The macro language design discussed in this paper provides a systematic means by which the SYSTEM/360 *assembler-language programmer can develop macroinstructions, thereby expanding the set of machine-oriented instructions that serve as the basis of the assembler language.*

Also treated is the format of macro definitions, the design of a macro generator, and the principal considerations that governed the design of the system as a whole.

Macro language design for SYSTEM/360

by D. N. Freeman

Of several programming languages for IBM SYSTEM/360, the Assembler Language (AL) is closest to the machine language in form and content.[1-3] Because AL enables the programmer to use all SYSTEM/360 facilities as directly as if he were coding in machine language, AL can be used for all types of applications, and affords the programmer complete freedom in adapting special programming techniques to his specific needs. This paper discusses a macro language that programmers can invoke to reduce AL programming effort and shorten AL source programs.

design principles

With the aid of the macro language,[4] any sequence of AL statements can be "summarized" into a single *macro definition.*[5] Once prepared, this definition may be stored and referred to at any time. In each case, only a single statement, a *macroinstruction,*[5] is written by the programmer. In using macroinstructions, the programmer retains access to all machine facilities; each macroinstruction is expanded into individual AL statements in a predetermined way, and the programmer can intermingle macroinstructions and AL statements.

Systematic employment of macroinstructions simplifies the coding of programs, reduces the frequency of programming errors, and encourages the use of carefully standardized sequences of AL statements for routine functions such as subroutine linkage. Because the programmer has to code fewer lines, fewer source cards are used. Since the programmer often suppresses the listing of generated AL statements when writing in macro language, only

fragments of the total program listing require review. Computer throughput on macro assemblies is thus improved by the reduced volume of printing.

Each macroinstruction is expanded by the *macro generator* into a sequence of AL statements, the exact sequence being governed by the corresponding macro definition. The generated AL statements are then processed like any other AL statements: the basic assembly program translates them into machine-language instructions, assigns storage locations, and performs auxiliary functions necessary to produce an executable machine-language program.

An additional facility, called *conditional assembly*, allows the programmer to specify AL statements which may or may not be generated, depending upon certain programmer-controlled conditions. These conditions are usually tests of values, which may be defined, set, changed, and tested during the course of the assembly itself. By design, conditional assembly may be used independently of macroinstructions, as well as with macroinstructions.

design objectives

A major objective in the design of the macro language was the inclusion of the most successful features of prior macro languages, as long as these features were not contradictory. It was felt that most macroinstruction formats from prior languages should be acceptable to the SYSTEM/360 macro generator. To retain further continuity with the past, macroinstructions should be expandable, if possible, into one-for-one statements (one AL statement for one machine instruction) that are functionally equivalent to one-for-one statements in prior assembly languages.

Some additional design objectives were:

- Macro definitions should be simple in syntax and only require a small number of distinct facilities.
- The macro language should be a nucleus language that meets the linguistic needs of most users, but can be enlarged to accommodate specialized requirements.
- Marginally useful facilities whose implementation would require excessive amounts of main storage should be excluded.
- Assembly speeds should be excellent for simple source programs and acceptable for complex source programs.

The macro language

macro-instruction

When writing a program in SYSTEM/360 macro language, two categories of macroinstructions may be used:

System macroinstructions. These instructions correspond to system macro definitions available in the systems library. Such macro definitions (also called library macro definitions) for standard instruction sequences have been prepared by IBM and can be supplemented by the user. These definitions are available for unlimited use in any number of source programs. Once a macro definition has been edited into the macro library, it can be used

and reused by writing its corresponding macroinstruction into source programs.

Programmer macroinstructions. These instructions and their corresponding programmer macro definitions are created by the programmer for certain sequences in a single program as the need arises. Programmer macro definitions must be placed at the beginning of the source program so that they can be totally edited onto one sequential file. For this type of macroinstruction, the programmer may select his own mnemonic operation codes. If a programmer macro definition becomes useful for several applications, it may be entered into the macro library; it then becomes a library macro definition. There are no format distinctions between programmer and library macro definitions.

Most macro generators can retrieve library macro definitions faster than programmer macro definitions, because the former are read at high speed from systems residence, whereas the latter are usually read from a card reader. Thus, use of library macroinstructions (and definitions) results in faster assembly speeds and reduced card handling.

