The hardware-implemented high-level machine language for SYMBOL

by GILMAN D. CHESLEY and WILLIAM R. SMITH

Fairchild Camera and Instrument Corporation
Palo Alto, California

INTRODUCTION

Through the years since the specification and development of the first computer systems, machine instruction sets have undergone the least modification of any aspect of these systems. A process of evolutionary growth through accretion of new components has taken place in contrast to the revolution in the programming and systems areas. One can point to stack commands, indexing, and microprogramming, all developed several years ago, and then the list of new concepts in instructions runs out. Similarly, the number of papers in the literature relating to the hardware implementation of programming languages is sparse. This seems anomalous considering that reductions in the cost of logic associated with the development of integrated circuits allow the possibility of implementing much more complex functions in hardware than with past design practices.

The efforts as reported in the literature break down into several attacks. The first approach, contemporaneous with the development of procedural languages, was the NCR 304 as reported by Yowell\textsuperscript{1} and described in Sammet's "Programming Languages", and was an attempt to implement an Autocoder level in hardware, bridging the gap between machine languages and high-level languages. This direction of development has ceased, probably because of the explosive growth of high-level languages as the user language.

The second approach is that of directly implementing a severely restricted subset of an existing high-order language\textsuperscript{2,3,4} although none of these proposals seem to have resulted in hardware. A more recent effort that was consummated is the microprogramming of EULER as reported by Weber.\textsuperscript{5} Sometimes evolutionary steps are suggested\textsuperscript{6} to aid the system in the compilation and execution of languages without attempting to encompass the whole effort in hardware. The Burroughs family of machines growing out of the B5000 is a very practical realization of this approach.

Another direction, epitomized by ADAM\textsuperscript{7,8} is that of designing a unique high-order language to be hardware implemented with characteristics selected to ease the implementation problem. Of course, once outside the realm of general-purpose high-order languages, many languages become interesting candidates for hardware because of features designed with existing system structures in mind.\textsuperscript{9}

The approach taken with the SYMBOL system was to design a concise general-purpose language, well within the mainstream of contemporary language development, but with novel features as appropriate and with as few linguistic limitations as possible. The SYMBOL language can be characterized as a high-level, procedural, general-purpose, hierarchical language with variable-length processing and storage to allow the explicit representation of structures that are variable in size, shape, and field length, and without type and size declarations since conversion and space management are handled automatically. It is a highly efficient language for the user since all of the language features contribute directly to execution of the program rather than to easing compilation or memory management. Most of the language is directly implemented in hardware and is now running in a multi-processor virtual-memory environment.

SYMBOL DESIGN GOALS

The specification of the SYMBOL* architecture\textsuperscript{10} and the design of the SYMBOL* language was a con-\textsuperscript{*}Although some confusion might ensue from calling both the hardware system and the language by the same name, this seemed proper in this case where the language is hardware implemented and thus becomes a part of the specification of the system.
joint effort with each affecting the other. Very early in the project it was recognized that the language should be very concise to allow for a practical implementation. At the same time it was felt that it should be a general-purpose, procedural, “state of the art” language with no functional restrictions imposed on it to cater to its hardware implementation. Outside the language area, a design philosophy existed within the project that memory space management and data type testing and conversion could be done with hardware substituting for the traditional software handling of these areas. It was possible to combine these concepts by designing dynamic variable-length data space management and type conversion hardware and eliminating all size and type declarations from the language.

Similarly, by specifying the most general hardware capabilities possible, language limitations such as the number of subscripts or the size of a name were eliminated. In addition, the language is highly hierarchical, much in the spirit of ALGOL, further simplifying the translation and error checking hardware, as well as contributing to the ease of programming in the language. To minimize the conversion problem, all number processing is done in a decimal, floating point, variable field length internal mode. A special binary form exists for storing status information.

HARDWARE REQUIREMENTS

The major requirement imposed by the language on the hardware was complete variability of data size. Thus, both string and arithmetic processing had to be capable of operating on variable length fields (any length strings are allowed although numbers are limited to 99 digits). The system is capable of storing and accessing variable length data. This capability serves both the processing or execution aspect of the language as well as the compiling and system functions (e.g., variable length identifiers).

A subcripting capability implies structures, and, in keeping with the general philosophy, it was decided that no limitations should be imposed. Thus, structure handling hardware allows complete variability in size and shape, regular or irregular, and complete dynamicism whereby a complete structure may be assigned to a substructure or field of any other structure. This requires extensive and efficient space allocation and recovery mechanisms but removes these concerns from the language portion of the SYMBOL system.

