Chapter 30

# The SYMBOL Computer
# SYMBOL: A Large Experimental System Exploring Major Hardware Replacement of Software[1]

*William R. Smith / Rex Rice / Gilman D. Chesley / Theodore A. Laliotis / Stephen F. Lundstrom / Myron A. Calhoun / Lawrence D. Gerould / Thomas G. Cook*

## Introduction

The SYMBOL system is the result of a major developmental effort to increase the functional capability of hardware. Part of the charter of the broad based project was to reexamine the traditional division between hardware and software, to reexamine the respective roles of program instruction and data storage, and to reduce the overall complexity and cost of computing [Rice and Smith, 1971]. In order to adequately evaluate the concepts that had been developed it was concluded that an experimental, usable, real system must be built. The SYMBOL system, now operational, is the embodiment of this effort.

The system was developed in an environment with hardware and software design considered in common. Virtually no one associated with the project could refer to himself as a hardware or software specialist exclusively. As an example, the logic design of the field process units was done by an individual with a basic programming background [Mazor, 1968]. The wire routing automation was developed by an engineer who was formerly a pure logic design specialist.

Even before the system became operational much had been learned about the practical aspects of building highly capable hardware. No claim is made that SYMBOL represents an optimum general purpose, time-sharing, multiprocessing system. In contrast, numerous simplifying assumptions were made in the system where they did not serve the goals of the project. Certain modularity restrictions are examples of this. It is claimed that SYMBOL represents a significant advance in systems technology and provides the foundation for a significant reduction in the cost of computing. As the system moves into an intensive evaluation phase it should prove to be a real asset for advanced systems research.

This paper represents an overview of the SYMBOL organization. An attempt is made to give simplified examples of various key features in contrast to a broad brush treatment of many topics.

## Gross Organization

The system has eight specialized processors that operate as autonomous units. Each functional unit is linked to the system by the Main Bus. See Fig. I. Consider some of the features of the system and their relationship to the gross processor organization as outlined in the following sections.

### *Dynamic Memory Management*

Direct hardware memory management is perhaps the most unique feature of the SYMBOL system. The memory management centers around a special purpose processor called the Memory Controller (MC). The MC effectively isolates the main memory from the main bus and the other processors and in turn provides a more sophisticated storage function for the various processors. In contrast to simple read/write memory operations the MC has a set of fifteen operations that are available to the

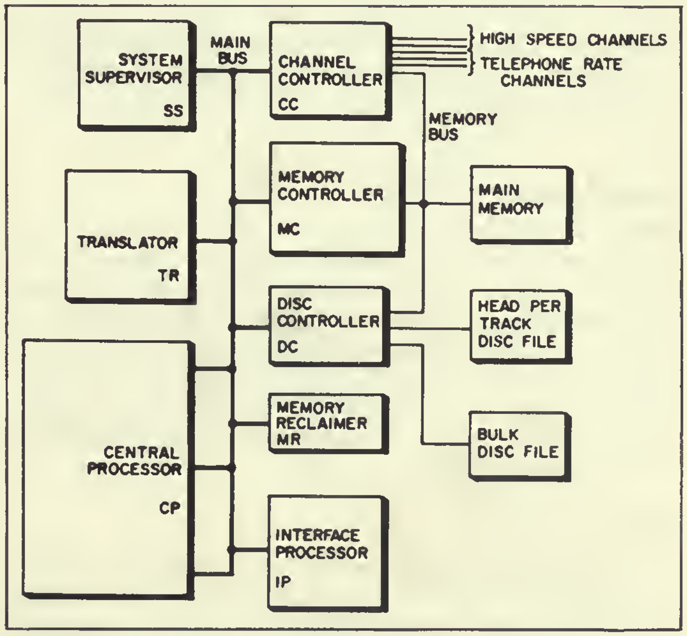

**Fig. 1. Gross block diagram of the SYMBOL system.**

[1]*AFIPS SJCC*, 1971, pp. 601–616.

other processors of the system. The MC is a special purpose processor that allocates memory space on demand, performs address arithmetic, and manages the associative memory needed for paging. The Memory Reclaimer (MR) supports the MC by reprocessing used space to make it available for subsequent reuse. It is a separate unit to allow the task to be performed using a low priority access to the memory.

### *Direct Compilation*

The Translator (TR) accepts the high level SYMBOL language [Chesley and Smith, 1971] as input and produces a reverse Polish object string and name table suitable for processing by the Central Processor (CP). The TR performs the direct hardware compilation using only a small table of about 100 words stored in main memory.

### *Dynamic Variable Field Length*

Within the Central Processor all field processing is done with dynamically variable field lengths. All alphanumeric string processing is done by the Format Processor (FP) while all numeric processing is done by the Arithmetic Processor (AP). The resources of the MC are used extensively by the CP in handling the storage of data.

### *Dynamically Variable Data Structures*

Complete variability of data structures is allowed. They can change size, shape, and depth during processing. Within the CP the Reference Processor (RP) manages the storage and referencing of all data arrays and structure. The MC functions are used extensively by the RP.

### *Time-Sharing Supervision*

The System Supervisor (SS) is the task scheduler for the system. All transitions from one processing mode to another are handled by the SS. Queues are maintained for all of the time-shared processors. The SS executes two important hardware algorithms, job task scheduling and paging management. A real-time clock is used in the process of rationing out critical resources such as central processor time. The SS also performs key information transfers needed to tie hardware algorithms into software system management procedures.

### *Direct Text Editing*

The Interface Processor (IP) and Channel Controller (CC) perform the input/output tasks of the system. The IP has ability to handle general text editing in support of interactive communication via a special terminal. Input/output and text editing do not use the CP resources.

### *Virtual Memory Management*

When the MC detects that a page is not in main memory it notifies the requesting processor and the system supervisor. The SS then utilizes a paging algorithm to supply the appropriate disk transfer commands to the Disk Controller (DC). Each user of memory must, upon receiving a page-out response, be able to shut down and save its current state and status and restart after paging is complete.

## System Configuration

The system has a small complement of peripheral and storage equipment associated with the main frame. This complement of equipment has proven sufficient for the experimental purposes of the system. The main memory is an 8K word × 64 bit/word core memory with a cycle time of 2.5 microsec. It is organized into 32 pages with 256 words/page. The main paging memory is a small Burroughs head-per-track disk divided into 800 pages. The bulk paging memory is a Data Products Disk-file organized into 50,000 pages.

The Channel Controller is designed to handle up to 31 channels. This low limit was deemed sufficient for evaluation of the experimental system. As of this writing one high speed (100,000 bits/sec. effective data rate) channel and three phone line (up to 2400 baud) channels have been implemented. More can be added during the evaluation phase.

The main frame contains about 18,000 dual in-line CTμL components. Its physical properties are described in other papers [Cowart, Rice, and Lundstrom, 1971; Smith, 1968]. In order to get a relative measure of the size of the various autonomous processors a chart is given in Fig. 2.

## System Communication

The main bus of the system is a time-shared, global communication path. It uses the special properties of the CTμL family in its implementation [Cowart, Rice, and Lundstrom, 1971; Smith, 1968]. The bus contains 111 parallel lines. They are distributed as follows:

| | |
|---|---|
| Data Bus | 64 |
| Address Bus | 24 |
| Operation Code Bus | 6 |
| Terminal Identification Bus | 5 |
| Priority Bus | 10 |
| System Clock | 1 |
| System Clear | 1 |

Four types of bus usage are available. They are:

Processor to MC transfers

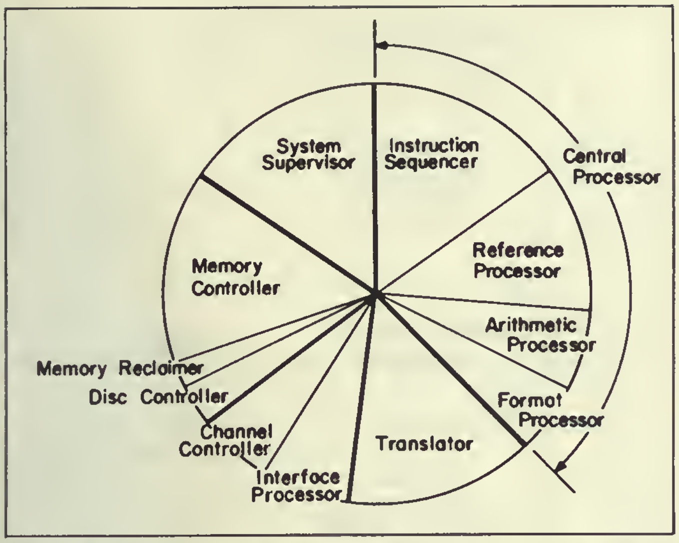

Fig. 2. Breakdown of the SYMBOL hardware showing the relative sizes of the various processors.

MC to Processor transfers

Processor to Processor transfers

Control exchange cycles

The basic information transfers are priority sequenced. The priority bus indicates the desired bus usage for the following cycle; if a unit desires to use the bus it raises its priority line and then checks the priority bus to see if there are any higher priority requests. If not it uses the bus on the following cycle.

Control exchange cycles are used to communicate control information between the SS and the various processors over the data and address buses. See Fig. 3. During a control cycle the data and address bus lines have preassigned uses. Certain lines are used to start the CP. Others indicate the completion mode for the TR. During a given cycle any combination of the paths can be used. The SS has autonomous interface control functions that are used to communicate with the processors during control cycles so that more than one control signal can be transmitted during a given cycle.

## Memory Organization

### *Virtual Memory*

The SYMBOL memory is organized as a simple two-level, fixed page size virtual memory [Kilburn et al., 1962]. The page has 256 words with each word having 64 bits. Virtual memory is accessed by a 24 bit address with 16 bits used to select the page and 8 bits to select a particular word within a page. See Fig. 4.

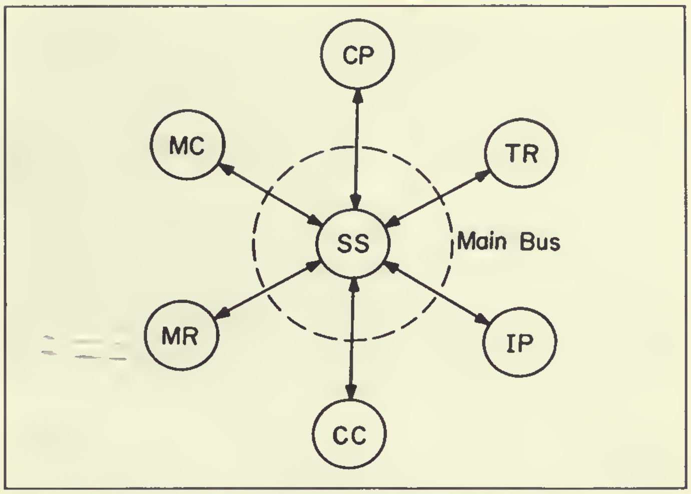

Fig. 3. Use of the main bus for control exchange cycles.