In addition to macroinstructions, each source program may include single AL statements. Such statements, requiring no expansion, are forwarded by the macro generator without any substantive change to the basic assembly process.

macro definition format

Every macro definition consists of the following sequence of statements:

1. Header statement
2. Prototype statement
3. Declarative statements (optional)
4. One or more model statements
   COPY, MEXIT, and MNOTE statements
   Conditional assembly statements
   These statements are optional (although model statements are used in most macro definitions) and may be intermixed within a macro definition.
5. Trailer statement

The *header statement* (MACRO) identifies the following text of the program as a macro definition.

The *prototype statement* specifies the format of the corresponding macroinstruction and names the *symbolic parameters.*[5] The name-field parameter can be referenced anywhere in the text of the macro definition. The *macro operation code*[5] is the symbol establishing the correspondence of macro definition to macroinstruction. The operand field comprises two consecutive strings of symbolic parameters, each of indefinite (possibly null) length. *Positional parameters* are an ordered sequence of *variable symbols*[6] delimited by commas. *Positional operands* (in relevant macroinstructions) correspond to positional parameters by their left-to-right order. Omitted operands are designated by back-to-back commas; a

Table 1 Keyword parameters for file-definition macroinstructions

Name	*Operation*	*Operand*
FILEA	DTFCD	DEVICE=2540, RECFORM=FIXUNB IOAREA1=BUFFER1 . . .

trailing string of such commas may be omitted from a macroinstruction. *Keyword parameters* are an unordered sequence of variable symbols, such as &KEYPAR1 and &KEYPAR2 in the following prototype statement:

```
&MN   MOP  &PSPAR1,···,
           &PSPARN,&KEYPAR1=VALUE1,&KEYPAR2=VALUE2
```

where

&MN	is the name-field parameter
MOP	is the macro operation mnemonic
&PSPAR1	is the first positional parameter
&PSPARN	is the last positional parameter
&KEYPAR1	is one keyword parameter
&KEYPAR2	is another keyword parameter

In the corresponding macroinstructions, keyword operands need be furnished only if the *standard values* (as furnished in the prototype statement of the macro definition) are to be overridden. Thus, the number of required keyword operands is usually far less than the number of keyword parameters. Furthermore, keyword operands may be written in any order, in contrast to the strict order dependency of positional operands.

Since positional operands are briefer, they are commonly used in *imperative* macroinstructions. For example,

```
GET FILEA,WORKAREA
FETCH PHASE1
```

use positional operands FILEA, WORKAREA, and PHASE1. For file-definition macroinstructions, keyword parameters are preferable. An example for the 16K basic operating system of SYSTEM/360 is given in Table 1; here, some or all of the keyword operands can be omitted if they assume standard values.

Immediately following the prototype statement are the *global* (GBL) *and local* (LCL) *statements* that declare all counters (SETA variables), switches (SETB variables), workboxes (SETC variables), and their associated dimensions (if any). These declarative statements serve three functions: (1) They identify variable symbols to the macro generator (facilitating the detection of subsequent misspellings). (2) They distinguish local SET variables from global

SET variables. *Local* SET variables are reset each time the macro definition is used: local SETA variables—often used as loop counters—are reset to 0; local SETB variables are reset to 0; and local SETC variables are reset to zero-length character strings. Local SET variables have transitory main-storage requirements during macro generation, i.e., they appear when relevant, and vanish when irrelevant. *Global* SET variables permit communication among macroinstructions and between macroinstructions and the main-line program. They are more versatile than symbolic parameters, which only communicate values *from* macroinstruction *to* macro definition. (3) GBL and LCL statements permit the programmer to dimension his SET variables for subsequent indexed references. Thus, the macro generator acquires the familiar advantages of any syntax with indexing: e.g., reduction of total program size, increased speed of reference.

Model statements[5] may be copied unchanged into the generated text; or any desired portion(s) of their name, operation, or operand fields may be replaced with character strings. Portions to be replaced are represented by variable symbols—symbolic parameters, SET variables, or system variable symbols. Generated model statements are in fact the AL statements appearing in the final assembled program.