LANGUAGE HIGHLIGHTS

The semantics of the language will be covered in the next section and the syntax is contained in Appendix A. This section will attempt to convey a flavor of the language by introducing some of the more novel features. First, however, the question “Why a new language?” must be considered.

In attempting to implement in the hardware an existing high-order, main stream language, FORTRAN, ALGOL, BASIC, and a subset of PL/I would probably be the chief candidates. The successful implementation of SYMBOL suggests that any of these languages could also be hardware implemented and that their rejection becomes more a matter of choice than of necessity. BASIC was felt to be too primitive. ALGOL lacks input/output facilities and thus cannot be considered a complete language. FORTRAN was a strong contender because of its ubiquity. As the first high-level language, it is far from being a state-of-the-art language. Also, as constructed it is quite machine dependent (consider the problems of COMMON and EQUIVALENCE in a variable field length environment). Thus, although FORTRAN is familiar to most of the programming world, it was felt that choosing this language would be both retrogressive and overly restrictive. Finally, PL/I was eliminated because full PL/I was considered too complex for efficient and economic hardware implementation and because it lacked the overall simplicity desired for an end user orientated language. In summary, it was felt that the existing languages were biased too much toward a multitude of data types and hardware influenced elements and that for hardware implementation a cleaner language with a high degree of generality was desirable.

Therefore, a new language, SYMBOL, was developed with only two design requirements: first, it should be free of declarations about the nature of an identifier and, second, structures should be literally representable in the program and the external medium. Other features that developed as the design evolved were the complete variability of field and structure size and the execution time dynamicism of this flexibility.

Considering first the conventional aspects of the language, SYMBOL is a general purpose, procedural language fully in the main stream of language development. Expressions can contain the normal set of arithmetic, comparison, and binary operators having standard precedence relationships. Operands can be of various degrees of indirection: literals, simple variables, subscripted variables with any number of simple or complex subscripts, or procedure calls. The input/output statements are flexible but simple, reflecting the ability of the system to do data space management. A standard conditional statement is supplied, but with the final group of embedded statements closed with an END to avoid the dangling ELSE problem. A very
flexible loop statement is available, allowing an unlimited number of loop clauses which have conventional default values if not present. It is a block structure language with default local identifiers and a floating global statement to move identifiers out to the enclosing block. Full procedural capability is available to the user with call-by-name parameters, automatic recursion as required, and a return statement for functional use which returns a generated value to the calling point.

Turning now to the more unusual characteristics of the language, its declaration-free nature is evidenced by the absence of size or type declarations or of implicit restrictions on field or processing length. This characteristic is built on the dual assumptions of automatic convertibility between data types and variable field length processing and storage. The premise here is that these are burdens best removed from the user and given to the hardware. Another salient characteristic of the language is its ability to represent structures literally. Thus, a vector containing the integers one through nine would be represented:

\[
\langle 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9 \rangle
\]

with the left and right angle marks and the vertical bar (called group and field marks, respectively) specifically reserved in the language for this purpose. Regular or irregular structures may be represented in this manner with any depth or shape and when stored in memory may be subscripted normally. For example, the three dimensional array A may be created by an initialization statement in the program:

\[
A \langle 11 \mid 12 \mid 13 \times 21 \mid 22 \mid 23 \times 31 \mid 32 \mid 33 \rangle;
\]

and a reference to A[3, 2] would access the number 32 and a reference to A[2] would access the vector \langle 21 \mid 22 \mid 23 \rangle. The declaration-free and dynamic features of structure handling can be illustrated by considering the execution of the following assignment statement:

\[
\]

giving in A:

\[
\langle 32 \times 21 \mid 22 \mid 23 \times 21 \mid 22 \mid 23 \times 31 \mid 32 \mid 33 \rangle
\]

There are essentially no limitations on this kind of structure manipulation. The same flexibility is allowed at the field level:

\[
A[1, 2, 1] = \text{STRING LITERALS ARE BETWEEN FIELD MARKS};
\]

giving:

\[
\langle 32 \text{STRING LITERALS ARE BETWEEN FIELD MARKS} \mid 22 \mid 23 \times 21 \mid 22 \mid 23 \times 31 \mid 32 \mid 33 \rangle
\]

Referencing a structure point that does not exist causes the necessary structure to be created and filled with null fields. Thus:

\[
B[5,3] = 444;
\]

causes in B:

\[
\langle 1\mid 1\mid 444 \rangle
\]

It was mentioned before that structures may be initialized in the language by writing the literal structure with its identifier as a statement in the language. A switching structure may be similarly initialized by preceding the structure identifier with the reserved word SWITCH which causes the structure fields to be interpreted as labels rather than as data. For example:

\[
\text{SWITCH} L \langle X \mid Y \mid Z \rangle;
\]

sets up the switching structure L and:

\[
\text{GO TO} L[2];
\]

causes a transfer to a program point labeled Y. This allows computed transfers to be performed by making the subscript of the transfer statement an expression to be executed.

