PROCESS III — A Compiler-Assembler for No. 1 ESS

By N. A. MARTELLOTTO, H. OEHRING and M. C. PAULL

(Manuscript received January 20, 1964)

A description of a compiler-assembler program named PROCESS III is given. This program is used to translate No. 1 ESS symbolic source programs to No. 1 ESS binary object programs. Included is a discussion of the PROCESS language, the Compool and macro facilities of the compiler, and some requirements that motivated its design.

I. INTRODUCTION

No. 1 ESS is a stored-program control telephone switching system.[1,2,3] The program consists of more than 100,000 instructions, each 44 bits in length, made up of 37 information bits and 7 check bits. To write such a program in binary (or octal) machine language is clearly impractical. To efficiently produce such large programs, modern techniques include the use of mnemonics or a symbolic language by the programmer. We say the programmer writes a *source* program in some kind of symbolic language, whereas the central control[2] executes an *object* program in its machine (binary) language.

This article mainly describes the vehicle that translates No. 1 ESS source programs to No. 1 ESS object programs. Certain other items explain the progress of a No. 1 ESS program from inception as a source program to its final state as an object program.

The vehicle developed for translating from No. 1 ESS source programs to No. 1 ESS object programs is itself a program; this program, named PROCESS III,* is executed on the IBM 7094 general-purpose computer. In keeping with the current usage for the words "compiler" and "assembler," PROCESS III is said to be a "compiler-assembler," since it performs both functions, as will be described. Consequently, in this article the words "compiler" and "assembler" are used interchangeably unless otherwise noted.

* PROCESS is an acronym for PROgram to Compile ESS programs.

1.1 *Background*

A compiler should come early in the development of a stored program system. The sooner one can translate to the object program, the sooner interpretive simulation may be undertaken; in turn, this means one may begin to feel confident sooner about such things as adequacy of order structure, and construction and size of the object program. PROCESS III was started early; although there were some definite ideas initially as to what some of its requirements would be, it was necessary to build a flexible structure to accommodate the many new requirements that would arise as the development of the No. 1 ESS program progressed.

Early in the development of No. 1 ESS, it was recognized that the object program would be large, and that it would be written, compiled, and tested in small sections of 1000 to 5000 instructions. Also known was the desirability of not having to disturb all parts of this large program in the semipermanent twistor memory when small sections were being tested and recompiled. Yet the sections had to communicate with each other and transfer control to each other. These considerations led to one basic design criterion for PROCESS III: while still insuring that communication and control among sections remain intact, it must be possible to recompile and reinsert sections of the total object program without disturbing the whole.

A more fundamental design criterion for the compiler arose from a lack of knowledge of the precise nature of the source program. It was known, of course, that the program was to do data processing in connection with telephone calls. Such actions are difficult to describe completely in a simple or mathematical or uniform way. That is to say, there did not exist a "telephone language" that could be used to describe completely the telephone data processing functions. However, it was known that some telephone functions are amenable to simple, mathematical, uniform description; furthermore, as learning took place, it was hoped that the balance of the telephone functions could in time be described in a straightforward manner. Thus, a fundamental design criterion for PROCESS III became flexibility: not only must the compiler be able to handle known straightforward descriptions of telephone functions, but it must also be capable of being used to construct, accept, and retain new descriptions of telephone functions.

The requirements of independent compilation, integrity of communication among sections of object programs, and flexibility have been met by PROCESS III because it:

(1) normally produces relocatable object programs,

(2) has a communications pool (Compool) facility, and

(3) has a powerful macro facility.

A relocatable object program has each of its instructions assigned an address relative to zero, the address of its first instruction. The address field of each No. 1 ESS instruction in a given program section P may in turn refer to one of four kinds of numbers:

(a) constants; these are said to be absolute numbers, since they will not be altered by subsequent processing;

(b) a local program point, in which case the number is a relocatable address within the range of the object program instruction addresses of P;

(c) a global program point, in which case the number is a pseudo-address that refers to a program other than P and therefore cannot be assigned explicity at the time P is compiled; and

(d) fixed call store or program store locations; these numbers are absolute addresses outside the program P but within the range of the two memories.

The total No. 1 ESS object program consists of many relocatable sections like P. Each section contains various numbers, as described. An immediate need for proper execution of the total object program by central control is a loading scheme for inserting this program into the twistor program store. The scheme is implemented by another IBM 7094 program called a "loader." The loader accepts as inputs many object programs generated by PROCESS III and produces as output a single unified (linked) object program containing only absolute addresses. Thus, having determined where sections of the object program will reside in the twistor store, the relocatable program points of (b) and the pseudo-addresses of (c) are changed by the loader to absolute addresses. The absolute numbers of (a) and (d) are not altered. The loader also has the ability to accept a more current version of a section (or sections) of the object program without disturbing the remaining sections. This means that fewer twistor cards need to be remagnetized during the checkout and debugging phases. Finally, the loader will generate and prefix 7 check bits to each 37-bit instruction. Since the Hamming code for these bits is a function of the 37 information bits *and* the absolute address at which the 37 bits reside in the twistor store, they cannot be generated prior to load time.

The Compool facility of PROCESS III enables one to refer conveniently to addresses of type (d). A major consideration in the design of the total object program is the temporary (call store) memory configuration. Temporary memory must be assigned to the call registers,

the network map, various queues, and so on.[3,4] Since there are many programmers involved in constructing the large No. 1 ESS object program, it is efficient to centralize the assignment of temporary memory. The Compool of PROCESS III provides this centralization. Having once defined call store areas in the Compool, the many individual programmers need not concern themselves about such areas except to refer to them as necessary. In their programs, reference to these areas must be made symbolically, but there is no need for individual definition of such areas; the communications problem among programmers using the same call store areas is thereby substantially reduced by the Compool facility.