Macroinstructions themselves may be used as model statements, in which case they are called *inner macroinstructions. Outer macroinstructions* are those used in the main program; they are also called *level-1 macroinstructions. Level-2 macroinstructions* are those used in level-1 definitions, etc.

COPY statements are used to copy AL statements (and/or macro language statements) from a system library into a macro definition or main program. MEXIT statements terminate processing of a macro definition. MNOTE statements generate error messages when the rules for writing a particular macroinstruction are violated; they also may be used to generate other in-line commentary to the user.

The three functions of the *conditional-assembly statements*[5] are: (1) to facilitate elegant, concise representations of the model statements; (2) to generalize a single set of model statements to serve a wide range of operand formats in different macroinstructions;[7] and (3) to permit the macro definition writer to validate macroinstruction operands.

The *trailer statement* (MEND) indicates the end of a macro definition. Like MEXIT, it also terminates processing of a macro definition.

SET statements

One of the significant facilities of the SYSTEM/360 macro language is the SET statement, which assigns a new value to a variable symbol. Three examples of SET statements are:

```
&SUMBOX  SETA 3        (assign a count of 3 to a sumbox)
&SWITCH  SETB 1        (set a switch to TRUE)
&STRING  SETC 'ABC'    (insert a character string into a workbox)
```

Table 2 ADDVEC usage

	Name	*Operation*	*Operand*	*Notes*
Macro definition		MACRO		1
	&MN	ADDVEC	&PAR1, &PAR2	2
		LCLA	&COUNT	3
	&MN	L	0, &PAR(1)	4
	.LOOP	A	0, &PAR1 (&COUNT+2)	5
	&COUNT	SETA	&COUNT+1	6
		AIF	(&COUNT LT N'&PAR1−1).LOOP	7
		ST	0, &PAR2	8
		MEND		9
Macroinstruction		ADDVEC	(OPD1, OPD2, ⋯ , OPDN), SUM	
Generated AL statements		L	0, OPD1	
		A	0, OPD2	
		.		
		.		
		.		
		A	0, OPDN	
		ST	0, SUM	

Notes:
1. Macro-definition header
2. Prototype statement, defining two positional parameters (the first is subsequently used as a sublist parameter)
3. Declaration of a local SETA variable, initial value of 0
4. Model statement with generated name-field symbol; also references first sublist operand
5. Beginning of loop, which generates as many ADD instructions as there are sublist operands
6. Increase counter
7. Loop back if counter is less than remaining number of sublist operands
8. Model statement
9. Macro-definition trailer statement

The name field of each SET statement contains a (possibly subscripted) SET variable, which may be local or global as declared at the beginning of the macro definition.

The three principal uses of SETA variables are as follows: (1) to build length fields and numeric suffixes in generated statements, (2) to index over other variable symbols, and (3) as loop counters during the repeated generation of statements. (See Table 2 for sample uses of a SETA variable.) The operand field of a SETA statement may be as simple as in the first example above, or it may be a compound arithmetic expression (like those of FORTRAN IV) defined by the four arithmetic operators and the following operands: (1) self-defining values, e.g., integers and character constants; (2) symbolic parameters, plus the length, scaling, number, and count attributes of the referenced macroinstruction operands; and (3) other SET variables.

SETB variables are used principally as switches; ordinarily, they are set, reset, and tested in several different statements of

a macro definition. They are a convenience rather than a necessity, i.e., the corresponding logical expressions could be written out at each point of reference.

The operand field of a SETB statement may be as simple as 0 (false) or 1 (true), or it may be a complex logical expression built from arithmetic and character relations. The relational operators are those of FORTRAN IV (although they require no delimiting dots):

EQ NE LT LE GT GE

which mean $=, \neq, <, \leq, >, \geq$, respectively. The logical operators are:

AND OR NOT

Logical expressions testing macroinstruction operands and global SET variables can be used to determine which model statements should or should not be generated.

SETC variables have two principal functions: (1) to analyze piecemeal the operand fields of macroinstructions, and (2) to build symbols in generated statements.

The operand field of a SETC statement may contain: (1) a character string, (2) a symbolic parameter, (3) another SET variable, (4) a concatenation of character strings and variable symbols, or (5) a substring of (1) through (4), i.e., a string of n characters from the total string, beginning with the mth character.