A natural concomitant to VFL storage is VFL processing. All processing operations, numeric or string, are performed serially on variable length data fields with the restriction that numeric processing is limited to a 99 digit normalized fraction portion of a floating point number. The creation of irrational numbers is a special problem for a VFL system. Consider processing \(\sqrt{2}\) which would require infinite time and space to complete the operation. Two techniques are available in SYMBOL for controlling the length of numeric processing and thus processing speed and storage space. The first is a hardware register, referenced in the language by the reserved word LIMIT, which controls the truncation and rounding point of processing. VFL processing continues until the length of the fraction portion of the result reaches the value of the limit register, causing processing to terminate. The default value of the limit register is nine and it can be manipulated as an ordinary variable:

\[
\text{LIMIT} = \text{LIMIT}-4;
\]

would give LIMIT a value of five allowing processing to be faster, particularly for multiply and divide. Another method for controlling processing length is a precision mode whereby any number entering the system may be tagged with an EM (empirical) suffix indicating that the number is accurate only to the precision (length) supplied. Any number not so tagged is assumed to be exact; that is, of infinite accuracy and having implied low order zeros. The precision of the result of processing is equal to the least precision of either operand or the limit register. The result is empirical if either operand is empirical or if limiting
testing, replacement, insertion, etc. Two operators are string data and often used to convert between less features follow those suggested in Reference 11. To available, FORMAT and MASK, with the former operating on numeric data and the latter operating on dense code sets and the internal set. Many of the format includes format control operators to reconfigure data takes place. Consider the following examples:

<table>
<thead>
<tr>
<th>CONTROL FIELD</th>
<th>RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIMIT = 9; \frac{1}{3}</td>
<td>is .33333333 EM</td>
</tr>
<tr>
<td>LIMIT = 5; \frac{1}{4}</td>
<td>is .3333 EM</td>
</tr>
<tr>
<td>LIMIT = 5; \frac{1}{4}</td>
<td>is .25</td>
</tr>
<tr>
<td>LIMIT = 5; 1/4.00 EM</td>
<td>is .250 EM</td>
</tr>
</tbody>
</table>

Along with the normal operators, SYMBOL also includes format control operators to reconfigure data into a desired format using character-by-character testing, replacement, insertion, etc. Two operators are available, FORMAT and MASK, with the former operating on numeric data and the latter operating on string data and often converted between less dense code sets and the internal set. Many of the format features follow those suggested in Reference 11. To illustrate formatting, consider the following format control fields operating on the internally stored number 12345.6789:

<table>
<thead>
<tr>
<th>CONTROL FIELD</th>
<th>RESULTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>+9ZV</td>
<td>+12345</td>
</tr>
<tr>
<td>-9Z FD</td>
<td>12345.6789</td>
</tr>
<tr>
<td>10D.10D</td>
<td>0000012345.6789000000</td>
</tr>
<tr>
<td>ZZZZZZZZZZ.DD</td>
<td>12,345.67</td>
</tr>
<tr>
<td>ZBBBZBBBZBB</td>
<td>1 2 3 4 5</td>
</tr>
<tr>
<td>$<em><strong>C</strong><strong>C</strong></em>.DD</td>
<td>$***12,345.67</td>
</tr>
<tr>
<td>8B+D.FD_{10}+ND</td>
<td>+1.23456789_{10}+4</td>
</tr>
<tr>
<td>B-D.FD_{10}-DDMB</td>
<td>1.23456789_{10}04</td>
</tr>
<tr>
<td>B-F.DDMB</td>
<td>12345.6789</td>
</tr>
</tbody>
</table>

The last two control field examples are wired into the format unit and are used for automatic conversion from numeric to string type. The floating form is used if the absolute value of the exponent is larger than nine. The fixed form is used for all other cases. The full set of control characters is tabulated in Appendix B but a brief description will be included here to flesh out the above examples. Two replicators are used: a one or two digit number causing the following control character to be repeated that many times and the “enough” replicator (F) causing replication until the fraction or integer portion of the data field is used up. Literal insertion (of a dollar sign) is shown in the sixth example and “B” causes insertion of a blank and “\_” causes insertion of the exponent tag. Three leading zero suppression characters are illustrated: “N” causes suppression only, “Z” causes replacement by blanks, “*” causes replacement by asterisks, “C” causes a comma or the adjacent zero suppression character to be inserted. The signs cause both signs or minus only to be transferred. Decimal point placement is controlled with “.” and “V” with only the former causing inser-
items and can be subsequently deactivated after the debugging phase is complete.