The Compool, therefore, is a collection of symbols that are assigned call store addresses; these symbols correspond to areas set aside in memory in a particular configuration for the purpose of doing telephone data processing. Of course, the configuration may change as the development of the object program progresses. In this case, PROCESS III is used to update the Compool, and the new configuration is used in compiling all source programs thereafter. It may be necessary for source programs to be recompiled, depending on the change in call store configuration. This too is partially automated: when a Compool run (i.e., a change in call store configuration) is made, PROCESS III refers to its "bookkeeping file" to ascertain which source programs, if any, need to be recompiled. A list is printed out and the individual programmers are notified. They need only recompile without concerning themselves as to precisely what call store changes have been made.

It was mentioned that PROCESS III has a powerful macro facility. In this context, a macro is defined to be a fixed amount of code, in some language, that will result in a variable amount of object program when it is properly "called." Just as basic instructions have variable fields (e.g., address, index, masking),[2] so do macros; in the case of macros, the variable fields are called "parameters." Since calls occur in the source program, the names of the macros, or more simply the macros, are said to be part of the source language. If one is given the ability to define his own macros he may thereby extend the source language. A set of macros which has been incorporated permanently into PROCESS III constitutes just such an extension. These permanent macros are saved as a special part of the Compool so that they may be refined and extended still further as more is learned about telephone data processing functions. In addition, individual programmers may define and call their own macros.

The balance of this paper is devoted primarily to three things:

(1) an explanation of how call store configurations are defined,
(2) a description of the source language and some of its extensions, and
(3) a discussion of the tools available to construct extensions to the source language.

II. STORAGE ALLOCATION

The design of the compiler was influenced by the real-time and space-limited nature of the resulting object program on the one hand, and the demands of an intricate basic machine order structure on the other. An obvious solution to the space problem was to pack information into subunits smaller than the natural dimensions of the available memory units (37 binary bits in the program store and 23 bits in the call store). Thus it was desirable to provide means in the compiler for naming such subunits, called "items," in memory.

The object program organization[3] itself demanded of the compiler the ability to define several types of homogeneous blocks of temporary memory, each composed of several basic memory units. A group of call registers[3,4] is essentially a group of similar blocks of memory, where each word within one block serves the same functions as the corresponding word in all the other blocks. For example, the high-order bit of the second word in each block may indicate whether this call register is controlling a telephone call or not. Memory blocks of this call register type are called "scatter tables" and are defined in PROCESS III by the following statement:

```
(1)  XX      SCATABLE   N1,N2
```

Beginning at an address called XX, statement (1) reserves space in memory for N1 tables each containing N2 words.

The statement,

```
(2)  YY      TABLE      N1,N2
```

defines N1 tables each containing N2 words. Memory defined with TABLE differs from that defined with SCATABLE; all the words in any one of the N1 tables defined with TABLE have the same function, but the function may vary from table to table. For example, with N1 = 2, the rightmost 17 bits of each word in the first table might be used to store the line equipment number, while the second table might consist entirely of trunk equipment numbers in the rightmost 15 bits.

It is also convenient to be able to define one continuous block of storage. The compiler accepts statements like:

```
(3)  ZZ      BLOCK     N
```

This reserves N words of space in memory and names the address of the first word ZZ.

In order to refer to memory items smaller than the basic word by name, one must be able to define them. The statements below serve this purpose:

```
(4)  YY      TABLE     10,50
(5)  YY0     LAYOUT    ABB----------CCCCCCCC
(6)  IT1     ITEM      A
(7)  IT2     ITEM      B
(8)  IT3     ITEM      C
```

Statement (5) lays out all 50 words of the first table in the set of 10 tables defined by (4) in the call store area. The high-order bit is defined as one item, the next 2 bits as a second item, the dashes indicate bits not being defined, and the rightmost 8 bits constitute a third item. Statements (6), (7), and (8) assign names to the three items.
Thus IT1 is the name assigned to the item indicated by A in the LAYOUT statement, IT2 is the name of the B item, and IT3 the name of the C item. In a similar way items may be defined for SCATABLE and BLOCK.

The following characteristics of items are useful in manipulating them, as will become obvious in the following sections.

```
A.ITEM = the address of the item
M.ITEM = the mask of the item; i.e., a 23-bit constant with all ones
         in the position of the item and zeroes elsewhere
D.ITEM = the displacement of the item from the right
S.ITEM = the size of the item; i.e., the number of bits contained in
         the item.
```

The qualifiers A., M., D., and S. are prefixed to the name of the item to refer to these characteristics. For example, the items defined in statements (6), (7), and (8) have the following characteristics:

```
A.IT1 = A.IT2 = A.IT3 = address named YY0
M.IT1 = O.20000000*
```

* PROCESS III assumes integers to be decimal unless qualified by O. to indicate they are octal.

M.IT2 = O.14000000
M.IT3 = O.377
D.IT1 = 22
D.IT2 = 20
D.IT3 = 0
S.IT1 = 1
S.IT2 = 2
S.IT3 = 8

III. SOURCE LANGUAGE

3.1 *Basic Orders*

Some of the storage allocation or storage defining facilities of PROCESS III were tailored for use by the No. 1 ESS basic orders. Since the nature and aims of the basic order structure have been discussed in some detail in an earlier article,[2] only some of their pertinent characteristics in connection with the use of item qualifiers are illustrated here.

(9) MK YY0

This instruction moves the 23-bit contents of the word named YY0 into the K register. The following instruction (10) does exactly the same thing:

(10) MK A.IT1

The triplet below [instruction (11)] sets the logic register to the mask of IT2, moves the contents of the item IT2 into the K register, masking out everything but the item by logical product (PL), and then right adjusts it in the K register by using the displacement of the item.

(11) WL M.IT2
MK A.IT2,,PL
HC D.IT2

The same actions are accomplished in a different way by the next three instructions:

(12) WX A.IT2
MK M.IT2,X,PS
HC D.IT2

Because of the PS option the mask is set up and used in the same order.