AIF, AGO, and ANOP statements

The single conditional branch in the macro language is AIF, which tests a logical expression and skips at generation time to the statement bearing a certain *sequence symbol* if the expression is true. This is shown in Table 2.

The AGO statement branches unconditionally.

Sequence symbols may be written in the name fields of most assembler statements and macro definition statements. They serve only as reference points for AIF and AGO statements; they are blanked out of the generated statements, since they are irrelevant to the basic assembly process.

The ANOP statement is analogous to the CONTINUE statement of FORTRAN; its sole use is as a reference point for statements whose name fields are pre-empted by symbols or variable symbols.

sublists

Although macroinstructions normally contain lists of scalar operands, it is sometimes convenient to define a vector operand. In the macro language, a vector operand is called a *sublist*. The macro definition can access elements of a sublist by subscripting the corresponding symbolic parameter. The ADDVEC macroinstruction, for example, can generate a variable number of instructions using only a single subscripted parameter to reference the sublist operands. Notice the use of a SETA variable both as a loop counter and as a subscript in the ADDVEC macro definition of Table 2.

There are three system variable symbols with highly specialized functions. The values of the three—&SYSLIST, &SYSECT, and &SYSNDX—are automatically set by the macro generator as each macroinstruction is expanded. Symbol &SYSLIST(n) is an alias for the *n*th parameter of the prototype statement; thus, indexing is possible over the operand list of a macroinstruction. Symbol &SYSECT saves the name of the current control section during the expansion of each macroinstruction. Macro definitions often switch control sections between model statements; such definitions usually resume the original control section before returning to the main-line program, using the following sequence:

system variable symbols

```
&SYSECT    CSECT
           MEND
```

The symbol &SYSNDX tallies the total number of macroinstructions. Its principal use is to concatenate unique numeric suffixes onto symbols generated by the same macro definition at different points in a single program.

The macro generator

The term "macro generator" denotes the initial phases of each SYSTEM/360 assembler having macro capabilities. It recognizes macroinstructions, retrieves the corresponding macro definitions, and generates suitable AL statements prior to the basic assembly process.

Another principal function is the conditional assembly of statements, i.e., varying the number and format of the generated statements. This occurs both during the expansion of macro definitions and also as an independent facility.

The SYSTEM/360 macro generator is a character-manipulation facility only slightly biased towards the instruction formats and data types of the assembler language. It does not interact with the assignment of location values to symbols. During assembly, control does not alternate between macro generator and the basic assembly process. Instead, the SYSTEM/360 macro generator completely transforms a source program into AL statements before location values are assigned.

One important reason for this syntactical restriction has been to improve generation speeds. The groups implementing SYSTEM/360 macro generators have isolated the following system parameters that significantly influence generator speed and design: main-storage size, file-storage speed and access method, and the number of utility files available to the macro generator. (The speed of main storage and the central processing unit influence the speed of macro generators, but generally not their design.) These parameters are now discussed in reference to the implementation on 16K–64K systems.

influence of system parameters

The most significant parameter affecting macro generator and assembler design has been the main-storage size. In a system with limited main storage, each phase can perform only a few

main-storage considerations

processing functions on the text of the source program. Useful activity per statement (i.e., other than merely copying text) is necessarily low. Even if additional main storage is made available, most orthodox algorithms can only allocate the additional main storage to symbol tables (reducing the number of iterations per phase) or to I/O buffers, permitting an improved balance between I/O and computation. Phase re-combination would constitute a truly distinct large-system algorithm.

file considerations

For tape systems, the number of drives determines the number of utility files; in general, there is no advantage in stacking several logical files on one tape reel. Also, each logical file requires a file-definition control block of non-trivial size. Therefore, small macro assemblers cannot profitably use more than three or four tape drives, lest their limited main storage be siphoned off into I/O buffers and file-definition blocks of marginal utility. On the other hand, two tape drives are insufficient for macro generation: in addition to the two drives from which the text is copied back and forth, a non-trivial file of macro definitions must be available on a third drive.