The main memory for the experimental system is logically divided into 32 pages. The relative portion of the address is used directly while the page number accesses an associative memory which in turn supplies the current page address in main memory.

The associative memory has one cell for each page in the main memory. By providing an associative memory tied to the main memory the individual processors need not be concerned with the location association process. This provides a significant reduction in the logical complexity of the processors even though it may lead to slightly more overall electronics.

The paging disk memory has fixed assignment of page locations.

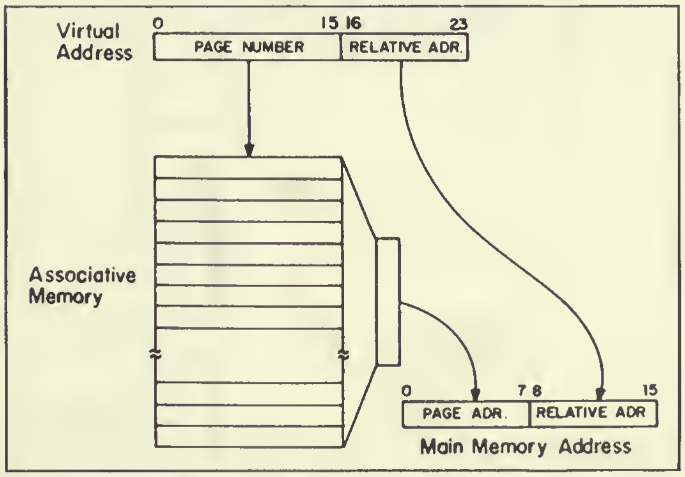

Fig. 4. The simple two level addressing structure for the virtual memory.

See Fig. 5. A page is brought into an available location in main memory upon demand. When it is purged back to disk it is transferred back to the same location on disk. (The return transfer is omitted if the page was not changed in main memory.)

The main memory organization is shown in Fig. 6. The first page is used for system tables. This includes a reserved word table for the translator, a software call table, and the control words for memory allocation and queuing. The next set of pages are used for storing the control words of the various terminals or users on the system. Each active terminal has 24 words of control information in low core. Much of the control information could have been placed in virtual memory as would certainly be required for a system with a larger channel capacity. As a simplifying restriction for SYMBOL all channel tables were placed in main memory.

The input/output buffers for the various active channels are also held in core. The buffers require 16 words per active channel. Variable buffer sizes although possible were not implemented.

The remainder of main memory is available for virtual memory buffering. Paging is managed by the hardware with the page selection for purging under the control of the system supervisor. The algorithm is a very flexible parameterized process that allows most of the conventional paging algorithms to be executed. The parameters are maintained for each terminal so that the paging dynamics can be tailored on a terminal by terminal basis.

The virtual memory organization is quite simple for SYMBOL in contrast to the more common segmentation schemes [Glaser, Couleur, and Oliver, 1965; Corbato and Vyssotsky, 1965]. The

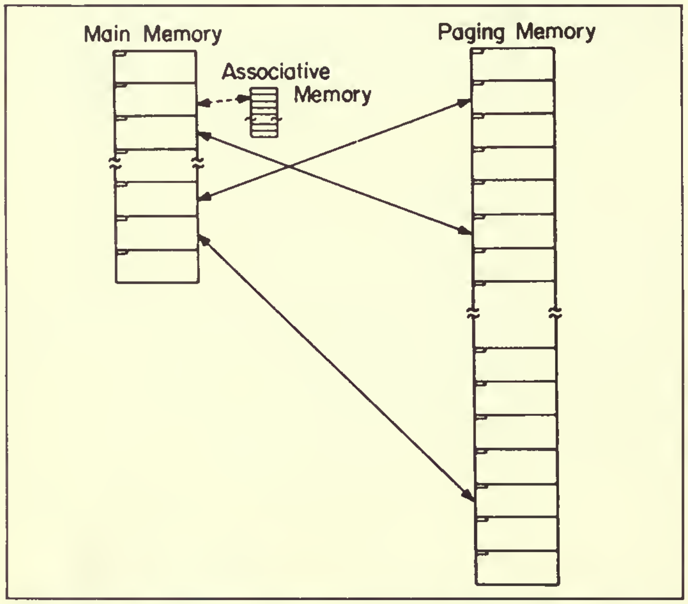

Fig. 5. Virtual memory organization showing the fixed location of pages in the paging memory.

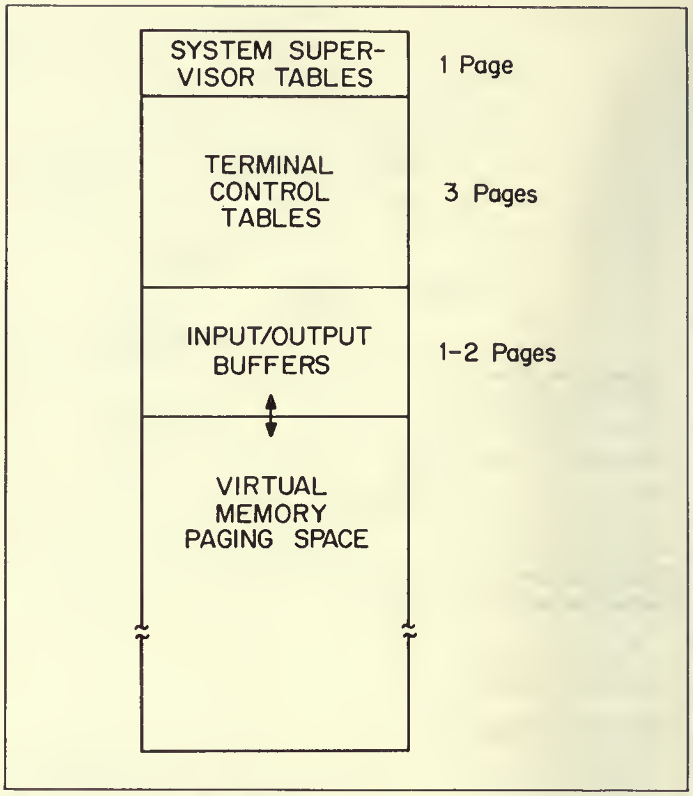

Fig. 6. Layout of main memory.

primary difference that allows the simplified approach to be taken in SYMBOL is that contiguous addressing above the page level is not needed. All users and channels share the same virtual memory space. The 24 bit address space is thoroughly used. With space allocated only upon demand and with no restriction on a scrambled assignment of pages to users it is anticipated that 24 bits will be sufficient for many more than the 31 possible terminals. If file space is needed beyond the 24 bits of address space it can be addressed via special block input/output transfers.

### *Page Lists*

Pages are associated together with the use of linked page lists. Pages available for assignment are maintained on an available page list. As each user needs space a user page list is started by transferring a page from the available page list to the particular user. A control word is established at this time as a focal point for all future page list management for the user. As more space is needed pages are added to form a variable length storage area for general purpose usage. See Fig. 7.

A given user may have more than one page list. Typical page list usage for a terminal would be one page list for program source

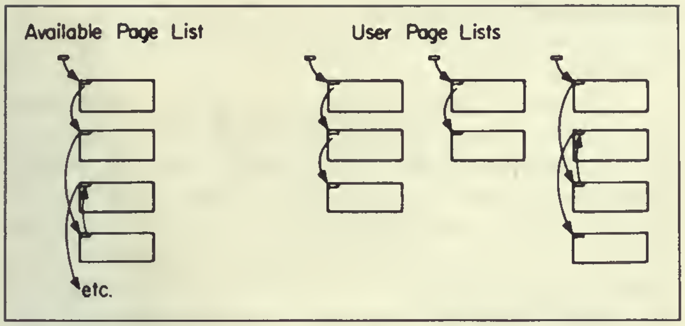

Fig. 7. Simplified page list structure within the virtual memory.

text, another for the compiled object program, and a third for data variable storage. Other page lists are used for long or short term file storage.

Page lists grow monotonically as space is needed. When an entire list is no longer needed it is given back to the system by returning it to the available page list.

### *Page Organization*

In order to handle non-contiguous address space a certain amount of the storage space must be devoted to linking or association data overhead. In SYMBOL about 11 percent of the storage space is for overhead bookkeeping.

Pages have three distinct information regions as shown in Fig. 8. The first region called the page header is used to maintain the page lists and manage the space within the page. The second region is a set of 28 words. The third region is a set of 28 groups

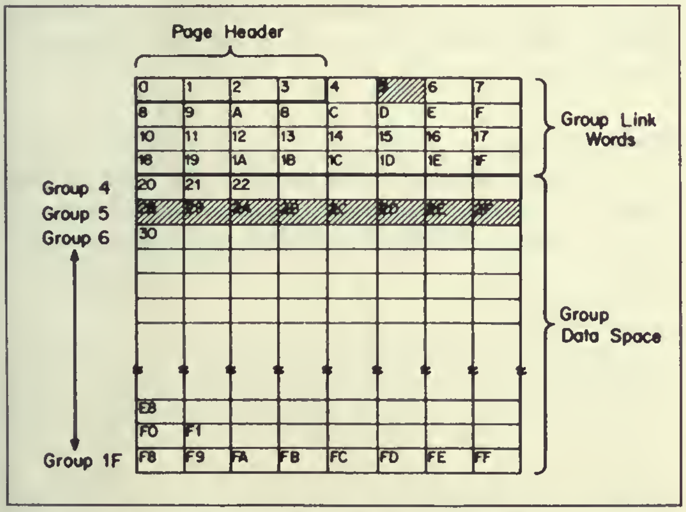

Fig. 8. Page organization showing group and link word layout where addresses are given in HEX notation.

with each group containing eight words. Each group has a corresponding group link word associated by a simple address mapping. Consider in Fig. 8 word 5 and the corresponding group 5. Data is stored in words 28 through 2F. This eight word group is the fundamental quantum of space allocation. It is the smallest amount of memory space that is assignable to a given purpose.

When data is needed for some purpose groups are assigned. For example, if six words were needed to store a data vector one group would be assigned. If space for a vector of 14 one word items were needed two groups would be assigned. Variable length information areas are developed by chaining together these basic units of storage.

### *Information Strings*

Variable length lists of storage locations are used for general information storage in SYMBOL. They are logically contiguous memory cells but not necessarily physically contiguous cells.

Consider a typical variable length information string in Fig. 9. Data space for 24 words of information is tied together by way of the associated group link words. If access to the start of the string is known it is possible to follow the entire string by accessing the corresponding group link word each time the end of a group is encountered. It is also possible to traverse the string backwards by using the back links also stored in the group link word.