Given a source program input such as the basic order instructions in (9), (10), (11), or (12), PROCESS III will produce an object program as output with the property that for each input instruction there will be exactly one output instruction. This one-to-one correspondence between input source program instructions and output object program instructions defines an assembling process. On the other hand, a compiling process is defined to be a one-to-many correspondence between input source program "instructions" and output object program instructions. For a compiler the input "instructions" are called "statements," and the set of all statements meaningful to the compiler is called a "language." PROCESS III will accept as inputs basic order instructions and statements of its own language, called PROCESS, in any mixture.* The source language of No. 1 ESS programs, therefore, is nominally PROCESS plus the basic order instructions. It will be shown that the PROCESS language can be extended.

3.2 *PROCESS Language*

The basic goal of the PROCESS language is to provide the No. 1 ESS programmer with a means to write his source programs quickly and efficiently. As with any higher-level programming language, a program written in PROCESS cannot be better in terms of object program length than the same program written with basic orders by an *expert programmer*. When used properly, however, the PROCESS language gives object programs that are no worse than the great majority of those written with basic orders by average programmers.

The PROCESS language is intended to provide the fundamental programming tools needed to write telephone programs. The functions required in any program, including telephone programs, can be classified into three categories:

(1) moving data from one place to another,
(2) making decisions using these data, and
(3) performing arithmetic or logic operations on these data.

While these requirements were predetermined, there were some additional telephone-oriented programming problems that evolved as the programming effort got under way. These problems were solved readily because of the flexible macro facility of PROCESS III and the resulting ease in extending the source language by defining new procedures.

Essentially, the PROCESS language consists of a set of procedures

* Hence the designation of PROCESS III as a "compiler-assembler."

that can be called by the programmer with varying input information, called "parameters." A description of the allowable parameters and procedures follows.

The nature of parameters may be discussed on two levels:

(*i*) On the lower level, parameters can assume the identity of any of the basic units of data handled by the basic orders. The various possibilities are:

(a) full memory words, indexed or not
(b) items or partial words, indexed or not
(c) constants
(d) central control registers.

For (a) and (b), indexing is specified by enclosing the address and index in parentheses, as (address, index).

(*ii*) Parameters can also assume a more general nature that enables a programmer to nest procedures. In this case a parameter can be represented by:

$$\mathrm{OP}(a_1 , a_2 , \cdots , a_n)$$

Here the operator OP of the parameter can be any of the basic procedures of the language or the character C; the use of C indicates that the body of the parameter $(\cdots)$ should be complemented. Each a_i again can be of the form $\mathrm{OP}(b_1 , \cdots , b_m)$ or one of the forms defined by (a) through (d) above.

The general-purpose procedures of the PROCESS language are:

1. Data moving

MOVE OP(x),b,c,d,$\cdots$

Function: Move the quantity specified by x to the destinations designated by b, c, d, and so on. If OP is a basic procedure, perform this operation on x before moving the result to b and c, etc.

2. Decision making

(a) IF OP(x),($r_1 , r_2 , \cdots$),OP(b),($d_1 , d_2 , \cdots$)

Function: If the quantity x or the result of the operation (if any) performed on x has the relation r_i to b or to the result of the operation (if any) performed on b, then control is transferred to a program location named d_i (i = 1,2,3). The allowable relations r_i and their meanings are:

E = arithmetically equal
NE = arithmetically not equal

LE = arithmetically less than or equal to
GE = arithmetically greater than or equal to
L = arithmetically less than
G = arithmetically greater than
XE = logically equal
XU = logically unequal.

(b) COMP $\quad OP(x),r,(b_1, b_2, \cdots, b_n),$
$(d_1, d_2, \cdots, d_n)$

Function: If the quantity x or the result of the operation performed on x has the relation r to b_i, program control is transferred to d_i $(i = 1, 2, \cdots, n)$.

(c) IFOR $\quad OP(x),r,(b_1, b_2, \cdots b_n),d$

Function: If the quantity x or the result of the operation performed on x has the relation r to any one of quantities b_i $(i = 1, \cdots, n)$, program control is transferred to d.

(d) IF $\quad x,r,b,OP_1(c_1),\cdots,OP_n(c_n),$
$ELSE(OP_{n+1}(c_{n+1}),\cdots,OP_m(c_m))$

Function: If x has the relation r to the quantity b, then perform the procedures $OP_1, \cdots, OP_n$; if not, then do $OP_{n+1}, \cdots, OP_m$.

(e) IF $\quad x,r,b,OP_1(c_1),\cdots,OP_n(c_n),$
$ALSO(OP_{n+1}(c_{n+1}),\cdots,OP_m(c_m))$

Function: If x has the relation r to the quantity b then perform the procedures $OP_1, \cdots, OP_n$; in any case then do the procedures $OP_{n+1}, \cdots, OP_m$.

3. Arithmetic and logic procedures

(a) SUM $\quad OP(x),OP(y),b,c,\cdots$

(b) DIFF $\quad OP(x),OP(y),b,c,\cdots$

(c) AND $\quad OP(x),OP(y),b,c,\cdots$

(d) OR $\quad OP(x),OP(y),b,c,\cdots$

(e) EXOR $\quad OP(x),OP(y),b,c,\cdots$

Function: Perform the indicated operation on the parameters x and y after executing the function of the operators on x and y, if any, and put the result into b, c, etc.

4. Loop control

m	LOOP	i,f,c,v
	⋮	
m	ENDLOOP	

Function: These two statements control a loop that begins with the call of LOOP and ends with the call of ENDLOOP which specifies the same m (name) in the location field. i and f are the initial and final values of the loop variable. At the end of each pass through the loop, the loop variable is incremented by c. When this new value exceeds f, control passes to the next instruction outside the loop; otherwise control is transferred to the beginning of the loop. v specifies the loop variable to be used. If v is not specified, the central control register Z will be used as a loop variable. It is possible to nest loops within loops. If the same loop variable is specified for more than one loop, the value of that variable is saved and reset when entering and leaving another loop.

5. Initialization facility

INIT	$b,c,\cdots$

Function: Place c and any following parameters into consecutive locations starting with the location specified by b.