The 8K tape assembler uses tape systems residence and two tape utility files, the 16K tape assembler three tape utility files. Currently, tape systems residence is not available for any larger assemblers. Larger systems with direct-access storage devices (DASD) always use DASD residence to achieve flexibility and to improve system performance. The intermediate and large macro assemblers use three utility files; if systems residence is on disk (or other DASD), the three utility files may be (1) all on disk (either the systems-residence drive or other drives), (2) all on tape, (3) two on tape and one on disk, or (4) vice versa. Thus, at most four algorithms are needed to serve the different possible tape/disk configurations.

The tactical difference of configuration 2 from the pure-tape configuration is only minor. On disk systems, several logical files can consist of tracks allocated to a single drive; the total number of logical files is potentially unlimited. However, each logical file requires a file-definition block, just as in the case of tape. Furthermore, processing time may be wasted if too many files are defined on a single disk drive, since Seek commands must be issued to reposition read/write heads on the proper cylinders. This explains the limitation to three utility files on a disk system.

generator strategies

There are four macro assemblers for small-to-intermediate SYSTEM/360 configurations, identified in the sequel by the main storage size (i.e., 8K, 16K, 32K, or 64K) for which they are intended.

The resident tape for the 8K macro assembler contains not only the program phases, but also the file of library macro definitions in pre-edited form. Thus, the basic "merge" operation of macro generation—macro definitions against source text to produce generated text—uses all three tapes in the system. (Three tapes are needed for many other commercial applications.) Systems with less than three tapes (and no disk storage) have no

macro language support. The 8K macro assembler does not allow programmer macro definitions; all definitions must be edited into the systems library prior to their use.

The 8K macro language is a subset, although a fairly large one, of the full language: SET symbols are limited in number and completely stylized in format. Parameter sublists are not permitted, and attributes are not available for symbols used during macro generation.

The remainder of the implementation discussion is restricted to the techniques of the 16K, 32K, and 64K macro generators for the respective operating systems of SYSTEM/360. These macro assemblers use many common tactics and implement the full SYSTEM/360 language, with a few minor exceptions in the 16K assembler. In a tape-oriented environment, they require three utility files; in many cases, system residence need be searched only once for library macro definitions, which are then edited onto the third utility tape. This reduces tape-searching on system residence to a minimum. In a disk-oriented environment, this same tactic is used; macro generation uses disk utility files serially rather than randomly to reduce arm motion as much as possible.

During macro generation, the physical address of each edited macro definition is held in main storage, i.e., a tape block count or a disk track-and-record address. As source text is read during the generation phase, macroinstructions are intermittently encountered; the address directory is consulted at each encounter, and the file of edited definitions is positioned to the text of the corresponding definition. The definition is then "merged" into the generated text as a purely sequential operation with the following exceptions: (1) AIF and AGO statements force forward and/or backward skips within the macro definition, and (2) whenever an inner macroinstruction is encountered, the edited file must be spaced to the corresponding text of the inner macro definition; when text from the latter is completely generated, processing of the outer macro definition resumes from the point of interruption.

In these small-memory systems, the key tactic has been to edit macro definition text—and conditional assembly statements in the main program—into formats requiring minimum processing during statement generation. (If any substantial interpretation and scanning activities had been deferred until the generation phases, the program logic for interpretation, scanning, and conditional assembly would have far exceeded available main storage.) Since conditional assembly and statement-generation logic just fit into available memory, the edited format must be easy to interpret and without redundancy—single-byte operators and two-byte operand pointers, for example.

Thus, one or more phases of text editing are required before generation on the 16K–64K macro assemblers. The conversion of symbols and variable symbols into operand pointers is a major task—as in most language translators using such a tactic. During

this conversion, statement syntax is thoroughly checked and attributes are collected for symbols that are used during program generation.

dictionaries

Various dictionaries (i.e., generalized symbol tables) are accumulated and interrogated during this symbol conversion: each *macro dictionary* accumulates the local variable symbols and sequence symbols for a single macro definition. The *main dictionary* accumulates this information plus attribute information for the main program. The *global dictionary* records all global variable symbols and macro operation codes. Thus, the global dictionary is relevant to all macro definitions and the main program; other dictionaries have restricted context. This division of the editing process into independent activities substantially reduces the aggregate time for editing all definitions and the main program.