Each processor uses the variable length storage service of the memory controller (MC) without cognizance of the address sequence that is involved. For example, when a processor needs space to store a vector of data fields an Assign Group (AG) command is sent to the MC along with a tag specifying a page list with which the string is to be associated. The MC then selects an available group from the page list and returns the address of the first word of the group to the requesting processor. When the

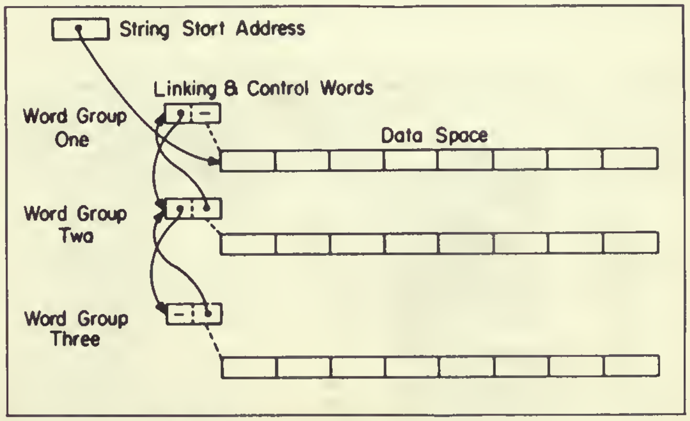

Fig. 9. Structure of a variable length string.

processor is ready to store a word it transmits the data and the address previously assigned to the MC along with the command Store and Assign (SA). The MC stores the word and generates the address of the next available word. When the end of the group or string is encountered the MC assigns another group and links it into the string.

In the string storing process the requesting processor receives addresses from the MC and resubmits them to the MC at a later time for future extension of the string. All address arithmetic is done by the MC. Consider the example in Table 1. The first five commands result in the words A, B, C, and D being stored in a string beginning with word A.

To reaccess the string the original start address A is submitted to the MC with the Fetch and Follow (FF) command. The data in cell A is returned along with the next address in the logical sequence. When the string is no longer needed a Delete String (DS) command along with the string starting address is submitted to the MC. The entire string is then placed on a space reclamation list. The Memory Reclaimer processor scans the space reclamation lists of the various page lists during idle memory time and makes the groups of the deleted strings reassignable.

The basic memory usage process deals with variations of the AG, SA, FF, and DS operations. Eleven other memory commands are available to give a full memory service complement.

Space utilization efficiency was an important aspect of the SYMBOL memory design. Studies have been made into the optimum size of the space allocation group [Smith, 1963]. The trade-offs center on balancing the linking overhead cost and the unused group fragments cost. The overhead cost is compensated by the allocation on demand approach. In most machines, fixed size data arrays are allocated to their maximum needed size. When the average array usage is considered a substantial amount of demand allocation overhead can be afforded before approaching the normal excess fixed allocation space usage.

**Table 1 Simplified Example of a Memory Usage Sequence**

| *Mnemonic* | *Operation* | *Address to MC* | *Return address* | *Data to MC* | *Return data* |
|---|---|---|---|---|---|
| AG | Assign Group | ....... | a | ....... | - |
| SA | Store & Assign | a | b | A | - |
| SA | Store & Assign | b | c | B | - |
| SA | Store & Assign | c | d | C | - |
| SA | Store & Assign | d | e | D | - |
| FF | Fetch & Follow | a | b | - | A |
| FF | Fetch & Follow | b | c | - | B |
| FF | Fetch & Follow | c | d | - | C |
| FF | Fetch & Follow | d | ....... | - | D |
| DS | Delete String | a | ....... | - | - |

## Information Forms

### *Data Fields*

Two basic data types are defined in the system, namely string and numeric fields. The string field is characterized by a special String Start (SS) character followed by a variable length set of ASCII alphanumeric characters terminated by a special String End (SE) character. This illustrates perhaps the most significant aspect of all SYMBOL data representations. The type and length of the datum is carried with the datum. The instruction code is independent of the dynamic attributes of the data.

The second data type is a variable length, packed decimal, floating point number. The numeric form also carries a designator of the class of precision. Numbers may be *exact* with an infinite number of trailing zeros implied or they may be *empirical* implying that all following digits are unknown and cannot be assumed present for calculation purposes. Like the string field all attributes of the datum are carried by the datum itself.

As a simplifying hardware design decision other forms of data fields were not implemented. It is straightforward to extend the SYMBOL concepts to packed variable-length binary strings, fixed length binary numerics, variable length binary numerics, etc. In any of these cases the datum must carry a type designator and an explicit or implicit designation of field length.

### *Source Programs*

Source programs are special forms of string fields. They are variable length ASCII character strings with delimiters defining length and type. They can be treated as data fields during preparation and then later used as program source for compilation. Source procedures may be assembled into libraries of various forms as long as they retain the string field attributes for compilation purposes.

### *Data Structures*

Data structures are defined as a variable length group of items where an item may be a string field, a numeric field, or another group of items. With this recursive definition a structure could be a vector, a matrix, or an irregular array. There is no limit to the depth or size of an array providing a field or a group does not exceed the size of main memory during execution.

Consider the example of a simple vector shown in Fig. 10. The special graphics <,|, and> have been introduced for representing field boundaries and groupings of fields. They are used to define the extent of variable length fields and referred to as left group marks, field marks, and right group marks, respectively. In memory the string fields are delimited by String Start (SS) and String End (SE) characters. Another special character called the End Vector (EV) code terminates a group of fields. The storage

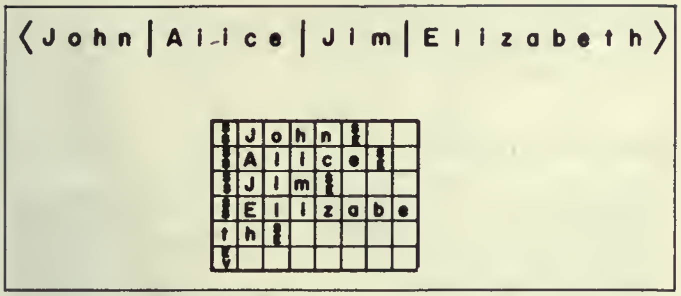

**Fig. 10. A vector of string fields and the corresponding representation of the data in memory.**

representation in Fig. 10 shows a series of string fields followed by a special End Vector (EV) code which again is a length indicator with the data. The string fields are aligned to start on machine word boundaries. In the case of Elizabeth two machine words are needed to store the field.

In Fig. 11 the matrix representation is similar to the vector example except that two levels of vectors exist. The definition of a structure could be restated as a variable length group of items where an item may be a string field, a numeric field, or an address link to another group.

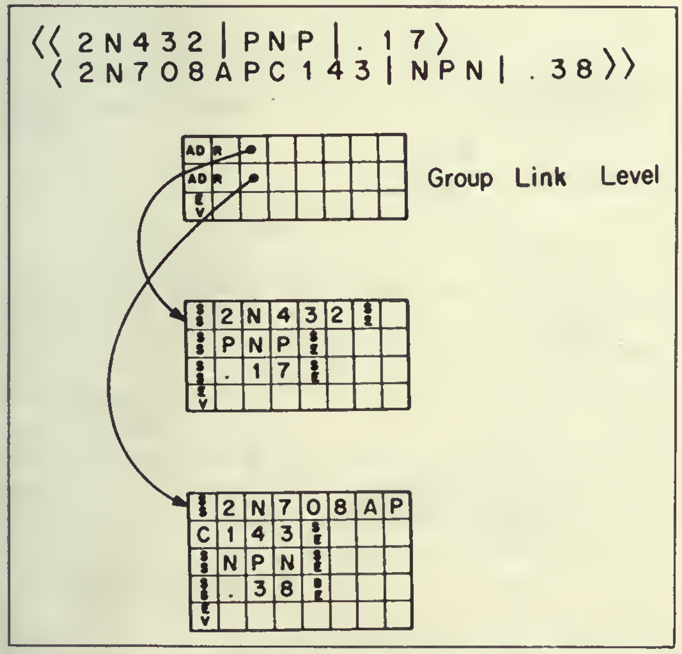

**Fig. 11. A simple two dimensional array and the corresponding three variable length memory strings that are used.**

## *Object String and Name Tables*

When a program is compiled the translator creates a reverse Polish string with postfix operator notation and a structured name table. The Polish string, called the object string, and the name table are the basic information forms used during program execution by the central processor. The use of a separate name table during execution is perhaps one of the most distinctive departures from traditional processing forms. Where in most systems the program string to be executed contains address references to the data space to be utilized, with the SYMBOL system the object string contains references to entries in the name table which act as a centralized point where all information about a given identifier is kept. It is this feature that gives the system its extreme execution time dynamicism. Whenever the nature of an identifier is modified in any way—location, size, type; etc.—only the name table entry need be changed since all references in the object string to an identifier must go through this entry.

The source form of a simple assignment statement and the corresponding object string and name table are shown in Fig. 12.

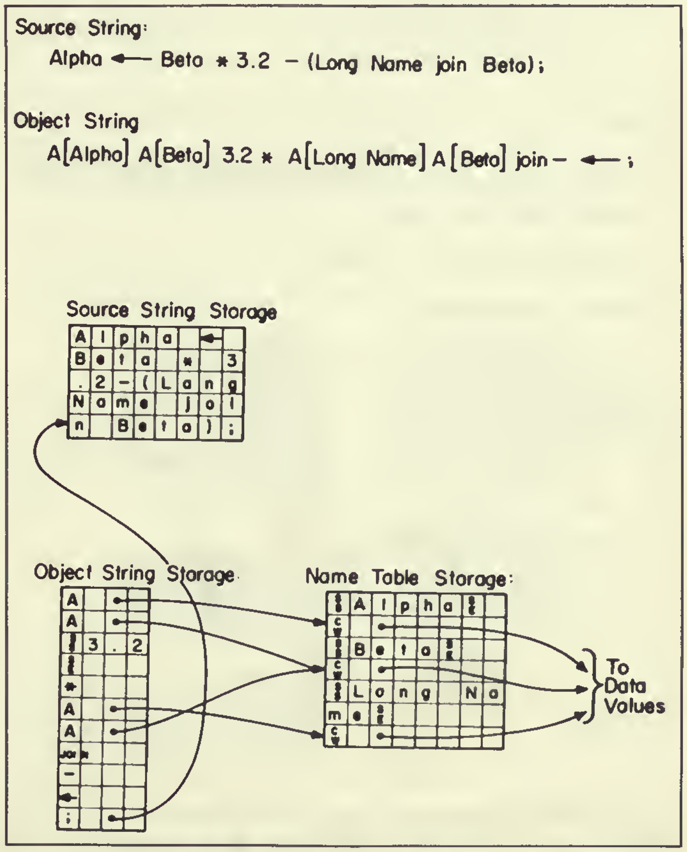

**Fig. 12. Information structure for a simple assignment statement.**

The identifiers are isolated and added to the name table when not already there. Note that the identifiers can be variable length and have more than one word. Associated with each identifier is a control word. All references in the object string involving the identifier will point to the corresponding control word. The object string is composed of name table addresses, literal data (the value 3.2), operators in postfix representation, and correspondence links back to the source string. The correspondence links are for simple error diagnosis and are therefore ignored during normal execution. The object string and name table are totally independent of the future size and data type of the variable.