A DESCRIPTION OF THE LANGUAGE

Components

Reserved characters are graphics having a unique interpretation to the system. They are also required as delimiters for multi-character names. Reserved characters may be categorized as follows:

Break Characters: space CR TAB
Grouping: () [] {}
Separators: , | ; : ≠
Operators: * + - / =

Reserved names are contiguous strings of alphabetic characters having a specific meaning to the system and interpreted as a single entity. In fact, if a large enough character set were available, all of the reserved names could be replaced by reserved characters, albeit at the cost of readability. They can be grouped as follows:

Operators: GREATER, GTE, EQUALS, NEQ, LTE, LESS, BEFORE, SAME, AFTER, AND, OR, NOT, ABS, JOIN

Statement: INPUT, OUTPUT, GO, IF, LOOP, BLOCK, GLOBAL, PROCEDURE, RETURN, ON, ENABLE, DISABLE

Statement Clauses: LINK, TO, FROM, THEN, ELSE, END, INTERRUPT, FOR, WHILE, FROM, BY, THRU

Non-reserved names, sometimes called identifiers, are alphanumeric strings starting with an alphabetic character, isolated by reserved characters, and optionally contain embedded spaces allowing multi-word names. They are used to identify variables, procedures, or label transfer points in the program.

Any string of characters between field marks is a string literal. That is, it is interpreted as data rather than a reference to data. Literals meeting valid number syntax need not be enclosed within field marks. The form for numbers is a string of digits with an optional decimal point and an optional exponent suffix consisting of the character “e” followed by an optional sign and one or two digits.

Expressions are language elements used to generate or modify data, unlike the simpler elements discussed in the previous paragraphs which reference or represent data. Expressions may be used any place that a name reference is proper (except, of course, as an assignment reference) and are formed out of the following elements: reserved character operators, reserved word operators, variable operands, string literal operands, numeric literal operands, parentheses, and other components to be covered later.

Because the SYMBOL language is declaration-free, variable names do not have an inherent data type associated with them. The issue of data type can be minimized but not ignored altogether because the standard operators of a high-order language partition into at least two categories: numeric and string (Boolean and binary data types can be considered a subset of string type and that is the approach taken with SYMBOL.) Thus, in SYMBOL, the operators require a data type rather than the variables and an error is called when an operator/operand mismatch occurs rather than when data is stored to a variable as with most high-order languages. The error is indicated only if the automatic conversion process between the two data types is not successful.

The operators are partitioned as follows:

Numeric Operands: ABS, +, - , * , / , GREATER, GTE, EQUALS, NEQ, LTE, LESS

String Operands: BEFORE, SAME, AFTER, JOIN, AND, OR, NOT

Numeric Result: ABS, +, - , * , / , GREATER, GTE, EQUALS, NEQ, LTE, LESS

String Result: GREATER, GTE, EQUALS, NEQ, LTE, LESS, BEFORE, SAME, AFTER, JOIN, AND, OR, NOT

The operators AND, OR, NOT require strings of 0/1 characters only. Only JOIN produces a full string while the comparisons produce a 0/1 string character and the binary operators AND, OR, NOT produce a string of 0/1 characters. Other than the previously discussed characteristics, all operations—operands and results—are variable field length.

Mathematical usage has established a convention with respect to precedence among the arithmetic operators. Thus, A+B*C means A+(B*C) rather than (A+B)*C. It has been customary with artificial languages to extend this notion to give all of the operators in the language a relative precedence. For SYMBOL,
the precedence relationships are as follows:

```
HIGH   ABS
   + (monadic), -(monadic)
   * , / 
   + (dyadic), - (dyadic)
JOIN 
comparison operators
      NOT
      AND
LOW    OR
```

Parentheses are used for vitiating the precedence relationships so that interpretations such as \((A+B)\times C\) can be made.

**Simple statements**

Statements in the language are basically used either to transfer or modify data stored in memory or to change and control the sequence of execution of these data statements.

The basic form of the assignment statement is:

```
identifier = expression
```

(Italics will be used throughout this paper to represent syntax elements. The full syntax of the language is given in Appendix A.) The value generated by the expression is stored in the variable represented by the identifier, replacing any previous value without regard for field size or data type.