Each dictionary for the 16K and 32K systems contains 16K bytes of two-level storage for symbols and their attributes, whereas the 64K generator builds a combined global/local dictionary. If less main storage is available for a dictionary (which is the normal situation for small systems), its currently accumulating segments are carried in main storage and the remainder in fixed-length segments on a utility file. However, program logic using each dictionary is unaware of this segmentation, since the routine servicing the dictionary retrieves any out-of-main-storage segments as needed. The segment number and the within-segment position of an entry are linear functions of the segment length.

To minimize the number of segment retrievals from the utility file, backward chaining of synonymous entries is used in the macro generators. This procedure ensures that the most recently entered synonyms are in main storage (or in "nearby" segments on the utility file). Symbols are entered and retrieved from each dictionary by a key-transformation technique.[8]

As the main dictionary is built, certain symbols are defined prior to references in macroinstruction operands; other symbols are referenced before they are defined; still other symbols are irrelevant to macro generation. Therefore, the smaller macro generators accumulate a list of *relevant symbols* on one text pass; the attributes for relevant symbols are collected on a subsequent pass.

After each dictionary is complete (e.g., after editing a single macro definition), its extraneous material can be discarded and the dictionary telescoped to a fraction of its original size. Much extraneous material is accumulated in a dictionary during the editing phases: the character representation for each symbol is itself useless during generation, since all attributes are accessed by pointers; the chain links used for synonymous entries in the dictionary are clearly useless; and certain attributes required for editing are not needed during the generation phases. This data reduction activity conserves much main storage for the generation phases—a crucial consideration.

Accessing information in the telescoped dictionary requires

virtual pointers. As each symbol is entered during the editing phases, the position of the corresponding telescoped-dictionary entry can be predicted with complete accuracy. This position is thus the virtual pointer for this symbol, which is inserted into the edited text at each point of reference.

virtual pointers

Although the 16K, 32K, and 64K macro generators have different phase structures, they perform the same six functions described below. The smaller generators collect all program and macro definition text before commencing to edit it; the larger generator completely edits source program text at first encounter. The collection of attributes also differs somewhat among the generators.

The following discussion approximates the flow of the 64K macro generator:

generator flow

Step 1 — Initiate the macro assembly. Subsequent steps are initialized to reflect the assembly-time environment. All available main storage is requested from the control program, dictionaries and other tables are initialized, and the System Input file is opened. If programmer macro definitions are present, control passes to Step 2, otherwise to Step 3.

Step 2 — Edit the programmer macro definitions. Each definition is edited as follows:

(1) All variable symbols are replaced by ordered pairs of flags and virtual pointers. All symbolic parameters have one flag byte; all local unsubscripted SETA variables have a different flag byte; all system variable symbols have still another flag byte, and so forth. Thus, each two-byte pointer references a table identified by the preceding flag byte.

(2) All operation codes are looked up, and certain pseudo-operations, such as COPY, are immediately performed.

(3) Whenever a conditional-assembly statement is encountered, its operand field is completely edited to virtual pointers and arithmetical/logical operators.

(4) All sequence symbols are associated with five-byte file addresses in the macro dictionary: three bytes identify the physical record in which each sequence symbol is defined, and two bytes point to its relative position within the record. Each reference to a sequence symbol is converted to a virtual pointer to its dictionary entry.

Global pointers in the macro definition text refer to tables accumulated without interruption during Steps 2 through 4. Global pointers are necessary for macro operation codes and global SET symbols.

After each macro definition is fully edited, its dictionary is telescoped (as described in the above discussion of virtual pointers), then written onto a utility file just after the corresponding edited text.

Step 3 — Edit the main program. Main-program statements are edited exactly as are macro definition statements. However, attributes must be collected for all ordinary symbols relevant to macro generation and conditional assembly. Each such symbol, together with its attributes, is entered into the main dictionary. Each generation-time reference to a symbol is tagged with the corresponding virtual pointer to the main dictionary. At the end of Step 3, the main-program text file is re-positioned to its initial record, e.g., rewound if on tape. This is, of course, the principal input to the generation phases. The associated main dictionary is telescoped and written onto a different utility file.