Now consider the name table after execution has begun and assume that the data variables have current values. In Fig. 13 the variables Beta and Gamma are simple fields. Gamma is a multiword string and therefore it is stored in a memory string with a link address placed in the corresponding control word. Beta is a short field such that it can be stored in one word directly in the name table. Alpha is an irregular structure. The name table for Alpha contains a link to the first group which in turn contains two string fields, two link addresses, and an end vector mark. The link addresses point to two groups, one containing two fields and one containing three fields. As execution progresses the attributes and storage representation of the variables may change. In any event, the name table and the data itself will contain all the attributes of the variables.

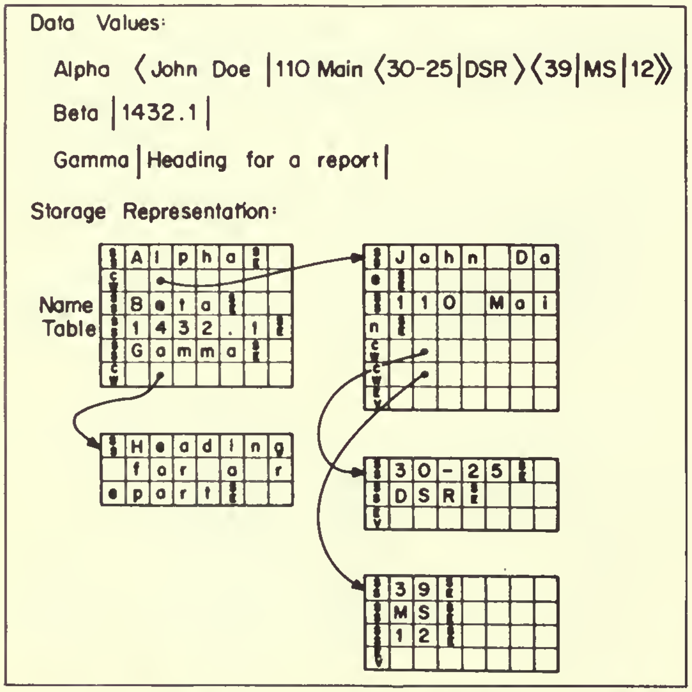

Fig. 13. Examples of a structure and two fields and how they are stored into memory along with the name table.

## Basic Information Flow

In order to observe how the various processors of SYMBOL are used to serve the end users problem temporarily ignore the multiprocessing aspects of the system. A user at a terminal operates in various modes; program loading, program compilation, and program execution are the fundamental usage modes. Consider the state diagram in Fig. 14. A user would start in the OFF-LINE mode and by some transitional control means he would initialize his tasks into the ON-LINE IDLE mode. From here he can go into the LOAD mode to develop a program. When he is ready to execute his program and assuming he is a perfect programmer, he would have his program compiled and executed. At the end of execution he can restart and rerun his program or he can return to the LOAD mode and modify his program.

The following sections deal with examples of the information flow for the basic operational modes of a terminal. A more detailed system block diagram in Fig. 15 will be used to support the description. Visualize the time sequence of the terminal operational states of Fig. 14 in conjunction with the static hardware diagram of Fig. 15.

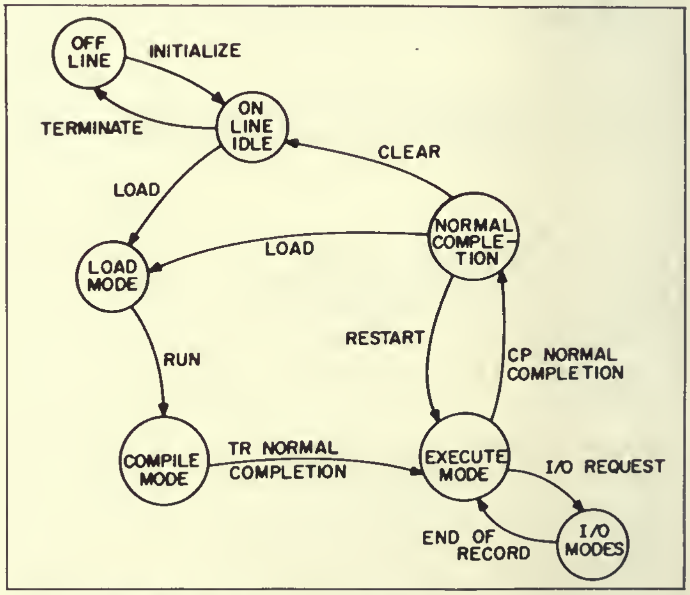

Fig. 14. Idealized task flow for one terminal.

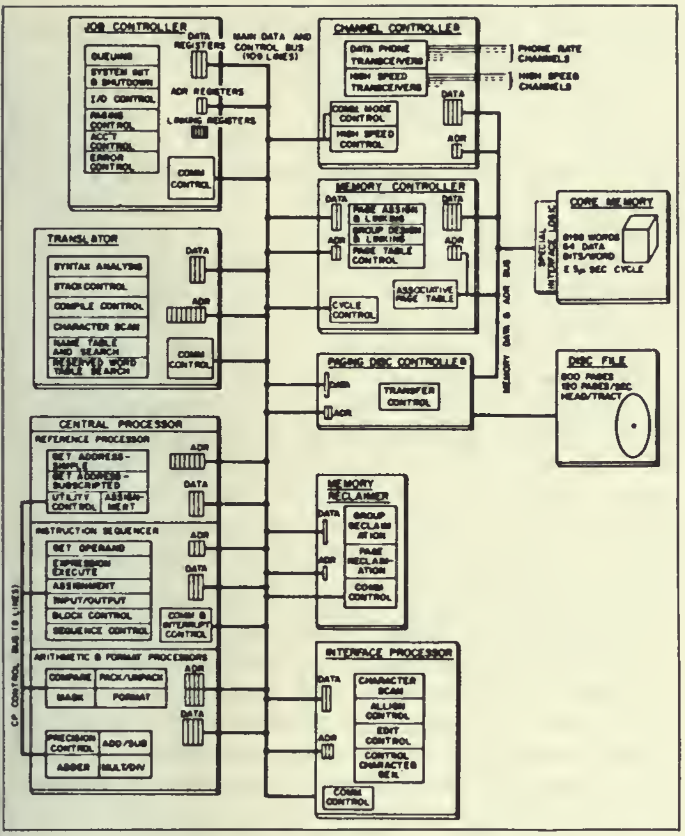

**Fig. 15. A more detailed block diagram of the SYMBOL system showing register configuration and major functions within each processor.**

## *Load Mode*

The LOAD mode is an input/output text editing mode. Its primary purpose is for program source loading. In the normal case a separate page list is used to store the text string. This area is called the Transient Working Area (TWA).

Three processors work together to perform the text editing tasks. The Channel Controller (CC) transfers data characters to and from I/O devices from and to the I/O Buffers in main memory respectively. When the CC detects control characters in the I/O stream, it communicates the control information directly to the SS by way of a control exchange cycle. The CC is a character oriented processor which services up to 32 processors in a commutating manner. The CC also has a high speed (block) operating mode which is priority driven to allow servicing of disk and high speed tape devices. The block mode is not used in the LOAD or normal I/O mode.

The Interface Processor (IP) operating on a burst basis empties or fills I/O buffers and transfers appropriate characters to and from the virtual memory. The IP works with a current text pointer while performing its functions. The IP functions include basic text insertion, searching, displaying designated text portions, deletion of designated text portions, and moving the current pointer. In Fig. 16 the basic information flow during the LOAD mode is summarized.

Part of the justification for implementing editing functions in hardware came from the desire to eliminate the CP from many of the system overhead tasks. In addition, response times would be unacceptable if the CC were to communicate directly with virtual memory. The IP was developed to make the basic transfers between small buffers and paging memory. Once a special processor was developed it was found that many editing tasks and double buffering could be handled using essentially the same data transfer hardware.

This IP/CC/SS process is available for both LOAD mode data preparation and program execution I/O. The full text editing facilities are available for any program input statement.

## *Compile Mode*

Program compilation and address linkage editing functions are performed by the Translator (TR). The TR accepts the language source string from the TWA or some other source text area in virtual memory. The high level language is converted into a reverse Polish string and a structured name (identifier) table. The Polish string, called the Object String, and the Name Table may be stored in Virtual Memory on separate page lists or on a common page list. The gross flow of information in the Translation mode is shown in Fig. 17.

The TR performs a one pass compilation generating the object

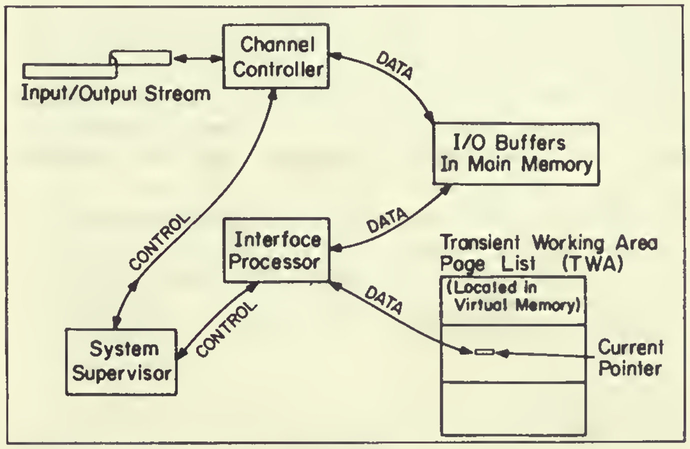

**Fig. 16. Information flow in the LOAD mode.**

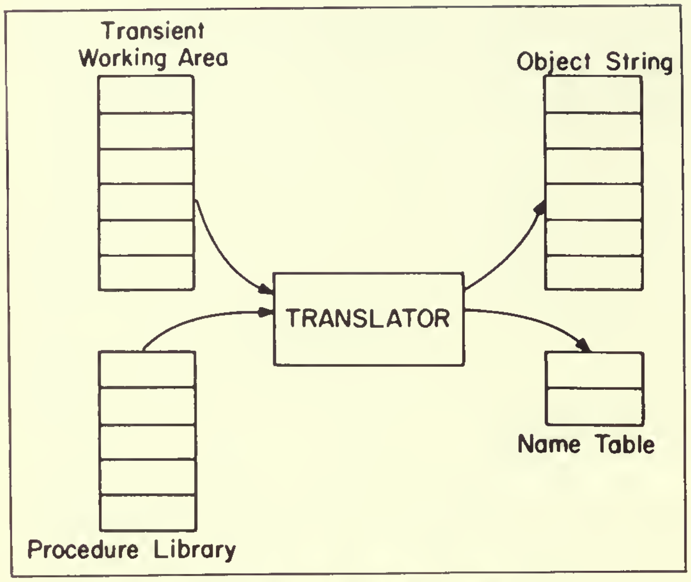

**Fig. 17. Information flow in the COMPILE mode.**

string as it scans the source string. It also builds the name table during this scan on a program block-by-block basis. At the end of the source pass the TR processes the name table and resolves all global references by creating appropriate indirect links. External procedure references are resolved during the name table pass and they are compiled and included with the object string as needed.