The forms of input/output statements are:

```
INPUT[FROM expression,] L, identifier
```
```
OUTPUT[TO expression,] L, expression
```

(final bit of metalanguage: capital letter words represent the actual reserved word, brackets enclose an optional syntax element, the list operator \(L\) causes a list of one or more elements to be constructed and separated by the given punctuation mark).

The I/O statements cause data to be transferred between memory and external devices. If no TO/ FROM clause is supplied, the transfer takes place between memory and the default device. Lists of data are terminated by an end-of-record mark \((\neq)\). Each input datum in sequence is assigned to a different variable, and output data generated by each expression is placed on a different line (if a printing medium is being used). A STRING qualifier may be used which prevents automatic spacing action on output or structuring on input. A DATA qualifier requires/causes identifiers to be associated with the data being transferred. EX/EM qualifiers may be used on input to cause numeric data to be packed and tagged unless specified otherwise externally.

The basic form of the transfer statement is:

```
GO[TO] label
```

Any statement may be preceded by one or more identifiers followed by colons. This establishes the statement as a program transfer point and the identifier as a label. The transfer statement is the basic language mechanism to cause a transfer of the sequence of statement execution, when encountered, from one point in the program to another.

The basic form of the conditional statement is:

```
IF expression THEN body [ELSE body] END
```

where the expression must generate a Boolean result. The body is a list of zero or more statements separated by semicolons, and may contain conditional statements, allowing nesting of conditionals. Either the first body or the second (optional) body is executed, depending on whether the expression is true or false.

**Blocks**

The basic form of a block is:

```
BLOCK body END
```

where the body can contain one or more global statements:

```
GLOBAL L, identifier
```

The block mechanism is a device for preventing interference and interaction between identifiers when several program segments are brought together to form a single package. It causes all uses of an identifier to be local to the body of the block containing the identifier and thus have no connection to uses of the same identifier outside the block.

The GLOBAL statement provides the capability of establishing an identity between the same identifier in different blocks by causing all identifiers in the global list to become known to the program body immediately outside the block (i.e., global statements carry identifiers up one block level). This allows two or more different blocks packaged together in the same program to communicate via common identifiers contained within their global statements. The following uses of identifiers are automatically local and thus these identifiers may not appear in a global list: initialization, switch, label, on list, procedure declaration and procedure formal parameter. Although global statements only carry identifiers up one block level, blocks can be nested to any depth and enough global statements
must be supplied to boost identifiers up to a desired level.

**Procedures**

A procedure is a block with a name identifier. The basic form of a procedure is:

```
PROCEDURE identifier; body END
```

The reserved word `PROCEDURE` causes the procedure body to be established as a program block to be executed in place of and whenever the identifier occurs in the program. Referencing a procedure name causes an automatic transfer to the start of the procedure body, execution of that body, and an automatic return to the reference point when the body is completed. Procedures can only be entered by reference to the procedure name although they can be exited via a transfer statement. A procedure body is transferred around when encountered in-line. Procedures can be recursively entered when the procedure is referenced within its own block.

Procedures are often used in a functional manner—after execution of the procedure body there is a need to return a generated value to the calling point, replacing the procedure reference. The return statement provides this capability and has the following form:

```
RETURN expression
```

The return statement causes its expression operand to be calculated and returned to the calling point replacing the reference before terminating the procedure body. Assignment destinations may be returned to the calling point, the procedure terminates in a normal (non-functional) manner when the `END` is reached.

Parameters may be used to transfer constants, variables, expressions, or labels (syntactically called references) between the calling point and the body of a procedure by following the procedure name at the calling point with a list of these items surrounded by parentheses. These “actual parameters” are correlated one-to-one to a list of “formal parameter” identifiers immediately following the procedure identifier and preceding the procedure body. Whenever the formal parameter is encountered in executing the procedure body, it is replaced by its corresponding actual parameter which is executed in its place. This is referred to as a “call-by-name” parameter correlation. The complete form for procedures is shown in Figure 1.

The implementation of the parameter mechanism in the SYMBOL system allows a “procedureless parameter” to be defined. It is created by an assignment statement of the following form:

```
identifier = LINK reference
```

The parameter mechanism is activated whenever the identifier appears in any context in a program causing the reference to be executed in its place. The link assignment is a dynamic (execution time) action and has no connection with a procedure.