6. Unconditional transfer of program control

GO*TO	OP(x),(b_1 , b_2, $\cdots$, b_n)

Function: At execution time, x or the result of the operation performed on x, if any, specifies a number i, and program control is transferred to b_i . If b_i is not specified, program control is transferred to x.

Some typical calls for these procedures are:

```
START   LOOP      1,10,1
        MOVE      (ITEM,X),FULL
        IF        (ITEM,Y),E,0,DEST
        IF        (0,Y),E,L,MOVE(0,(0,Y)),
                    ELSE(MOVE(L,(0,Y)))
DEST    SUM       SUM(1,(ITEM,X)),K,(0,Z)
START   ENDLOOP
        GO*TO     SUM(Y,(ITEM,Z)),(D1,D2,D3)
```

The procedures described constitute the initial general-purpose subset of the PROCESS language. A realistic example showing the use of many of these procedures is given in Appendix A.1.

As the needs of the No. 1 ESS program became clearer, special telephone-oriented procedures were added to this initial set to extend the source language. For instance there are procedures to implement a change in network (CIN), a change in peripheral circuit configuration (CIC) or signal distributor (SD) actions. These procedures have enabled the programmer to implement such functions in a higher-level and more descriptive language, thereby relieving him of details involved in writing source programs at the basic order level.

Procedures in the PROCESS language are in fact macro calls. The corresponding macro definitions for the language are retained by the compiler in its Compool. To extend the source language, one defines new macros. The extensions may be global or local. Global extensions to the source language are macros that have proven to be of widespread use among the programmers; these are entered into the Compool and become part of the PROCESS language, capable of being used thereafter by all programmers. Local extensions to the language are macros that are defined and called by individual programmers in their own programs. Obviously, there may be many different kinds of local extensions to the source language. Extensions, whether global or local, are constructed using special macro orders in the macro definitions.

IV. MACRO DEFINITIONS

Each macro definition is associated with a definite name (or set of names) called a "macro name." When the compiler encounters a macro name in the operation field of the source program it looks for the definition associated with that name. The compiler then executes the orders in the macro definition, which results generally in No. 1 ESS code being generated. This code varies, depending on the parameters of the macro call.

A macro definition has the form:

DEFIN	op_1	$dum_1 , dum_2 , \cdots , dum_n$
	$order_1$	
	$order_2$	
	$\vdots$	
	$order_k$	

ENDEF

EQUAL op_1 $op_2 , op_3 , \cdots , op_n$

$op_1 , op_2 , op_3 , \cdots , op_n$ are all names of this definition. $dum_1 , dum_2 , \cdots , dum_n$ are dummy parameters. The body of the definition consists of the series of orders: $order_1 , order_2 , \cdots , order_k$. These orders are of four types: No. 1 ESS instructions, macro calls, macro orders, and pseudo operations. The macro orders are a special subset of the PROCESS language useful mainly in writing macro definitions. They instruct the compiler to take certain actions *during compile time*. To help distinguish macro orders from other types of orders, the symbol * is the first character of each macro order name. The meaning and syntax of the four order types are discussed more fully in this and succeeding sections.

The actual notation used by programmers in writing macro definitions is limited by available keypunch symbols, which leads to awkward notation in some cases; for these cases a notation more suitable for exposition is used here. As previously, the remainder of this section uses small letters for variables, whereas capitals and special symbols such as $ are used literally. A detailed example of a macro definition, a macro call, and the operation of the compiler in expanding the definition is given in Appendix A.2.

4.1 *Parameter References*

Depending on the exact nature of a macro call, different codes are generated by its macro definition. In order to express this dependence of the code to be generated on the various parts of the macro call, a general scheme for referring to these parts is needed. The form of a macro call is:

loc op $p_2 , p_3 , \cdots , p_n$

A call is composed of a location (also called p_0), an operation (also called p_1) and a series of parameters. Syntactically, the location and operation are strings of six or fewer alphanumerics. The parameters, on the other hand, may have some internal structure. A parameter p_s is either a string of alphanumerics (including the null string) or it has the form $o_s(p_{s,1} , p_{s,2} , \cdots , p_{s,n_s})$ in which s is an ordered set of (position) integers, and o_s is any string of alphanumerics. An o_s is called the *operator* of parameter p_s. p_s with o_s removed is called the *body* of p_s. The *strip* of p_s is the body of p_s with the outside parenthesis removed. The *remainder* of p_s is defined only for $s = 2, 3, \cdots , n$. The remainder of p_s

is p_s , p_{s+1} , $\cdots$, p_n . The location, operation, and the parameter parts, operator, body, strip, and remainder of any p_s may all be referred to directly within a PROCESS III macro definition.

There are two methods to refer to parameters and parameter parts. The first method is by using "parameter indices." There are six "parameter indices" named I0,$\cdots$,I5 for use in writing macro definitions. A parameter index may be set to any parameter by the macro order *SET. For example:

*SET x,I0,s

This order sets parameter index I0 to parameter p_s if it exists; otherwise control jumps forward to location x in the macro definition. There is a companion macro order *ADV. For example:

*ADV x,I0,t

Assuming t is an integer and I0 is originally set to p_s , where $s = s_1 , j$, then this order will set I0 to $p_{s_1,(j+t)}$ if such a parameter exists, and control passes to the next order; otherwise control jumps forward to location x in the macro. For example, if I0 is set to $p_{2,1}$ and $t = 1$, then after a successful *ADV, I0 is set to $p_{2,2}$.

One may refer to the parameter part to which a parameter index is set as follows: assume Ij is set to p_s

"*O""Ij" refers to the operator of p_s .
"Ij" refers to the body of p_s
"*S""Ij" refers to the strip of p_s
"*R""Ij" refers to the remainder of p_s .