The TR includes external procedures by accessing procedure source libraries and compiling needed procedures into the object string. The procedure libraries are organized into two sets, namely privileged and non-privileged procedures. Privileged programs differ from normal programs in that they can contain privileged statements for direct memory manipulation using the MC operations. Storage protection is obtained by controlling the privileged status of user programs and the programs that they can reference. Non-privileged programs have a high degree of storage protection both from other programs and from themselves due to the hardware storage management and central processor algorithms. Programs using privileged statements lose some of the protection. By controlling the access to privileged programs and the manner in which they are used the overall storage protection in the system is quite satisfactory for multiterminal operation.

### *Execution Mode*

The Central Processor (CP) is the execution unit for the translated language receiving the translated source string along with the nested name table blocks as input. Because the CP operates on a high order language—actually a Polish string, postfixed operator object string—the CP uses a push-down stack for its operands. That is, the data reference is generated with all indirections traced out until a memory reference point is reached, and then this reference is pushed into the stack. This process continues until the postfixed operators are encountered in the object string. Each operator causes the top one or two (monadic or dyadic operator) stack entries to be pulled up, processing to take place, and the result to replace the operand(s) on the stack.

Substructure referencing, also known as subscripting, is a much more formidable task in SYMBOL than with conventional systems. This is due to the extremely dynamic flexibility of these structures. With conventional, systems, accessing an element of a vector is a simple matter of assigning a base along with an index register for the subscript variable and at execution time merely doing an address calculation to find the desired element. With SYMBOL there can be no possibility of a base address or an address calculation both because of the dynamic nature of space allocation as well as the fact that logically contiguous data need not be physically contiguous in memory. The Reference Processor (RP) has the charter for finding substructure points, basically through a scanning technique along with several speed-ups.

Another novel aspect of the CP is that all processing operations are done on variable length data. The string operations can be of any length, the only limitation being that they must fit into the main memory. The numeric operations are limited to a 99 digit fractional length (numbers are represented internally as normalized floating-point decimal numbers). Furthermore, the length of numeric processing is controlled by the limit register. Also, a precision mode exists whereby numbers tagged with EM (empirical) will limit processing precision to the number of fractional digits they contain, unless the limit register is set to a smaller value.

The information flow for the CP is summarized in Fig. 18. The CP has four distinct sections, namely the Instruction Sequencer (IS), the Reference Processor (RP), the Arithmetic Processor (AP), and the Format and String Processor (FP). As shown in Fig. 15 the CP has a common control bus that is used to control the various processors during program execution. The following four sections describe the functions of each of the processors in the CP.

### *Instruction Sequencer*

The IS portion of the CP is the master controller and switching unit of the CP. It has the task of scanning the object string, and accumulating items in the stack for the various units it supplies. For example, operands are accumulated for the process units and any type conversion required is sensed and requested of the FP by the IS, as appropriate. Similarly, a structure reference and all of its subscripts are computed and placed into the stack which is then turned over to the RP for access.

The IS also prepares data for assignment by the RP or output by

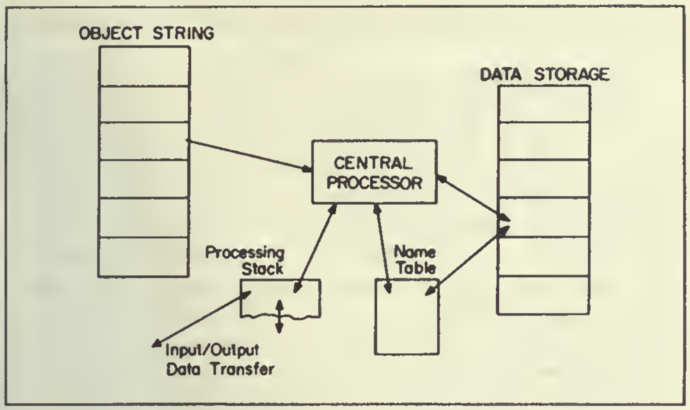

Fig. 18. Information flow during program execution.

the I/O unit. It does this in the former case by stacking both the assignment reference and the data and in the latter case by stacking the data and turning control back to the system.

Another major task performed by the IS is that of dynamically creating nested languaged blocks. Reference should be made to the companion paper on the SYMBOL language [Chesley and Smith, 1971] if the concept is new to the reader. In quick review, blocks are language constructs consisting of program segments contained between the reserved words BLOCK and END (PROCEDURE and ON also establish blocks). Within a block, all uses of an identifier are local to that block, unless contained within a GLOBAL statement, and thus a different name table is constructed for each block. The overall structure of name tables has a static aspect determined by the way the program is written and a dynamic aspect determined by the sequence in which these blocks are executed. It is this latter characteristic that we are concerned with in this discussion. Whenever a new block is encountered by the IS, processing on the old block is suspended by pushing down all information about that block that must be retained (sometimes called the activation record) into the stack, and starting a new stack and activation record for the new block on top of the old stack. Of course, the new record must contain a link to the old record so that when the new block is completed, the old block with its status information can be reestablished.

A further complexity occurs with procedure blocks because of the need to correlate actual and formal parameters (again, see the language paper) [Chesley and Smith, 1971]. The IS transfers the links to the actual parameters from the object string to the stack, accesses the name table for the new block where the formal parameters occur as the first entries of this name table. The actual parameter links are then placed one-by-one into the formal parameter entries of the name table. Parameter linking completed, the remainder of the normal block action for the procedure is accomplished. Whenever the IS encounters a name table entry tagged as a formal parameter, it indirectly accesses the actual parameter in its place, which may not be a statement, but may be a variable, constant, label, literal, procedure, or expression. This indirection mechanism is also handled in the IS stack. A push down of a limited set of status information takes place, mostly consisting of the address where execution of the object string was temporarily discontinued. Then the new object string of the actual parameter is executed, using the stack until the return operator is encountered indicating the end of the actual parameter string. This causes the previous status to be recovered from the stack and execution of the object string recommences with the results of the execution of the actual parameter remaining in the top of the stack.

### *Reference Processor*

The basic task of the RP is to deal with structures. As a simple added duty, it accesses the address of an item from the name table for the IS. That is, the IS receives an address from the object string and turns it over to the RP with a request to "get simple address." The RP performs several actions depending on the nature of the identifier. If it is an existing data item it provides the address of the data along with a code indicating its nature. If it is an uninitialized data item, it first assigns space before supplying the data address. In a similar manner it provides links to labels and procedures and if any identifiers are global, it first traces out the global indirection before returning the link. Any anomalies in the name table cause an error return.

The structure handling task may be broken down into two subtasks: creation of structures and substructure and the referencing of substructure points. Recall that structures are dynamically variable in all aspects. Thus, there are two further subsets under the creation of structures: creation of basic structures and the reconfiguration of substructures. As a subset task to the referencing of substructures the language contains a character subscripting capability where the final subscript may be a "bound-pair" of subscripts which refer to the starting point and extent of a character subfield with the previous subscripts pointing to the field.

The RP receives a linear representation of the structure to be created in the IS stack. The RP must store this structure in memory, replacing its linear form with a hierarchical form with links to lower or deeper elements occurring at the next higher level. Refer to Figs. 10, 11, and 13. It achieves this by assigning a new memory group each time it encounters a new left group mark, creating a line to the new group in the higher group and filling that group with elements maintaining a link back to the higher group in its own group link stack. Whenever a right group mark is encountered in the IS stack, the current memory group is closed with an "end vector" tag and the next higher memory group continuation point is accessed from the group link stack. This process continues until the structure in the IS stack is exhausted and results in a linked, hierarchical structure.

A similar process takes place when a new structure is assigned to an existing substructure point. The old structure is deleted (for later recovery by the memory reclaimer) and the new linear structure in the stack is structured and linked into the proper substructure point. All combinations of replacement are allowed: structure by a structure, field by a field, structure by a field, field by a structure. The second situation of a field replacing a field can be a problem in the case where the new field is larger than the old field because vector expansion must take place (in the opposite situation, nulls are inserted). The simple solution of providing a non-hierarchical link out of a new space is inadequate for the situation where successive words of a large vector are sequentially expanded. The solution is to link in a new memory group only after checking if there is no space remaining in the present group or the next one, and then rewriting the remainder of the present group adjacent to the new field. In this way, expansion of many fields of a vector makes use of the newly created space.

The general algorithm for structure referencing is for the RP to scan back through the IS stack to find the structured link, and then to proceed upward a subscript at a time, accessing each vector using special speed-up techniques as appropriate, until the final subscript is reached. At this point the RP replaces the subscripted reference in the IS stack with a link to a substructure or a link to a field if the data level was reached. At any point in structure referencing, the structure previously stored may not extend to the referenced point (oversubscripting). The language rule in this situation is that new space should be created as required to expand the structure to the subscripted reference point (fields filled with nulls) and the RP is responsible for accomplishing this task.

If after structure referencing to the field level, a bound pair of subscripts appear in the IS stack, the RP scans and counts across the field, selecting the requisite characters and placing the result in the IS stack. An error is called if the bound pair is encountered before the field level is reached.

### *Arithmetic Processor*

The AP is a serial process unit operating on variable length data consisting of floating-point, normalized, decimal numbers. These operations are done from high-to-low order to simplify data handling by allowing the register operations for both string and numeric processing to be similar. Also, comparisons are faster because a mismatch is immediately known. Two other important features are included in the processing hardware: a limit register, loaded by the IS under command of the language, which causes processing to terminate at the precision specified, and a precision controlling mode whereby each operand can be specified to be accurate to its existing precision and thus control the precision of the result.

The operations add, subtract, multiply, and divide are performed. For add and subtract, one or the other operand is streamed through the unit (high-to-low) until the exponents are aligned, at which time both operands start to stream through. Since the number representation is magnitude plus sign, a positive result is desired so that the signs of the operands and the sign of the operator are combined to control which, if either, of the operands is streamed through in complemented form. High-to-low order arithmetic requires a nine's counter [Mullery, Schauer, and Rice, 1963] to delay output over an intervening string of nines until a carry/no carry decision is reached. Eventually, either an empirical end of an operand is reached, or the limit counter value is reached, or both exact numbers are ended. At this point, arithmetic is finished and control is turned back to the IS.

Multiply is accomplished by successive additions or subtractions followed by a shift until all of the multiplier digits are exhausted. Only after the full trapezoid of the partial product is produced is a rounding pass applied to achieve the precision requirements. The speed-up of adding one to the previous multiplier digit and subtracting from the partial product if the multiplier digit is larger than four is used. Of course, with multiply (and divide) exponents are added (subtracted) so that no shift of the fractional portions of the operands are required. Division is accomplished by a gradual non-restoring reduction of the partial dividend until the precision of the result is equal to the least precise of the two operands or the limit counter.