**ON block**

The on mechanism is a technique whereby a block of programming may be executed out-of-line by implicit rather than explicit (as with procedure) reference. The form of the on block is as follows:

```
ON L, identifier; body END
```

where the identifiers may refer to data variables, procedures, or labels. An on block is automatically executed by the following actions to its list of identifiers: after storing to a data identifier, before execution of a procedure, before transfer to a label. Since an on block is called implicitly it has no need for parameters or the return mechanism. After execution of the on block, processing continues at the point where the on block was activated.

The reserved word `INTERRUPT` may appear in the list of on conditions and refers to a hardware register that may be set by a number of external and internal conditions, thus causing that on block to be executed.
The disable/enable statements are used to control the deactivation and reactivation of on identifiers. The form is as follows:

\[
\text{DISABLE } L, \text{ identifier} \\
\text{ENABLE } L, \text{ identifier}
\]

When an identifier is disabled, the on block is no longer activated by a use of that identifier.

**Loop**

The loop statement provides a convenient mechanism for repeating the execution of a group of statements while varying an (optional) variable within the group until a given criterion is met which causes an exit from the loop. Loops may also be exited via a transfer statement within the loop. The form of the loop is as follows:

\[
\text{LOOP } [\text{identifier}] [L, \text{ clause}]; \text{ body END}
\]

where the clauses are as follows:

\[
\text{FOR } L, \text{ expression} \\
\text{[FROM expression][BY expression][THRU expression]}
\]

where the latter elements can be in any sequence and have default values of 1,1, infinity respectively. Either clause can be terminated by an optional "while" element.

The loop body is repetitively executed with the loop variable (if present) taking on the designated value, until the clause is satisfied or the while modifier becomes false. Then the next clause in sequence is executed, executing the body each time, until no more clauses remain, in which case the loop is terminated. Variables in the loop header may be modified at any time by actions within the loop body.

**Structures**

A structure is a group (or vector) of one or more elements consisting of basic elements or structures. Basic elements are separated by field marks (I) and groups are contained within group marks (< >). Structures need not be regular in size, shape, or data type and may vary dynamically. They may be nested to any depth. Depending on their program context they may contain data, labels, or expressions.

Structure points may be referenced on either side of an assignment statement or may be written with program segments contained within the group marks on the data side of an assignment statement. Explicit structures may be written and used as data. In this structure assignment case, each program segment within the literal structure is executed, a result is generated, and then the whole structure is assigned to the destination point. Using the structure marks, literal structures may be created at the input device and subsequently input to a variable. Similarly, structured variables may be directly transferred to an output medium.

Structures containing data or label identifiers may be given an initial value by writing them in the program, preceded by an identifier, as a separate statement. To initialize label structures, the reserved word SWITCH must also precede the structure identifier. Simple, non-structured variables may be initialized by the variable identifier followed by the literal data enclosed within field marks. Initialization takes place only once in a program when the initialization statement for a variable is first encountered. Subsequent passes through the program transfer around the initialization statements.

Full structures are accessed by reference to the structure identifier without subscripts. Substructures are accessed by enough subscripts to select the desired substructure level. Most substructure references will be with enough subscripts to select the field level. Subfields are accessed by a bound pair (expressions separated by a colon) as the last subscript after the field level. Characters are selected from the field starting at the left value of the bound pair and continuing to the number of characters indicated by the right value of the bound pair. All substructures above the field level are complete and valid structures and thus cannot be used in expressions (automatic vector or array operations are not included as part of the hardware language). Basically, structures can only be assigned or transferred between the external medium.

There are essentially no limitations to structure manipulations. Fields or vectors can be extended, contracted, or removed by assignment. Fields or vectors created by reference are filled with nulls. The only restriction is that data cannot be destroyed by referencing below an existing field level although it is legitimate to destroy it by assignment. The reserved word IN tests whether a particular structure point referenced by a subscripted variable exists, returning either a true or false value. Oversubscribing by an in reference does not cause null structure to be generated.

A switch structure containing labels can be the subscripted variable operand of a transfer statement, allowing a computed go to be performed.
Two techniques are available to control the length of variable field length arithmetic processing. The first is a precision limiting register, acting as a hardware variable in the language and referenced by the reserved word LIMIT. The default or standard value is 9 causing all numeric processing to cease at a maximum value of 9 digits of precision. Any other positive value up to 99 may be assigned to the limit register.

The second technique tags (directly or with an input qualifier) all numbers entering the system with an EM (empirical) suffix which indicates that they are accurate only to the number of digits (precision) supplied. All numbers not specified as EM are tagged by the system as EX (exact) and these numbers are assumed to possess an infinite string of low order zeros. These tags affect processing in an obvious and natural way and combine with the limit register to give precision controlled processing with a size override for the sake of efficiency. Any exact calculation terminated by the limit register produces an empirical result (when limiting takes place an internal status indicator is set which may be accessed and reset by the reserved word LIMITED). With this system, when the accuracy of an algorithm is not taken into account, meaningless results will be indicated by the precision shrinking to zero.