Step 4 — Edit the system macro definitions. During Steps 2 and 3, certain operation codes are detected as "undefined." In an error-free source program, these operation codes must correspond to system macro definitions. Rather than immediately match each undefined operation code against the directory of system macro definitions, Steps 2 and 3 merely collect these names. Step 4 retrieves all requested definitions from the system library. Each definition is edited into the same format as that of Step 2. Since level-1 macro definitions may contain level-2 macroinstructions, the list of "undefined" operation codes lengthens and contracts as level-1 system definitions are edited. After all level-1 definitions have been edited, any remaining undefined operation codes are looked up during a second scan through the macro-definition library. This process continues until either no undefined operation codes remain or none of the remainder are found during the preceding scan. Each remaining undefined code is treated as an assembly error. As in Step 2, each dictionary is telescoped and written after the associated text.

Step 5 — Initialize the main storage for generation. The telescoped main dictionary is read back into core. Also, the global dictionary is read back, and core storage is allocated for its global SET variables, which are set to their initial values ("0" for SETA and SETB variables, "empty" for SETC variables).

Step 6 — Generate and conditionally assemble the program. Steps 2 through 4 require one to three passes of the source text, depending upon the particular generator; Step 6 performs the last source-text pass of macro generation. At the end of Step 6, the program has been completely converted into AL statements.

The edited main program is read from Utility File 1, and generated statements are written onto Utility File 2, which is now empty and re-positioned. Main-program conditional assembly is performed as encountered, i.e., for all SET, AIF, and AGO statements. As each macroinstruction is encountered, the dictionary for the corresponding macro definition is retrieved from Utility File 3. This dictionary, in turn, points to the text of the macro definition just preceding it on the same file. Expansion of the macro definition is then performed using:

1. The telescoped main dictionary—furnishing attributes for macroinstruction operands, etc.
2. The telescoped macro dictionary—furnishing sequence-symbol addresses, etc.
3. Macro definition text segments—containing virtual pointers to Items 2, 4, and 5
4. Local-SET-variable values
5. Global-SET-variable values

After each definition has been expanded, Items 2 through 4 may be overlaid. Thus, there are two permanent data areas during Step 6 (Items 1 and 5) and three transient data areas (Items 2 through 4). When an inner macroinstruction is encountered, its dictionary is retrieved from Utility File 3. When the inner macroinstruction has been expanded, its three transient areas are freed and generation resumes from the outer macro definition. Thus, macro dictionaries and local-SET-variable values are allocated last-in first-out storage; more storage is required as the depth of "nesting" increases.

The reading, rereading, and repositioning of Utility File 3 is performed with reasonable speed, since the volume of information is small in view of the edited macro definition text and the telescoped dictionaries; required are perhaps one or two cylinders on a disk drive or a few feet of magnetic tape.

The key characteristic of Step 6 is retention of read-only data on secondary storage until required in the expansion of macroinstructions. Such data includes standard parameter values, symbol attributes, and various positional data. Large-system macro generators have traditionally kept this data resident in main storage; the 16K–64K macro generators cannot afford this luxury. However, random retrieval—even from magnetic tape—does not seriously degrade throughput.

Summary comment

The goal of SYSTEM/360 machine architecture—to describe a single computer system with the widest possible applicability—has its counterpart in the design of a single assembler language. Small-machine users can assemble on larger systems whenever the latter must be used, e.g., when an overflow load is shifted to a service installation. There is a single language syntax for all but the smallest systems; large-machine users can multiprogram assembly jobs with other jobs, selecting the assembler that is appropriate to their main-storage and file-storage resources. All users benefit from a single language and a single set of diagnostics.

The macro definition and conditional assembly functions that supplement the basic language are summarized in Table 3. These do not modify the character of the assembler language, but provide a systematic means whereby a user can expand the assembler language by creating macroinstructions.