Since processing in this system is accomplished serially in a decimal mode with few speed-ups, the speed of processing is sharply dependent on the size of the operands. When the limit counter is set to a small value, say 5, processing can be quite fast but 99 digit divides can be extremely slow. It is therefore important that the user selects only as much precision as he really needs.

The numeric comparisons are performed by the AP as a subtract operation but terminate immediately upon a mismatch and return a zero result rather than a one. The IS has the task of combining the result returned by the AP with the desired comparison operation to generate the overall result in the IS stack.

### *Format Processor*

The FP unit performs the string JOIN operation, the binary string operations AND, OR, NOT, the string comparison operations BEFORE, SAME, AFTER, the FORMAT and MASK operations, and the automatic type conversion on operands requested by the IS: numeric to string, string to numeric, and numeric to integer (used primarily for subscripts). These operations are also performed serially.

The JOIN operation is performed in the obvious manner of streaming the second operand onto the tail of the first operand, forming a single result operand.

The binary operations are performed character-by-character, performing the required operation by producing 0/1 result characters, filling in the shorter operand with zeros.

The string comparisons are also performed character-by-character, comparing successive characters until a mismatch is found according to the built-in ASCII collate sequence and returning a 0/1 result.

The FORMAT and MASK operators provide a powerful string manipulation capability for a wide variety of applications from payroll and banking forms preparation to system software character manipulation. FORMAT is a packed-numeric-to-string operator that allows the user to describe the format of the result with a pictorial like character string. The operation is performed in a serial manner as dictated by the operands. The standard default conversion from packed numeric form to string is a subset of the FORMAT operation. MASK is a string-to-string operator similar to FORMAT. MASK can be used for character insertion, deletion, and spacing control. It is often used to control or measure the length of the fields. MASK is also processed in a serial-by-character manner.

## System Supervision

The Load, Compile, Execution, and I/O comprise the basic processing modes for the system. Three additional modes are defined for a terminal, off-line, on-line idle, and normal completion. They are all passive modes and differ only in the allowed transitions that can take place upon an interrupt stimulus. For example, the normal completion state is the only state from which the RESTART execution command can be honored. RESTART is only allowed if the object string were left in a reusable state.

The diagram in Fig. 14 shows a few of the terminal state transitions. These transitions are significant in that they are all supported by hardware algorithms. When the control code corresponding to RUN is received by the SS the transition from the Load mode to the compile mode can be processed without software intervention. Many other transitions can occur but they generally require some system software assistance. The transition from the Load mode to the Compile mode involves the following steps. If the IP is active it must be allowed to complete in such a way that the source string is intact. The task is then removed from the queue for the IP and added to the queue for the TR. In addition, the control tables in main memory are initialized for the TR making available the address of the start of the source string and the address of the procedure libraries to be used.

A typical task queue is illustrated in Fig. 19. It is comprised of a linked list of entries (control words). The queue has a pointer to

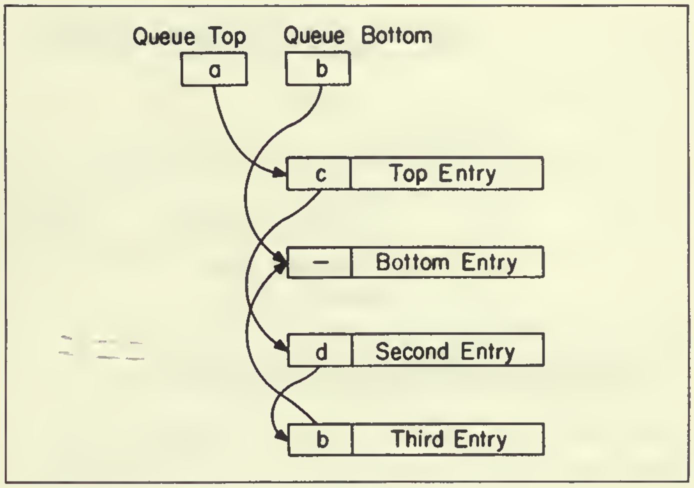

**Fig. 19. Typical task queue structure.**

the top entry and another pointer to the bottom entry. By maintaining both the top and bottom pointers it is easy to add an entry to either the top or the bottom of the queue.

Each time a control transition occurs the SS updates the queues by performing appropriate add or delete actions to each of the processor queues involved. This is part of first phase of any SS task processing. The second phase of SS processing involves assigning work to free processors that have assignable tasks on their queues.

The multiprocessing algorithm is centered around manipulation and use of queues for the CP, TR, IP, MR, and DC. The SS has a general purpose queue processor that allows an item to be added to the top, added to the bottom, or deleted from any queue. The algorithm has a default mode which is completely hardware controlled. Various parameters can be set by software that bias the operating dynamics. For example, two time values are maintained for each entry in the CP queue. One measures the accumulated processing time and the other measures the actual time that the task is on the top of a queue. The values are preset to parameter values when a task enters the queue. When the values have been counted down to zero an SS task is generated to modify the queues. In most cases this is used to move the task from a high priority position near the top of a queue to a low priority position near the bottom of a queue.

The processing flow in Fig. 14 is greatly oversimplified for general purpose system supervision. In Fig. 20, the control commands to and from the central processor are illustrated. The SS can command the CP to start on a task or to quit working on a task. The CP can terminate processing on a given task for one of six basic reasons. Consider the I/O completion. In most cases for most terminals the hardware algorithm for controlling I/O would be sufficient. If on the other hand, a batch processing terminal

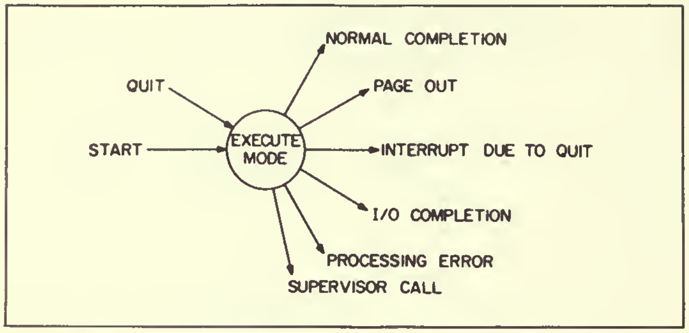

**Fig. 20. Mode transitions affecting the central processor.**

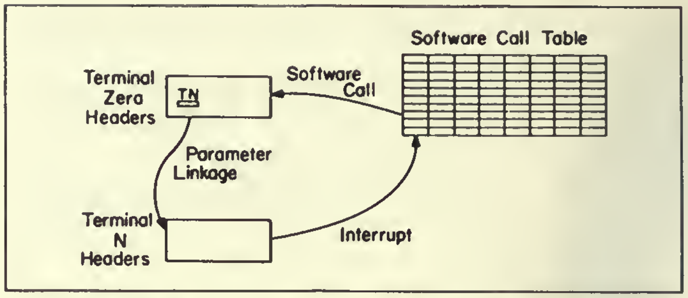

**Fig. 21. Mechanism for handling a software call caused by a transition interrupt.**

with spooled I/O were desired, it would be necessary to alter the control process for I/O with a system software procedure. To cause software to be called for a specific terminal upon an I/O service request, a specific control bit must be set in the terminal control word for that channel. This causes an automatic software call to be generated by the SS.

The software call is handled in SYMBOL by starting a pseudo terminal operating with the requesting channel number as a parameter. In this manner the control header tables for the requesting channel can be operated upon as data. This is illustrated in Fig. 21 where an interrupt of a specific class causes the corresponding program specified in a software call table to be selected and control transferred to the pseudo terminal with the parameter TN. Each different class of interrupt maps into a different control word in the software control table. In this manner only the software procedure desired will be accessed in virtual memory. In SYMBOL over 80 different software interrupts are controlled via the software control table located in the lower part of main memory. This represents the principal interface between hardware and system software.

## References

Chesley and Smith [1971]; Corbato and Vyssotsky [1965]; Cowart, Rice, and Lundstrom [1971]; Glaser, Couleur, and Oliver [1965]; Kilburn et al. [1962]; Mazor [1968]; Mullery, Schauer, and Rice [1963]; Rice and Smith [1971]; Smith [1963, 1968].

# The SYMBOL Computer
# SYMBOL: A Major Departure from Classic Software Dominated von Neumann Computing Systems[1]

*R. Rice / W. R. Smith*

## SYMBOL from a Performance Viewpoint

The evaluation phase of SYMBOL IIR is just beginning with the hardware near completion. In order to obtain a preview of the performance a set of measurements has been made on the hardware.

### *Basic Operation Rates*

The clock period on SYMBOL IIR now stands at 320 nsec and may be later reduced to about 200 nsec. All measurements were taken at the 320 nsec period. The basic clock period in SYMBOL IIR contains long logic chains allowing relatively complex tasks to be performed. Many of the key logic chains contain 20 to 25 levels of AND-OR logic. The system uses Fairchild CTμL, type I throughout. The core memory is a 1964 model with a basic 2.5 μsec cycle. Due to a semi-serial interface on the core memory it has an effective cycle of 4 μsec.

An improved system (referred to as SYMBOL II) has been studied and has been partially specified. This system is based on the technology of the experimental system, SYMBOL IIR, but has been considerably optimized. SYMBOL II is also specified to use

[1]From *AFIPS SJCC*, 1971, pp. 575–587.

the latest cost orientated hardware technology. Conservative performance estimates of SYMBOL II will be made to give a comparison of how the SYMBOL algorithms would stand up in a contemporary hardware technology design. They will be based on a clock period of 100 nsec using a circuit family such as CTμL, type II and an LSI memory with a 200 nsec period. One should keep in mind that the following comparisons are between SYMBOL, which is a VFL machine running in a very dynamic execution time environment, and a more conventional fixed field machine running a language with the data boundaries determined at translate time. The former places more demands on the hardware while the latter shifts the burden of data management to the user.

For the purposes of comparison SYMBOL IIR will be referred to as SIIR and SYMBOL II as SII.

## *Field Processing Operations*

SIIR performs all field operations in a VFL serial-by-character mode. It was always assumed that after system evaluation and bottle-neck analysis, if warranted, certain operators such as those shown below would be executed in a more parallel mode by using additional hardware. SII estimates are based on serial processing and known algorithm improvements that reduce or do not materially increase the hardware required.

The following table gives processing times measured on SIIR and estimated for SII. The execution time values are specified in microseconds and do not include the instruction fetch time or single word operand fetching and storing.