Many times a standard data format, automatically provided by the system will be sufficient. Often, however, an application demands a particular format, and two operators are provided by the SYMBOL language to control format. The FORMAT operator reconfigures numeric data and the MASK operator reconfigures string data. They both have a control operand and a data operand with the characters of the control operand operating on (selecting, inserting, replacing, etc.) characters of the data operand. Many of the characters of the format operator are concerned with check and payroll formats while many of the mask characters cause conversion between special data modes. The full set of format and mask control characters is summarized in Appendix B. To further aid in format control, the output statement has a STRING mode option which suppresses the normal carriage return spacing features associated with the output statement.

The reserved word TRAP is a separate language statement which, when encountered, causes the hardware Translator to interrupt the translation process and turn system control back to the master hardware scheduler. At this point, if desired, a system program would be executed. The reserved word SYSTEM performs a similar function for the execution unit (the Instruction Sequence), causing a system interrupt when encountered. It has no effect on the Translator, passing through the translation process without change.

The Privileged Memory Operations (PMO) are language statements causing actions identical to the actual hardware memory operations of the system which are described in the companion paper on system architecture. They are of the nature: "fetch the word at the address I give you and follow any indirect links to give me the next address in sequence" or "store the word I give you into the address I give you and assign new space and link it in if no following space exists and return the next address to me." The operands of the PMO's, as seen by the user, are hex coded character strings (so that they can be manipulated as valid data) of the data and address fields and are automatically packed and unpacked by hardware. Since all the status of the system is ultimately kept in memory, the PMO's have the ability to control and mimic all aspects of the system, albeit at a slower rate than hardware can accomplish the same task.

Since SYMBOL is a time-sharing system, it is necessary to prevent programs from inadvertently interfering with each other. As a hardware language, the protection is inherent: until now, there are no language mechanisms by which one program can interfere with another since all data is manipulated under symbolic names rather than as memory addresses. There is no danger. Thus, they are designed as privileged operations which can only be used under special restrictions. Basically, one must either be a system programmer or be using a system procedure to utilize the PMO's.

CONCLUSIONS

Hopefully, the SYMBOL language has been described in enough detail to give the reader familiar with programming languages the flavor of the language and to give the language expert enough information to actually use the language. As yet, few performance figures are available since the project is just entering the operation and evaluation phase, although preliminary measurements indicate potential compiling rates of hundreds of thousands of statements per minute. Of course, per-
APPENDIX A

SYMBOL SYNTAX

digit ::= 0|1|2|3|4|5|6|7|8|9
letter ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z

character ::= any character except _
break-char ::= _ carriage return, tab or space
identifier ::= letter[letter|digit|break-char]...
label ::= identifier

default-number ::= digit[.]...digit
exponent-part ::= e±[+]digit[.]...
number ::= default-number|exponent-part|default-number exponent-part

string ::= sequence of zero or more of any characters except < > | and _
field ::= string-number|string

data-field ::= ["""] field

named-data ::= identifier [data-field|data-structure]
dc ::= ["""] field
data-structure ::= dc [data-structure dc]...
lab ::= ["""] label
label-structure ::= <(label-structure la)...

as ::= ["""] exp

assignment-structure ::= <assignment-structure as>...

subscription ::= identifier ["""] exp

self-defined-data ::= named-data...

initialization-stm ::= named-data

switch-stm ::= SWITCH identifier | label-structure

for-clause ::= FOR la [exp WHILE exp]

step-clause ::= {FROM exp|BY exp|THRU exp|WHILE exp}...

loop-header ::= LOOP [designator|la|for-clause|step-clause]

loop-stm ::= loop-header body END

on-element-list ::= la|identifier|label|INTERRUPT

on-header ::= ON on-element-list;
on-control-stm ::= DISABLE|ENABLE on-element-list

return-stm ::= RETURN [reference]

procedure-header ::= PROCEDURE [identifier|"""]

conditional-stm ::= IF exp THEN body ELSE body END

scope-stm ::= GLOBAL identifier

block ::= BLOCK [on-header|procedure-header] body END

...