The second method allows one to refer to parameters p_1 , p_2 , $\cdots$ p_n and their respective parts directly by referring to the dummy parameters written on the DEFIN statement. The DEFIN statement has the form:

DEFIN op dum$_2$,$\cdots$dum$_j$,$\cdots$,dum$_n$

dum$_j$ may be any string of six or fewer alphanumerics. In writing the macro definition following a DEFIN statement, one may refer to parameters and their parts as follows:

"LOC" refers to p_0
"ZOP" refers to p_1
"*O""dum$_j$" refers to the operator of p_j
"dum$_j$" refers to the body of p_j

"*S""dum$_j$" refers to the strip of p_j
"*R""dum$_j$" refers to the remainder of p_j.

If by using an expression available for referring to parameters one refers to a parameter that does not exist, then that expression refers to the symbols MSP (missing parameter).

All the above ways of referring to parameters are called parameter references ⟨pr⟩. Parameter references can be concatenated with each other and can be concatenated with alphanumerics. (Alphanumerics exclude special symbols.)

† A concatenated parameter reference ⟨cpr⟩ is defined to be of the form:

$$x \text{ or } \langle \text{pr} \rangle \text{ or } \langle \text{cpr} \rangle \langle \text{cpr} \rangle^*$$

in which x is any string of alphanumerics.

† A ⟨pr⟩ refers to a parameter. A string of alphanumerics x refers to x. A concatenation of x's and ⟨pr⟩'s refers to the concatenation of the symbols to which the x's and ⟨pr⟩'s refer (referents). Generally, the concatenation of any set of expressions refers to the concatenation of the referents of the individual expressions.

4.2 *Numerical Indexing*

There are three "numerical indices," M0,M1,M2, available for use in writing macro definitions. These indices refer to numbers. A numerical index may be set to a number with the macro order *SFI. For example:

*SFI x,M0,n_1,n_2

This sets index M0 to n_1 with a limit of n_2, where both n_1 and n_2 are positive integers. If $n_2 < n_1$, the index will not be set and a jump forward to x will be executed. There is an associated macro order *AFI. It is written:

*AFI x,M0,n

If M0 is set to j when this order is encountered, then M0 will be set to $j + n$ provided $j + n$ does not exceed the limit established on the last *SFI order that referred to M0, and control jumps back to location x in the macro; otherwise control passes to the next order.

---

* This recursive definition states that an x or a ⟨pr⟩ is a ⟨cpr⟩ and that any concatenation of ⟨cpr⟩'s is also a ⟨cpr⟩.

† This and any paragraphs similarly marked may be omitted without loss in continuity by those not interested in the detailed syntax of macro definitions.

One may refer to the number to which a numerical index Mj refers by the expression "Mj".

† ⟨cnr⟩ is defined to be a concatenation of numerical index references. A ⟨cmr⟩ is defined to be of the form:

⟨cnr⟩ or ⟨cpr⟩ or ⟨cmr⟩⟨cmr⟩

⟨cmr⟩ in itself is not significant; it is used as a convenience in a definition given below.

4.3 *Naming Symbol Strings and Parts of Symbol Strings*

A string of alphanumerics may be named and later referred to by this name. Also, space must be allocated to hold the strings to which the name refers. One method of naming strings and at the same time allocating space is accomplished outside all macro definitions with the NAME statement:

```
NAME    nam             siz,strg
```

nam is the name of the string, siz is the maximum-length string to which this name refers, and strg is an initial string of symbols to which nam refers. The name of a string is limited to six characters.

A method of renaming strings is with the macro order *ST. For example,

```
*ST             s,p
```

gives the string s the name p previously assigned by a NAME statement. Later the string s may be referred to by writing [p].

† A string reference ⟨sr⟩ is defined to be of the form [⟨nr⟩] in which ⟨nr⟩ is defined to be of the form:

⟨cmr⟩ or ⟨sr⟩ or ⟨cmr⟩⟨sr⟩ or ⟨sr⟩⟨cmr⟩

An ⟨sr⟩ of form [⟨nr⟩] refers to the string whose name is the referent of ⟨nr⟩ as defined by using the NAME or *ST statements. If the referent of ⟨nr⟩ is not such a name, then [⟨nr⟩] refers to UN (undefined name).

4.4 *Special Functions*

Since it is expected that the parameters of a macro call will in many instances be the names of temporary storage elements such as items, registers, full words, and so on, means are provided for referring to properties of storage elements. These properties are:

(1) Type: [T.⟨nr⟩] refers to different characters, depending on what ⟨nr⟩ refers to.

If ⟨nr⟩ refers to:	[T. ⟨nr⟩] refers to:
an item	S
a full word	F
a number	W
a register item	P
a register	R
none of the above	UN

(2) Item properties: if ⟨nr⟩ refers to an item, then [S.⟨nr⟩], [D.⟨nr⟩], [M.⟨nr⟩], refer respectively to the size, displacement and mask of this item. If ⟨nr⟩ is not an item all three expressions refer to UN.

4.5 *Reference Expressions*

† A reference expression is a basic element in writing macro definitions. Recalling the various allowable bracketed expressions (i.e., [⟨nr⟩], [T.⟨nr⟩], [S.⟨nr⟩], [D.⟨nr⟩], [M.⟨nr⟩]), let ⟨csr⟩ be any concatenation of these, or null. A reference expression ⟨r⟩ is defined as any concatenation of ⟨nr⟩'s and ⟨csr⟩'s. ⟨r⟩ is of the form:

⟨nr⟩ or ⟨csr⟩ or ⟨r⟩⟨r⟩

† A legitimate reference expression always refers to some string of symbols. The interpretation of this string of symbols in turn depends on its position within the macro string.

4.6 *Conditional*

It has been shown how one can refer to parameters and various functions of parameters. Any such reference has been called a "reference expression," and has been symbolized by ⟨r⟩. The problem now is to produce code that depends upon these parameter functions. To do this some way of specifying decisions is required. The conditional is provided for this purpose.