**SYMBOL IIR Measured Execution Times in μsec**

| *Operation* | *SIIR* | *SII* |
|---|---|---|
| 1234+4321 | 5.6 | 1.2 |
| 12345678−87654321 | 10.0 | 1.6 |
| 50 digits + 50 digits | 45.0 | 5.0 |
| Convert to floating point 1234 | 5.2 | 1.2 |
| Convert to floating point 12345678 | 12.5 | 1.8 |
| Convert to floating point 50 digits | 120.0 | 18.0 |
| Compare 12345678,87654321 | 4.0 | 1.0 |
| Compare 12345678,12345670 | 6.5 | 1.2 |
| \|abc\|join\|def\| | 4.5 | 1.2 |
| \|12345678\|join\|12345678\| | 60.0 | 12.0 |
| 1234 format \|ZZZ.DD\| | 9.0 | 3.0 |
| 1234 format \|ZBZBV\| | 8.0 | 2.6 |
| 12345.6789 format\|·$·*C***C***.DD\| | 76.0 | 15.0 |

## *Compilation*

Several programs were compiled on SIIR and the overall times and space usage measured. The SIIR results are as follows:

**SYMBOL IIR Measured Compile Times in μsec**

| | *No. statements* | *Bytes of source* | *Bytes of object code* | *Average time per statement* |
|---|---|---|---|---|
| Program A | 195 | 8330 | 7315 | 820 |
| Program B | 70 | 3528 | 5112 | 1280 |
| Program C | 157 | 7560 | 6025 | 760 |

This represents about 75,000 statements compiled per minute on SIIR.

A comparative table for SII assuming added flexibility on SII for handling various other languages in addition to the SYMBOL language is given below. The data is based on a sampled study of object code and projected execution times of several recently developed algorithms.

**SYMBOL Estimated Compile Times in μsec**

| | *Statements* | *Bytes of source code* | *Bytes of object code* | *Average time per statement* |
|---|---|---|---|---|
| Program A | 195 | 8330 | 2350 | 185 |
| Program B | 70 | 3528 | 1735 | 220 |
| Program C | 157 | 7560 | 2110 | 185 |

This would give a compilation rate of 300,000 statements per minute.

## *Paging Overhead*

SYMBOL has very low overhead for paging. The algorithms are based on direct hardware execution using parameters set up by software. A count of worst case paging overhead for SIIR in terms of memory cycles for a CP page out is given below.

**SYMBOL IIR Paging Overhead in Memory Cycles**

| *Item* | *Worst case* | *Average* |
|---|---|---|
| CP Shut Down | 7 | 7 |
| SS Queuing and Push Selection | 50 | 30 |
| SS Disc Servicing | 8 | 6 |
| CP Start Up | 6 | 6 |
| Total memory cycles | 71 | 49 |

Assuming an average of 5 μsec per memory cycle counting internal cycles this gives 355 μsec worst case. In SII using an improved algorithm the overhead would be less than 20 μsec.

## *Input/Output*

The overhead for I/O for a time-sharing system becomes an important factor in providing adequate terminal response time. To illustrate the effect of the hidden software overhead an operation trace of a IBM 360/44 during FORTRAN IV output was performed. A similar operation was performed on SIIR. The equivalent output statements in both languages are shown in the table below.

**SYMBOL vs. FORTRAN Output Statement Traces In Memory Cycles**

| *Language* | *Statement* | *Traced* | *Est. overhead not traced* |
|---|---|---|---|
| SYMBOL | OUTPUT 12345.56 FORMAT \|D.DDD$_{10}$DD\|; | 130 | 0 |
| FORTRAN | WRITE (6,10)× 10 FORMAT (1×,E9.3) | 3466 | 1000 |

The trace of the FORTRAN statement indicated 1,753 instructions being executed. Each instruction requires an average of two memory cycles. The trace program does not trace any of the supervisor or channel operations so that well over 3,000 and more likely near 4,500 memory cycles were used in executing the FORTRAN statement.

### *Task Control Overhead*

In order to measure the overhead for compilation and execution a program consisting of one CONTINUE statement was executed on SIIR. This causes a null program to be entered, translated, and executed and thus places a large demand on any system resources required, isolating overhead from "useful" actions. All memory cycles were traced with the following distribution:

| *Processor used* | *Memory cycles* |
|---|---|
| SS | 41 |
| TR | 20 |
| CP | 18 |
| Total | 79 |

This could be compared with any contemporary system where the entire compiler would have been paged in and much of the supervisor would have been executed to establish many resources that would not have been needed.

### *Subscripting*

It would seem that VFL data structures imply slow data referencing. However, the SYMBOL project demonstrated that efficient handling of dynamically varying data can be achieved with sophisticated list processing techniques. SYMBOL IIR established the foundatiŏn and the algorithms have now been developed to be competitive with conventional fixed field indexing while retaining the VFL features. A few references and their equivalent memory cycles for SIIR are given below. The subscript Fetch cycles are not counted.

| *Reference* | *Typical memory cycles required* |
|---|---|
| A[4,9] | 4–6 |
| A[16,32,6] | 8–10 |
| A[3] | 2–3 |
| A[70] | 9–12 |
| A | 2 |

A substantial improvement has been obtained for SYMBOL II promising to make it as fast or in some cases faster than conventional indexing.

## SYMBOL from a Cost Viewpoint

A study of a modern computer installation and its users as a total "system" reveals where and how the computing dollar is divided. Consultants from Iowa State University made available all the necessary data for such a study early in the program [Rice, 1967]. Figure 1 illustrates the I.S.U. IBM 360/50 installation in 1966 at the time the study was made. This "pie" has since been compared with many other business and scientific installations of varying sizes with different computer systems. There is general agreement that the minor variations in the size of the slices for different installations do not materially affect the picture. This applies to most modern "classic software-dominated systems."

The objective of data processing is to solve problems where the

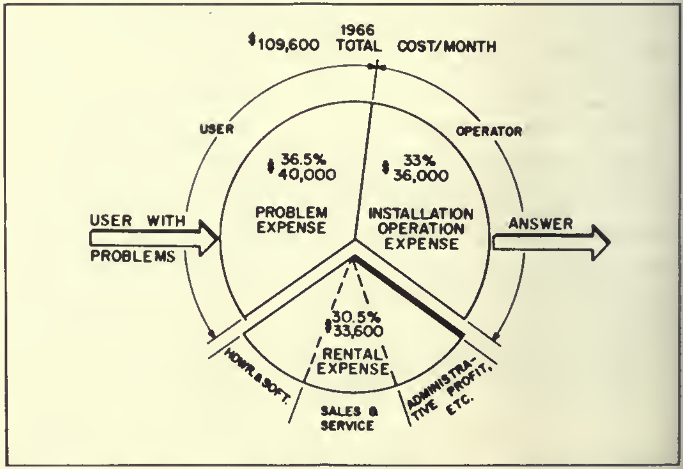

**Fig. 1. The computing pie illustrated for Iowa State University 360/50 Installation.**

"user with a PROBLEM" is the input and the "ANSWER" is the output. It is assumed that the user has his problem well defined and has the data available but the data is not yet programmed. The conversion of his problem to a computable language and the debugging necessary for correct execution is included in the total cost of operating an installation.

I.S.U. calculated the total system operation on this basis as approximately $109,600 per month. The rate and labor costs were adjusted to normal commercial standards for the calculations. Both commercial and scientific problems were run in the problem mix. The following sections discuss the breakdown of the overall cost.

About 37 percent or $40,000 is used by the problem originator and/or the professional programmer to convert the problem to a debugged, high-level language and to obtain answers.

Thirty three percent or $36,000 is required for operating personnel, keypunch operators, file clerks, systems programmers, administration, space, power, etc.

Thirty percent of the total pie or $33,000 goes for machine rental. It is estimated that about one third of the rental expense goes for direct development of hardware and system software (perhaps half and half), one third for sales, service, and application support, and one third for administrative costs, overhead, and profit.

The choice of a hardware configuration and its machine language is the tail wagging the dog. Inexpensive hardware and a good, easy-to-use programming system can reduce the size (i.e., total cost) of the pie but in conventional systems will not materially alter the relative size of the slices.

In the following text the computing pie is used to illustrate SYMBOL concepts from a cost point of view. Each major slice will be further subdivided into its own percentage parts (i.e., each major slice will be 100 percent of the portion under consideration and will be divided into its constituent parts).

Figure 2 shows the potential problem expense saving to be obtained from any good conversation-mode, high-level language, time-sharing system. It has been estimated that approximately 50 percent of the problem expense slice can be saved in reduced user learning time, increased throughput, less professional programming support required, etc. We estimate the SYMBOL system will further reduce these costs with its "clean" and "concise" directly implemented high-level language and simplified operating system [Chesley and Smith, 1971].

The savings in the operation of an installation comes from four sources. This is illustrated in Fig. 3. First: A good time-sharing system will reduce the administrative help such as file clerks, keypunch operators, etc. It is estimated that this saving can be ten to fifteen percent of the installation operating expense exclusive of system rental. The SYMBOL system with conversation-mode multiprocessing and multiprogramming will also share in this saving. Second: The "system software" support required in a conventional installation is a very significant portion of the expense. Here SYMBOL shows a definite added saving. What system software remains can be written in the high-level, general-purpose language and will be easier to write, debug, and understand later. This will reduce the number of professional personnel required. Third: The SYMBOL language is directly implemented in hardware and thus uses less main memory for "system software." For example, a resident compiler is not required. In addition, much less program swapping occurs and thus less virtual memory transfer time is needed. Hardware execution of algorithms is also faster and results in enhanced instruction execution speed. These features will require less programming attention and also provide more throughput per installation dollar spent. Fourth: The SYMBOL hardware is designed with modern integrated circuits and large two-layer printed circuit boards. The total system hardware package is compact and does not need raised floors, special air conditioning, or vast amounts of floor space. It is estimated that these SYMBOL features will reduce installation operating expense by an additional 20–35 percent or a total of 30–50 percent.

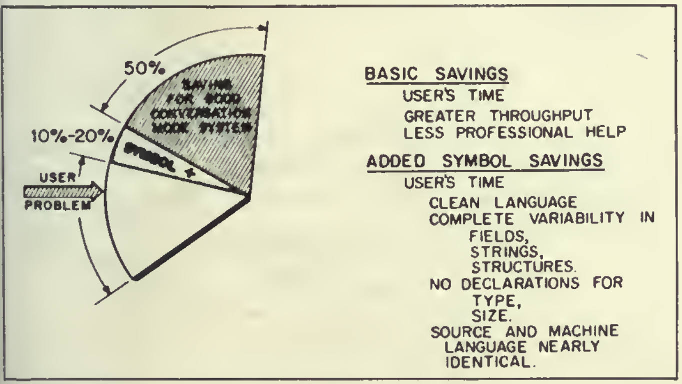

Fig. 2. Savings in problem expense.