APPENDIX A

SYMBOL SYNTAX

Performance in the SYMBOL system is an admixture of two opposing factors, hardware implemented functions and an extremely dynamic language, and it will be very difficult to separate these two contributions. It is probably true that the SYMBOL language, if compiled and executed on a conventional architecture, would be as slow as any other high-level language having comparable execution time dynamicism coupled with automatic data space management. Default PL/I would probably provide a fairly relevant basis for prediction. As implemented in the SYMBOL hardware, however, any task requiring the variable length processing and storage or the dynamic structure features of the language should show a considerable performance gain over conventional software/hardware systems.

Early experience within the SYMBOL project has shown the language to be an extremely simple one to learn and use, considering its powerful capabilities. The use of structures seems to indicate that their great flexibility and dynamicism completely eliminate the need to develop exotic strategies for data base management. For example, hash sorting buckets in SYMBOL can simply be vectors of an array since vectors automatically expand as the subscript increases. Since collisions are of little consequence, scatter storage techniques need not be complicated. Little practical experience has been gained with the precision controlled processing mode although it is expected to be a boon to numerical analysts as well as a source of surprise to less sophisticated users who have never considered the question of numerical accuracy since conventional systems tend to mask accuracy loss.

SYMBOL is a break with the existing trends in language development, as epitomized by ALGOL 68 and full PL/I. That is, the SYMBOL system hardware takes over from the user and performs the complete memory management task including all space allocation and recovery, structure expansion and contraction, and virtual memory paging between disk and main memory. The other languages do the opposite by
Appendix B

giving the user more and more control over space management and, in fact, require a fairly high level of expertise in arranging and programming problems for efficient use of space. Herein lies the sticking point: will future users of high-level language systems want more or less control over the mechanics of their data base? The SYMBOL philosophy is that user data space management will go the way of manual transmissions on automobiles: fine for special purposes but the average user will be willing to give up some performance for ease-of-use.

In a way, the situation is analogous to the introduction of high-level languages with FORTRAN. At the time there was great concern within the FORTRAN project that the system must have an execution time performance roughly equivalent to a machine language for efficient use of space. Herein lies the sticking point: will future users of high-level language systems want more or less control over the mechanics of their data base? The SYMBOL philosophy is that user data space management will go the way of manual transmissions on automobiles: fine for special purposes but the average user will be willing to give up some performance for ease-of-use.

REFERENCES

1 E C YOWELL
A mechanized approach to automatic coding
Automatic Coding Journ Franklin Inst Monograph No 3
Phila Pa April 1957 pp 103-111

2 J P ANDERSON
A computer for direct execution of algebraic languages
Proc EJCC Vol 20 1964 pp 184-193

3 A J MELBOURNE J M PUGMIRE
A small computer for the direct processing of FORTRAN statements
Computer Journ Vol 8 No 1 April 1965 pp 24-27

4 T R BASHKOW A SAISON A KRONFELD
A procedure-oriented machine language
IEEE Trans Elec Comp Vol EC-13 No 4 Aug 1964

5 H WEBER
System design of a FORTRAN machine
IEEE Trans Elec Comp Vol EC-16 No 4 Aug 1967

6 W M McKEEMAN
System design of a FORTRAN machine
IEEE Trans Elec Comp Vol EC-13 No 4 Aug 1964

7 A P MULLERY R F SCHAUER R RICE
A small computer for the direct processing of FORTRAN statements
Proc SJCC Vol 23 1963 pp 367-380

8 A P MULLERY
A microprogrammed implementation of EULER on the IBM System/360 Model 30
Comm ACM Vol 10 No 9 Sept 1967 pp 549-558

9 W R SMITH et al
System design of a cellular APL computer
Proc SJCC Vol 23 1963 pp 367-380

10 R MORRIS
A small computer for the direct processing of FORTRAN statements
Proc SJCC Vol 23 1963 pp 367-380

11 D E KNUTH
A proposal for input/output conventions in ALGOL 60
Comm ACM Vol 7 No 5 May 1964 pp 273-283

12 D E KNUTH
A computer for direct execution of algebraic languages
Proc EJCC Vol 20 1964 pp 184-193

APPENDIX B

<table>
<thead>
<tr>
<th>CODE</th>
<th>ACTION TO RESULT</th>
<th>CHM. LETER</th>
<th>ORI RESULTS</th>
<th>INFLUENCE</th>
<th>RESTRICTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>F</td>
<td>Literal string a inserted unless time as n</td>
<td>0,1 n</td>
<td>N</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

*Appendix Classes

- Standard Format Conversion:
- Fixed-exponent one digit or
- Floating-exponent two digits

NOTE: Standard Format Conversion:
- S-FD.FDMI (Fixed - exponent one digit) or
- F-FD.FDMI (Floating - exponent two digits)