One of the forms of the conditional is:

$$\$c, q_1, q_2, n_1, n_2\$$$

Syntactically, c, q_1, q_2, n_1, n_2 are all of the form ⟨r⟩. A legitimate c refers to the letters C, E, G, or L and indicates the type of comparison to be made. q_1 and q_2 refer either to strings of symbols or numbers that are to be compared depending on the interpretation of c. n_1 and n_2 refer to numbers that indicate how many characters following the conditional (after the second $) are to be omitted: n_1 characters if the condition is met, n_2 characters if the condition is not met. For the condition indicated

by the letter C, the compiler compares the two strings referred to by q_1 and q_2 for identity. For the conditions indicated by the letters E, G, and L the compiler compares the two numbers referred to by q_1 and q_2 to determine if q_1 is respectively equal to, greater than, or less than q_2.

Another conditional is of the form:

$$\$X, p, n_1 , n_2\$$$

Syntactically, n_1 and n_2 are ⟨r⟩'s, but p must be a parameter reference ⟨pr⟩. The interpretation of this conditional by the compiler is: if the parameter referred to by p exists, omit the next n_1 characters; if not, omit the next n_2 characters (after the second $).

Finally there is:

$$\$U,n\$$$

which means "omit the next n characters." n is of the form ⟨r⟩.

In general, conditionals may be concatenated with each other and with reference expressions.

4.7 *Form of Orders Used in Writing Macros*

A macro definition is composed of a series of orders. The form of these orders is:

loc $\quad$ op_1 $\quad$ $p_2 , p_3 , \cdots , p_n$

or

loc $\quad$ * $\quad$ $\mathrm{op}_2 , p_2 , p_3 , \cdots , p_n$

loc is an ⟨r⟩ consisting of six or fewer characters; op_1 is any concatenation of ⟨r⟩'s and conditionals totaling six or fewer characters in length. Since six characters do not allow many ⟨r⟩'s or conditionals for op_1, the second form is available, in which op_2 is the same as op_1 except there is no limit on its length. A parameter p is either a concatenation of ⟨r⟩'s and conditionals or of the form $\mathrm{op}(p,p,\cdots,p)$.

Thus an order used in a macro consists of conditionals which must be performed, reference expressions which must be interpreted, and operations which must be performed. The compiler does these things in the following fixed sequence.

(1) The conditionals are performed in sequence from left to right. A conditional is performed by:

(a) first interpreting all ⟨r⟩'s in the conditional (substituting referents for references) and then,

(b) certain parts of the order are omitted, depending upon the kind of conditional and substituted referents.

(2) All ⟨r⟩'s in that part of the order which remains are now interpreted.

(3) The resulting order (called an interpreted order) is performed. The resulting order is one of four types:

(a) an ESS instruction in the format required of such an instruction. If the compiler arrives at one of these in a macro definition, the ESS instruction is made part of the compiled object program.

(b) a macro call of the form described in Section 4.1. If the compiler arrives at one of these it transfers control to the definition of this macro and begins executing the orders in that definition.

(c) a macro order. Some of these already have been described, namely, *SET, *ADV, *ST, *SFI, and *AFI. The remainder of the macro orders are described below.

(d) a pseudo operation. The compiler executes the pseudo operation just as though it had been part of the input source program (see Section 5.1).

4.8 *Additional Macro Orders*

The remaining macro orders are all jumps or skips of one sort or another. Let x be any string of six or fewer alphanumerics, n a number.

$$*\mathrm{J}\begin{pmatrix}\mathrm{F}\\ \mathrm{B}\end{pmatrix} \qquad x$$

means jump $\begin{pmatrix}\text{forward}\\ \text{back}\end{pmatrix}$ to the location x.

$$*\mathrm{JF} \qquad \mathrm{OUT}$$

means jump out of this macro definition.

$$*\mathrm{S}\begin{pmatrix}\mathrm{F}\\ \mathrm{B}\end{pmatrix} \qquad n$$

mean skip $\begin{pmatrix}\text{forward}\\ \text{back}\end{pmatrix}$ over n orders.

$$*\mathrm{X}\begin{pmatrix}\mathrm{F}\\ \mathrm{B}\end{pmatrix} \qquad x$$

means execute the order at location x $\begin{pmatrix}\text{forward}\\ \text{back}\end{pmatrix}$ of this execute order and then return to the order directly after this execute order.

In the case of the execute order, the location x must be in the same definition as the execute order. In the case of the jump or skip orders the transfer of control can be outside the macro in which the jump or skip occurs.

A jump forward to location x in the definition of a macro called MAC causes the compiler to look for x in MAC, forward of the *JF order. If x is not found in MAC, the compiler continues to look forward of where the call for MAC occurred. This process continues until the x is found or the end of the input program is reached. A jump back, *JB, is executed similarly, except that if PROCESS III gets back to the input source program without having found an x, it will not look any further; the compiler will then process the next order in the source program. The skip macro orders follow corresponding rules. A detailed example of how the compiler handles a macro call is given in Appendix A.2.

V. SOME RELATED DETAILS

There are many features of PROCESS III that have been omitted for the sake of brevity. However, a few details are mentioned below in an effort to complete the general facilities of the compiler.

5.1 *Pseudo Operations and Output Listing*

PROCESS III has a variety of pseudo operations. Pseudo operations are orders to the compiler that cause it either to generate data or to take some special action. Many of the special actions have to do with print control of the output listing. Two typical pseudo operations are:

```
OCT      1000007777777
SPACE    2
```

The first generates 37 bits of data consisting of the octal number shown; the second causes two blank lines to be "printed" on the output listing.

The output listing of the compiler is part of the documentation of the No. 1 ESS program. The listing contains the symbolic source program as written by the programmer, and also an octal representation of the object program. An example is shown in the Appendix, Section A.3.

5.2 *Machine Restrictions*

An interesting feature of PROCESS III is its ability to check for (and sometimes correct) certain violations in the source program. In addition to the usual checking performed by an assembler (e.g., unde-

fined and multidefined symbols), PROCESS III checks for illegal sequences of basic order instructions. These sequences, usually couplets or triplets, are illegal because of timing restrictions of the No. 1 ESS central control. The compiler either flags the violations or inserts EE (no operation) instructions to correct the sequence.