Table 3 Macro definition and conditional assembly facilities

Operation Code	*Function*
ACTR	Limits the number of conditional and unconditional branches
AGO	Unconditional branch
AIF	Conditional branch
ANOP	No operation
COPY	Copy source statements
GBLA	Declare a global counter
GBLB	Declare a global switch
GBLC	Declare a global workbox
LCLA	Declare a local counter
LCLB	Declare a local switch
LCLC	Declare a local workbox
MACRO	Begin a macro definition
MEND	End a macro definition
MEXIT	Terminate generation of code from a macro definition
MNOTE	Notify the programmer of a macroinstruction usage error
SETA	Set counter variable to new (positive or negative) value
SETB	Set switch variable to 0 or 1
SETC	Set workbox to new character value

ACKNOWLEDGMENT

The author acknowledges with pleasure the contributions of SYSTEM/360 assembler development groups in the Endicott, Poughkeepsie, and San Jose SDD laboratories. The following persons contributed especially to the language design: A. Lichtman, T. Ragland, H. W. Schmid, Jr., and S. F. Zimmerman, Jr. Groups under J. R. Walters, Jr., and I. A. Tjomsland completed the 16K–64K implementations, which were initiated by the author and his colleagues. Many of the implementation techniques were suggested in G. H. Mealy's GAS monograph.[9]

CITED REFERENCES AND FOOTNOTES

1. G. H. Mealy, "The functional structure of OS/360, Part I, Introductory survey," *IBM Systems Journal* 5, No. 1, 3–11 (1966).
2. *IBM Operating System/360, Assembler Language*, C28-6514, IBM Data Processing Division, White Plains, New York.
3. G. A. Blaauw and F. P. Brooks, Jr., "The structure of SYSTEM/360, Part I, Outline of the logical structure," *IBM Systems Journal* 3, No. 2, 119–135 (1964).
4. The term "macro" serves as an abbreviation for "macroinstruction" when used in combined forms.
5. For the following expressions, the nomenclature for the SYSTEM/360 macro language departs from previous terminology:

   macroinstruction is *macro header* in 7080 Autocoder language;

   macro definition is *macroinstruction* in 7080 Autocoder language;

   symbolic parameter is *operand reference* in 7080 Autocoder language, *substitutable argument* in FAP/MAP;

macro operation code is *macro name* in 1401/1410 Autocoder language, *macro header operation code* in 7080 Autocoder language, *macro operation name* in FAP/MAP;

model statement is *Autocoder skeleton* in 7080 Autocoder language, *prototype instruction* in FAP/MAP;

conditional-assembly statement is *pseudo-macroinstruction* in 1401/1410 Autocoder language, *pseudo command* in 7080 Autocoder language, SET and *conditional-assembly pseudo-operation* in FAP/MAP.

6. A *variable symbol* takes one or more different values during macro generation. The three categories of variable symbols are symbolic parameters, SET symbols, and system variable symbols (&SYSNDX, &SYSECT, &SYSLIST).
7. C. J. Shaw[10] has pointed out two principal advantages in using macroinstructions:

   (1) The ability to generalize a set of operations using a single data type. This is accomplished by simple parametric substitution, as shown in a SYSTEM/360 example:

```
L  O,OPD11              L  O,OPD21
A  O,OPD12     and      A  O,OPD22
ST O,SUM1               ST O,SUM2
```

   may be reduced to

   ADD OPD11, OPD12, SUM1 and ADD OPD21, OPD22, SUM2 respectively.

   (2) The ability to generalize a single verb (i.e., macro operation) over a variety of data types. This requires both parametric substitution, availability of data attributes, and the ability to test these attributes:

```
L  O,OPD11              LH  O,OPD31
A  O,OPD12     and      AH  O,OPD32
ST O,SUM1               STH O,SUM3
```

   may be reduced to

   ADD OPD11, OPD12, SUM1 and ADD OPD31, OPD32, SUM3 respectively. This second capability is a powerful extension of the first. Any macro generator with this feature furnishes certain functions characteristic of procedure-oriented languages, e.g., FORTRAN: a small repertoire of verbs interrogates the data structures of the program to determine what machine instructions should be generated. The 7080 Autocoder contains such a macro generator, whose user acceptance has encouraged wider use of macros on other systems. The SYSTEM/360 macro language includes many of these attribute-interrogation features.

8. W. Buchholz, "File organization and addressing," *IBM Systems Journal* **2**, 86–111 (June 1963).
9. G. H. Mealy, *GAS: A Generalized Assembly System*, Rand Corporation Memorandum 3646, August 1963.
10. C. J. Shaw, unpublished critique of XPOP, System Development Corporation, March 13, 1964. See also M. Halpern, "XPOP: a metalanguage without metaphysics," *Proceedings of the Fall Joint Computer Conference*, 1964.