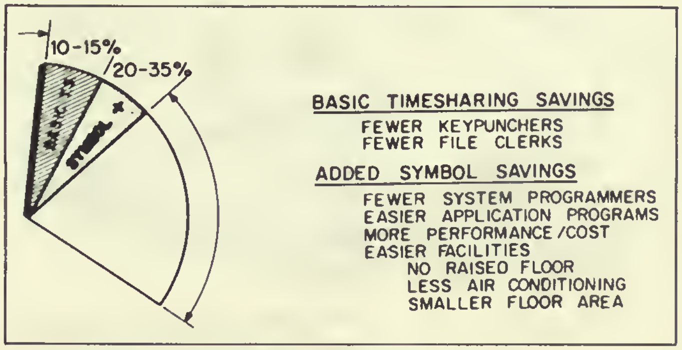

Fig. 3. Savings in installation operation expense.

The slice of the computing pie representing the computer manufacturer's hardware contribution is illustrated in Fig. 4; approximately seventeen percent of this slice is attributable to hardware. For large systems the peripheral equipment and the bulk files can approximate about one half of the total cost. The main storage is another quarter and the CPU logic is another quarter. Naturally some variation in these amounts will occur from installation to installation and for different system types.

The SYMBOL approach saves costs in several ways: The first area of savings is in the use of large two-layer printed circuit boards and two-layer printed circuit bases with cam-operated contacts for *all* system interconnections.

Except for cables to peripherals and wires used for correction of design errors and for logical extensions no wire exists in the system. It is estimated as much as a 50 percent saving will be achieved over small board, wire-wrap back panel, multi-cabinet conventional systems. This same technique reduces costs in terminal equipment but not to such a large degree. We estimate that three percent of the manufacturer's slice of the pie can be saved by this functionally-factored, bus-oriented, large printed circuit board design philosophy. The second way savings are obtained is in the hardware efficiency gained by the SYMBOL system. Since most of the normal system software is hard wired, very little resident main memory is used, thus providing much larger percentages of main memory for application programs. The execution of system instructions is done at "clock speeds" in a "macro" rather than a "micro" manner. This provides much faster high-powered instruction execution. Finally, more of the system hardware is simultaneously operating due to the system organization which allows multiple jobs to be in the main frame for overlapped execution. We estimate that an additional 2 percent of the manufacturer's slice of the pie is saved here.

The largest and most important single saving for SYMBOL is in the "System Software." Figure 5 illustrates this point. Irrespective of whether the system manufacturer or someone else produces the software for a conventional computer this large expense is real. The SYMBOL features directly implemented in logic (i.e., hard wired) make unnecessary at least 80 percent of the conventional system software used in large time-sharing machines. This represents an estimated 16 percent saving in the system manufacturer's slice of the computing pie.

The field support of the system software is a major expense. The sheer volume of paper and record keeping to keep current with the latest changes is a major problem. In the design of the SYMBOL system this problem was given great attention. In studying the software delivered with large systems using a relatively static high-level language, we note that most (if not all) of the changes made were on the programmed implementation or were due to programming errors. Many levels of machine and assembly language programs and machine runs were between the hardware language and the programmers' source language. This quite naturally introduces confusion (and errors) either in original programming or in understanding the hidden rules when using the system.

It may also be noted that as more and more applications are programmed in a language it automatically becomes more rigid. We believe that the "clean," high-level, general-purpose SYMBOL language is excellent for most uses. Since direct hardware implementation requires little field support in the software sense,

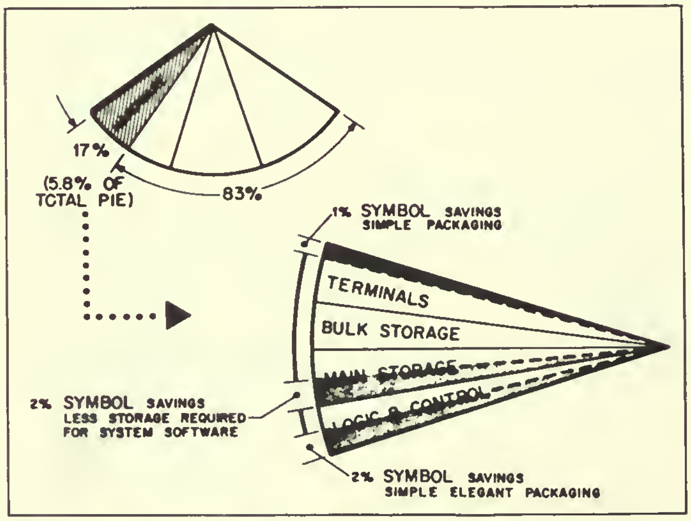
17%
(5.8% OF TOTAL PIE)
83%
1% SYMBOL SAVINGS SIMPLE PACKAGING
TERMINALS
BULK STORAGE
MAIN STORAGE
LOGIC & CONTROL
2% SYMBOL SAVINGS LESS STORAGE REQUIRED FOR SYSTEM SOFTWARE
2% SYMBOL SAVINGS SIMPLE ELEGANT PACKAGING

Fig. 4. Manufacturer's direct hardware expense.

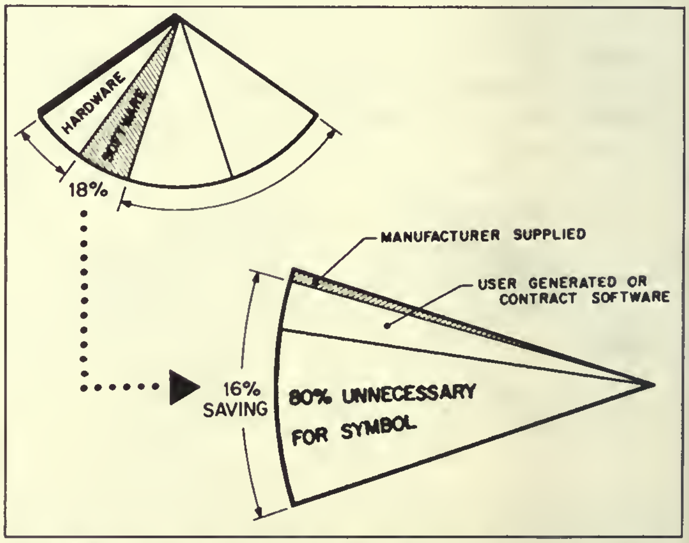
HARDWARE
SOFTWARE
18%
MANUFACTURER SUPPLIED
USER GENERATED OR CONTRACT SOFTWARE
16% SAVING
80% UNNECESSARY FOR SYMBOL

Fig. 5. Manufacturer's system software expense.

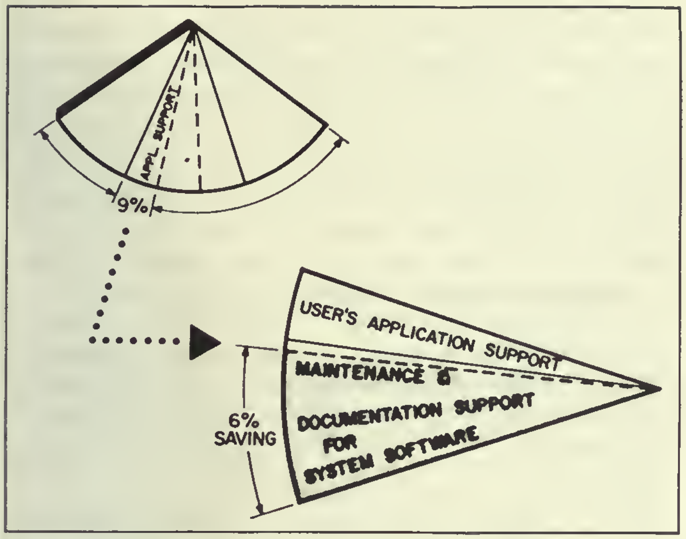
APPL SUPPORT
9%
USER'S APPLICATION SUPPORT
MAINTENANCE & DOCUMENTATION SUPPORT FOR SYSTEM SOFTWARE
6% SAVING

Fig. 6. Manufacturer's software application expense.

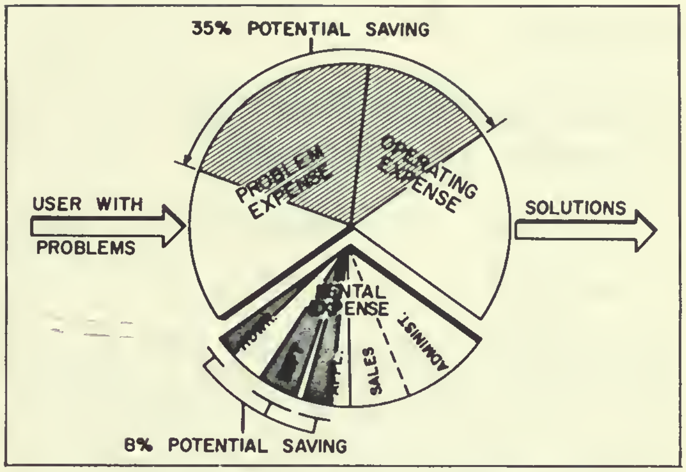

Fig. 7. Potential savings with a good conversation mode hardware/software system.

we estimate approximately a six percent saving in the manufacturer's support expense. This is illustrated in Fig. 6.

Good service is a must in a large system. The SYMBOL hardware has been engineered for good reliability and at the same time easy maintenance. We do not anticipate any added expense for SYMBOL hardware maintenance over conventional systems with equivalent storage and logic circuit counts. Our experience on the SYMBOL model has verified this belief.

The previous material has split the computing dollar up in parts and has described how major savings can be realized with a "total systems" approach. The SYMBOL techniques described herein together with good time-sharing, conversation mode practice can reduce computing costs up to 50 percent. Referring to Fig. 7, one may visualize how the savings in the whole computing pie add up.

## Conclusion[1]

The traditional boundary between hardware and software has been weakened during the past ten years and is due for a significant shift beyond the token improvements. It is believed that in SYMBOL a major step towards significantly more capable hardware has been attained.

The SYMBOL system is now entering an extensive evaluation phase where the system's strengths and weaknesses will become more apparent through actual day to day usage. The developers of the system have gained much insight into the merits of each of the approaches taken. The overall approach to memory management is considered a breakthrough. The moving of data attributes from instructions to the data is considered fundamental.

No claim is made that the SYMBOL system has been balanced for optimum performance and use of hardware. Certain critical areas of memory management and system supervision are felt to be 10 to 100 times more efficient than conventional means. Certain aspects of structure referencing are a major advance over software list processors but fall short of being competitive for some types of large array referencing. Many of the weaknesses in this first SYMBOL model were solved by the designers too late to be factored into the actual hardware. Many other aspects of the system such as the paging and system supervisor algorithms can be evaluated after significant usage experience.

The computing professionals have debated for many years the questions: Can a compiler be developed in hardware? Can the heart of system supervision be committed to hardware? Can data space management be taken over by hardware? Can hardware be designed to take over major software functions? Can complex hardware be debugged? These and many other questions have been positively answered with the running SYMBOL system. The most significant part of the entire project is that the concepts were reduced to full scale, operating hardware.

[1]This conclusion is taken from the paper by Smith et al. that makes up the first part of this chapter.

## References

Chesley and Smith [1971]; Rice [1967].