5.3 *Input and Correction Features*

The input to PROCESS III may be either tape or cards; in the case of cards, two formats are available, symbolic or crunched. Symbolic card input means that there is a single basic instruction or order or procedure or pseudo operation per physical card; crunched card input is simply a compressed version of the symbolic information, so that more than one instruction is introduced per physical card. With crunched input every instruction in the source program is assigned a sequence number. These sequence numbers may be used by the programmer to modify his source program conveniently when he needs to recompile.

VI. SUMMARY AND CONCLUSION

A description of PROCESS III, a compiler-assembler for No. 1 ESS, has been given. The emphasis has been on the factors influencing the design of the compiler, the built-in PROCESS language and the facilities available for extending the source language.

The approach used in the design of the compiler has proved very useful, primarily because of the flexibility it has provided. Outstanding among the merits of this approach is the fact that there now exist several telephone-oriented procedures in a language understandable to programmers. This is not to say, however, that PROCESS III is the final answer to a "telephone language." The authors feel that it is accurate to say that PROCESS III has laid a solid foundation for a future PROCESS n.

VII. ACKNOWLEDGMENTS

To acknowledge all contributors to the design and implementation of a compiler at this late date would be very difficult. The art of designing compilers has matured considerably in recent years but not so dramatically that one can point to unique clear-cut breakthroughs. A new compiler is almost always a few new ideas mixed in with many old ones. So it is with PROCESS III. Thus the authors single out no specific articles in the literature — thanks are due to all workers in this field. We should

like to mention, however, S. H. Unger, under whose direction a predecessor compiler was built; N. S. Friedman, who programmed the macro definition and executive routines; R. E. Archer, who programmed the Compool and loader facilities; and W. C. Jones, under whose direction some early work was done on PROCESS III.

APPENDIX

A.1 *Realistic Example of a Telephone Function and Its Program*[3,4]

The example shown in Fig. 1 is a realistic subprogram taken from the coin charge sequence of No. 1 ESS. It shows the application of the general-purpose procedures in programming telephone functions. It also demonstrates the usefulness of programmer-defined procedures such as LINK, which links two call registers, and SZREG A, which generates a program to hunt and reserve an idle call register specified by A. The accompanying flow chart (see Fig. 2) shows the close correspondence between the procedures of the PROCESS language and the telephone functions depicted on the sequence chart.

A.2 *Detailed Example of Macro Definition and Macro Call*

Definition of a macro named MV:

```
DEFIN   MV      A,B,C                       Order
        *SET    OUT,I0,2                    1
        MK      "*S" "A"                    2
XYZ     *ADV    OUT,I0,1                    3
        *SF     $C,[T."I0"],R,0,2 $ 2,0     4
        KM      "*S" "I0"                   5
        *JB     XYZ                         6
        W"I0"   0,K                         7
        *JB     XYZ                         8
ENDEF
```

Purpose: to move the contents of A to B to C, $\cdots$. A may be an indexed or unindexed memory location. B may be an indexed or unindexed memory location or a register. Example: assume the macro call is

```
        MV      JACK,X,(JILL,Y)
```

where JACK and JILL are call store locations and X and Y are index registers.

Upon seeing this call, the compiler goes to the definition of MV. The steps taken by the compiler in expanding this macro call follow:

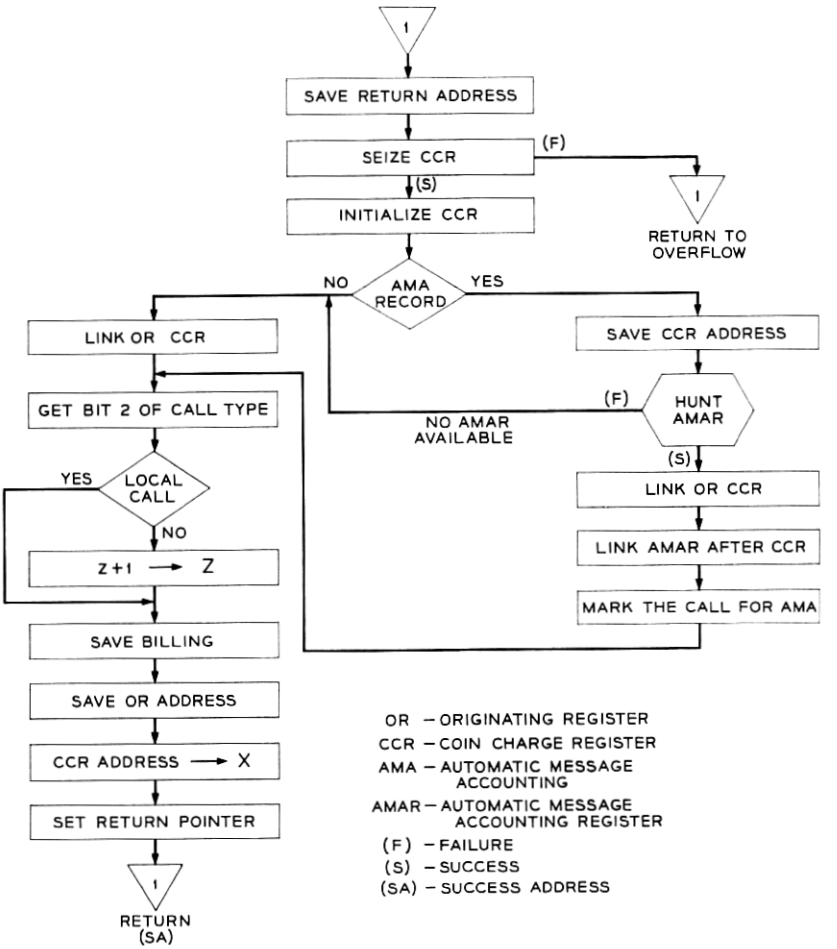

Fig. 1 — Subprogram from coin charge sequence.

