Step 1.

Starting at the initial entry of the pseudo-operation dictionary, we attempt to evaluate the variable field. (The dummy BEGIN is over-ridden if a blank counter BEGIN appears in the program. In this case, the first pseudo-operation may not be definable.) If it can be evaluated, and if the location counter value is known, the location-counter chain is followed: entries in the chain are defined by adding the separation count to the value of the previous entry. This process is terminated if the chain ends or if another pseudo-operation is encountered. In the latter case, the location-counter value is recorded in the first word of the entry.

Step 2.

If the variable field could not be evaluated or if the chain trace has been terminated, we use the second word of the entry to locate the next pseudo-operation and process it in the same way.

Step 3.

Eventually, we must sweep through all entries in the pseudo-operation dictionary. There are then two possibilities:

a) All entries have been defined; in this case the entire dictionary must be defined.
b) An entry is not defined. In this case, we return to reprocess the pseudo-operation dictionary, ignoring any previously defined pseudo-operations for which the location-counter path has been followed.

Step 4.

We must then terminate in one of two ways: Either by Step 3a, which gives complete definition, or by arriving at Step 3b without having defined a single pseudo-operation on the sweep. In the latter case, there must be some logical inconsistency in the program; for example, a circular definition such as

```
A  EQU  B
   ORG  A
B  CLA  1,4.
```

Tables A5 and A6 show this process carried out for the sample program. Starting with the dummy BEGIN for the blank counter, Step 1 of the definition algorithm applies, and Chain I is traced

to define each item until the BES N is reached. At this point we have

Entry location	*Item*	*Definition*	*Location ctr value*
D	blank ctr	0	0
D+5	X	0	0
D+6	W	3	3
D+1	end blank ctr	4	4
D+2	start ctr L	4	4
P	BES N	—	5

as the items defined in the dictionary. Since we have encountered a pseudo-operation (denoted by the presence of a 1 in the prefix of the word), we proceed to Step 2 of the algorithm.

As it happens, the BES is the next pseudo-operation. Its variable field cannot be evaluated because the EQU entry has not been processed, so that the dictionary entry for N is not defined. Similarly, the BEGIN is bypassed, since Z is not yet defined.

Table A5 Dictionaries after first definition sweep

Location	*Content*	*Remarks*
P	3/1,15/5,3/7,15/D+4	15/5, 3/7 } Location counter value of 5 is available.
P+1	3/4,15/0,3/1,15/P+4	BES still not defined.
P+3	3/1,15/0,3/0,15/D+7	BEGIN not
P+4	3/3,15/0,3/0,15/P+7	processed.
P+6	3/5,15/0,3/0,15/3	3/5, 15/3 } EQU is defined and its value is 3.
P+7	3/5,15/D+10,3/0,15/P+11	Processing of this entry complete
D	6/44,15/0,15/0	6/44, 15/0 } First value of blank counter defined as 0.
D+1	3/4,15/1,18/4	3/4, 18/4 } Final value is 4.
D+2	6/44,15/0,15/4	6/44, 15/4 } First value of location counter L is 4.
D+3	3/0,15/0,18/0	Final value not yet defined.
D+4	6/1,12/0,18/D+3	Entry for Z not yet defined.
D+5	6/41,12/0,18/0	6/41, 18/0 } X defined as 0.
D+6	6/41,12/3,18/3	W defined as 3.
D+7	6/4,15/0,15/D+9	Neither word of entry for
D+8	3/1,15/1,18/0	counter M yet defined.
D+9	6/1,12/0,18/D+8	Entry for Y not yet defined.
D+10	6/41,30/3	6/41, 30/3 } Value of N defined as 3 from pseudo-op entry evaluation.

Table A6 Dictionaries after second (final) definition sweep

Location	*Content*	*Remarks*
P	3/5,15/4,3/7,15/D+4	BES defined as 3, and location
P+1	3/4,15/3,3/7,15/P+4	counter path taken to define Z and final value of counter L.
P+3	3/5,15/7,3/7,15/D+7	BEGIN defined as 7, and location
P+4	3/3,15/7,3/7,15/P+7	counter path taken.
P+6	3/5,15/0,3/0,15/3	Previously defined EQU
P+7	3/5,15/D+10/3/0,15/P+11	not reprocessed.
D	6/44,15/0,15/0	Initial blank counter is 0.
D+1	3/4,15/1,18/4	Final blank counter 4.
D+2	6/44,15/0,15/4	Initial L counter is 4.
D+3	3/4,15/0,18/10	Final L counter is 10.
D+4	6/41,12/0,18/10	Z defined as location 10.
D+5	6/41,12/0,18/0	X defined as location 0.
D+6	6/41,12/3,18/3	W defined as location 3.
D+7	6/44,15/0,15/10	Initial M counter location is 10.
D+8	3/5,15/1,18/11	Final M counter location is 11.
D+9	6/41,12/0,18/10	Y defined as location 10.
D+10	6/41,30/3	N defined as 3.
		As items are defined, the definition replaces the location counter chain in the address of the word, and the sign is set negative.

We now proceed to the EQU and define N since W and X are defined. Since there is no location counter chain, and no more pseudo-operations, we arrive at Step 3, and the situation at the end of the first sweep (Table A5) is that

- The BES is not defined. This prevents definition of the end of Chain I; that is, Z and the last entry for location counter L.
- The BEGIN is not defined. This prevents definition of the entire Chain II.

Definition is complete at the end of the second sweep. For, with N defined, and with the location-counter value known, the BES is evaluated and the definition of Chain I extended to completion. In the process, Z is defined. Hence, the BEGIN can be evaluated, which allows Chain II to be defined completely.

The fully defined dictionary is exhibited in Table A6.

ACKNOWLEDGMENT

The author wishes to express his appreciation for the work of R. A. Rock and C. M. Wimberley who programmed a large part of the assembler, and contributed many valuable ideas to the design.

FOOTNOTES

1. If external references are indirect, as with a FAP subroutine, different coding may be necessary for a program referencing external data than for one referencing internal data.
2. Such as SAP, 9AP, FAP, and BEFAP. Readers familiar with the SCAT assemblers of the sos system will recognize the contribution made to IBMAP. Faced with the problem of deferred symbol definition, the solution was: first pass production of internal text (SQUOZE), a pseudo-operation dictionary (the footnotes), and a scattered name table. However, the form of text and the structure of the internal dictionary were chosen primarily to satisfy output requirements of the assembler, rather than by internal processing considerations.