(1) it sets I0 to JACK; (order 1)

(2) it produces:

MK JACK (order 2)

(3) it advances I0 to X; (order 3)

(4) it interprets the conditional,

$C,[T.X],R,0,2$

of order 4, which results in order 4 being interpreted as

*SF 2

```
PART ØF CØIN CHARGE PRØGRAM

                    7           EXTERN    ØRØVF2,AMSZNØ
                    9   S2X2    MØVE      J,I0
                   10           SZREG     CNC(K),ØRØVF2
                   11   XX      LØØP      1,V.S.CNC-1,1,Z
                   12           MØVE      0,(1,KA)
                   13   XX      ENDLØØP
                   14           IF        (RAMA,X),NE,V.RECRD,NØAMA1
                   15           MØVE      Y,T1
                   16           GØ*TØ     (AMSZNØ,,J)
                   17           GØ*TØ     NØAMA1
                   18           GØ*TØ     NØAMA1
                   19           MØVE      T1,Y
                   20           LINK      ØR(X),CCR(Y)
                   21           LINKA     AMA(Z),CCR(Y)
                   22           MØVE      1,(AMAR,Y)
                   23   MAIN    AND       (CTYP,X),2,Z
                   24           IF        (CZL,X),E,1,ØR(Z,1,Z)
                   25           MØVE      (BILL,X),(CHIN,Y)
                   26           MØVE      X,T1
                   27           MØVE      Y,X
                   28           SETPT     PT1
                   29           GØ*TØ     (T0,M)
                   30   NØAMA1  LINK      ØR(X),CCR(Y)
                   31           GØ*TØ     MAIN
0000064            32           END
```

Fig. 2 — Flow chart for sequence of Fig. 1.

(5) it arrives at order 7, which produces:

WX 0,K

(6) it returns to order 3; (order 8)

(7) it advances I0 to (JILL,Y); (order 3)

(8) it interprets the conditional, finding that (JILL,X) is not a register and skips to order 5;

(9) it produces:

KM JILL,Y (order 5)

(10) it returns to order 3; (order 6)

(11) it cannot advance I0 any further and therefore jumps out of the macro definition. (order 3)

A.3 *Example of an Output Listing*

Fig. 3 is a typical output listing.

Starting on the extreme left, the interpretations of the columns are:

(1) relocatable locations assigned to the object program

(2) a three-character console code corresponding to the operation part of the instruction

(3) the 37-bit octal representation of the instruction

(4) sequence numbers: each statement in the source program has such a number

```
                                PART ØF CØIN CHARGE PRØGRAM
                                6              EXTERN    ØRØVF2,AMSZNØ
                                7    S2X2      MØVE      J,TO
                       00000000      S2X2        SYN     X.
0000000   120 000500 0015621                     JM      TO
                                8              SZREG     CNC(K),ØRØVF2
0000001   010 040040 0000004                     T       YASCNC,,J
0000002   010 000040 0000001                     T       ØRØVF2
0000003   750 03364 00000000                     WK      0,Y
                                9    XX        LØØP      1,V.S.CNC-1,1,Z
                       00000014      9F0         SET     V.S.CNC-1
                       00000001      910         SET     1
0000004   730 00354 00000001                     WZ      1
                       00000005      XX          SYN     X.
                                10             MØVE      0,(1,KA)
0000005   042 120210 0000001                     EZEM    1,KA
                                11   XX        ENDLØØP
0000006   730 03754 00000001                     WZ      V.910,Z
0000007   430 07614 00000014                     CWR     V.9F0,Z
0000010   037 000166 0000005                     TCLE    XX
                                12             IF        (RAMA,X),NE,V.RECRD,NØAMA1
0000011   720 00350 00000010                     WL      M.RAMA
0000012   350 025642 0000016                     MK      A.RAMA,X,PL
0000013   742 04361 00000010                     CWK     V.V.RECRD•E.3
0000014   033 000146 0000060                     TCAU    NØAMA1
                                13             MØVE      Y,T1
0000015   130 000540 0015622                     YM      T1
                                14             GØ•TØ     (AMSZNØ,,J)
0000016   010 040040 0000002                     T       AMSZNØ,,J
                                15             GØ•TØ     NØAMA1
0000017   010 000040 0000060                     T       NØAMA1
                                16             GØ•TØ     NØAMA1
0000020   010 000040 0000060                     T       NØAMA1
                                17             MØVE      T1,Y
0000021   300 001400 0015622                     MY      T1
                                18             LINK      ØR(X),CCR(Y)
0000022   122 000510 0014004                     XM      QQC04
0000023   005 040022 0000003                     ENTJ    CØLINK
0000024   750 03364 00000000                     WK      0,Y
                                                 EXTERN  CØLINK
                                19             LINKA     AMA(Z),CCR(Y)
0000025   202 035010 0000000                     MB      A.Y4LI,Z
0000026   100 03440 40040000                     LM      M.Y4LI,Z,ES
0000027   720 00350 37777777                     WL      M.Y4LINK
0000030   350 031642 0000002                     MK      A.Y4LINK,Y,PL
0000031   132 030552 0000002                     ZM      A.Y4LINK,Y,EL
0000032   112 034452 0000002                     KM      A.Y4LINK,Z,EL
                                20             MØVE      1,(AMAR,Y)
0000033   750 00364 04000000                     WK      V.I•E.20
0000034   720 00350 04000000                     WL      M.AMAR
```

Fig. 3 — Typical output listing.

(5) source and object program symbolic statements: the indented statements were generated by the compiler and were not part of the source program.

REFERENCES

1. Keister, W., Ketchledge, R. W., and Vaughan, H. E., No. 1 ESS System Organization and Objectives, B.S.T.J., this issue, p. 1831.
2. Harr, J. A., Taylor, F. F., and Ulrich, W., Logical Organization of No. 1 ESS Central Processor, B.S.T.J., this issue, p. 1845.
3. Harr, J. A., Hoover, Mrs. E. S., and Smith, R. B., Organization of the No. 1 ESS Stored Program, B.S.T.J., this issue, p. 1923.
4. Carbaugh, D. H., Drew, G. G., Ghiron, H., and Hoover, Mrs. E. S., No. 1 ESS Call Processing, B.S.T.J., this issue, p. 2483.