Microtext—The design of a microprogrammed finite state search machine for full-text retrieval

by R. H. BULLEN, JR. and J. K. MILLEN
The MITRE Corporation
Bedford, Massachusetts

INTRODUCTION

The Microtext system represents a new approach to the design and implementation of a full-text retrieval system. The approach is unusual in that it integrates hardware, firmware, and software components in an attempt to provide a solution to the problems involved in processing large files of unformatted textual data. The system is based on a minicomputer specialized for high-speed full-text retrieval, through the use of a finite state search algorithm implemented in firmware.

Full-text retrieval

Full-text retrieval, as distinguished from other types of text and data processing, involves the location of patterns of characters, words, and phrases in text. In addition, bibliographic structures, such as title or author, as well as linguistic structures, such as sentence and paragraph, can be identified in text when the data base has been suitably constructed.

A variety of systems have been built to perform full-text retrieval.1 If generalizations are possible, these systems can be divided into two categories:

(1) Those systems which use an index, or concordance, of text words during retrieval, but which have access to the full text for display. In some cases, such systems can also perform a sequential search of the full text for query items not also in the index. Generally speaking, search performance with indexed systems is adequate, but index generation and update is time-consuming and, as a result, editing or augmenting the full text must usually be done off-line, if at all.

(2) Those systems which always make a direct search of the full text. Such systems lend themselves well to dynamically changing file collections, because no indexing need be done; but file size is usually restricted because search time is proportional to the amount of text searched. However, because searching can be done on a character-by-character basis, direct searching can permit considerably more detailed query patterns than are possible with most indexing schemes. In addition, editing and augmentation of the full text can be performed on-line, although sometimes with side-effects which can adversely affect later search performance (e.g., file fragmentation).

Regardless of which of the above categories a full-text system may fall into, it is at an immediate disadvantage with respect to retrieval performance when compared, for example, with structured data retrieval systems (i.e., data management or management information systems). In the latter case, requests for qualifying data base entries can be satisfied by inspection of a selected subset of fields in each data base entry, whereas a full-text system must concern itself with all of the text in each entry. This problem is particularly acute for direct search full-text systems, since all of the text must be scanned each time a search is performed rather than only once, at index generation time, in the case of indexed systems.

Full-text systems, and specifically direct search systems, are plagued with a second problem, which is at the heart of the motivation for the Microtext system. In addition to having to process a very large amount of data in response to retrieval requests, direct search systems have a performance disadvantage because of the inability to express full-text handling functions in the primitives and data structures available on most general-purpose computers. Software must be used to map these application-level functions, often with great
difficulty, into the facilities of fundamentally word- and arithmetic-oriented central processors; it is this mismatch of problem and tool, and the additional level of mapping required, which adversely affects full-text handling systems, and it is to this mismatch that the Microtext work is addressed.

An application architecture

One approach to the solution to these problems, and the approach which was taken in the development of Microtext, is to work toward the design of a computer system specialized for full-text processing and retrieval functions. The system envisioned would be built up from hardware, firmware, and software components in the following way:

(1) hardware: state-of-the-art, commercially available hardware would be used to provide a low-cost, easily reproduced base for the system. The hardware would be chosen with a view toward its eventual use by judging its inherent suitability for character string handling, its raw performance, and its ease of microprogramming.
(2) firmware: microcode would define the data structures, primitives, and basic architecture (execution environment) for text handling problems at a level which facilitates their expression. In addition, because Microtext is viewed as an application-oriented machine, many functions typically thought of as the province of an operating system would be implemented directly in microcode.*
(3) software: software would be used for most data- and user-oriented functions so that they could be easily changed to suit specific application requirements.

The question is: how to get there from here?

The Microtext development plan

Aside from the fact that a task of this magnitude would take considerable time and money, with few intermediate products along the way, there is also a fundamental technical problem involved here. If a designer were to take the theoretical approach and begin his task by specifying the system architecture, he might risk bounding the problem before it was identified in a practical sense. If, as in the case of Microtext, this involved a higher-level application machine specification, later changes to the system due to practical requirements could affect the basic architecture of the system, and changes at that late date might not be tolerable. A more conservative, practical approach was chosen for Microtext.

The development plan for Microtext makes use of a phased, or boot-strap, technique, wherein the output of each phase is an operational prototype, the application of which can proceed in parallel with the design of the following phase. The approach has the advantage that each phase can bind to the basic architecture of the system only that subset of the application environment which has been proven through practical experience and user feedback, leaving still experimental components untouched in software where they can be changed easily if the need arises.

Phase I

To test the design philosophy described above, it was decided that the first phase in the development activity should be a prototype software implementation of a full-text retrieval system, with an important, but manageable, subcomponent of the system implemented in firmware. The idea was to take a cautious, initial step, to prove the feasibility of the approach as well as to encourage, through the production of software support, the development and use of data structures and primitive operations fundamental to full-text handling problems, so that these facilities might be well enough understood to be applied in subsequent phases of the Microtext activity.

In line with this goal, a fundamental primitive of full-text retrieval—character string searching—was selected for implementation as the function of a microprogrammed, black-box peripheral device attached to a larger host computer system, specifically an IBM System 370/155. A search algorithm was designed, using techniques of finite state automata theory, and was implemented in firmware on a Digital Scientific Corporation Meta-4 computer. Higher-level language software was used to implement the control logic for the device, as well as the application logic necessary to demonstrate the operation of the system.

The sections which follow describe the overall structure of the system, the special finite state search algorithm designed for the application, and the implementation of the algorithm in firmware. A final section dis-

---

* A recent project at MITRE has demonstrated ways in which operating system functions can be distributed between firmware and software.°
cusses some of the refinements planned for the Phase I system and suggests possible directions for Phase II activity.

**SYSTEM STRUCTURE**

**Application environment**

The goal in the specification of an application environment for the initial version of the Microtext system was to model the operational characteristics of a full-text retrieval system, without actually implementing all the bells and whistles which a demanding user might desire. For this first pass, we were most interested in basic structure, not so much in form.

From the user's point of view, Microtext provides an on-line full-text retrieval capability, available through a time-sharing system on IBM 2260 displays, 2741 terminals, and teletypes. The heart of the terminal environment is provided by three commands, described below.

**DQUERY—Display query questionnaire**

This command causes display at the user's terminal of a questionnaire which is used to specify basic parameter data for the search, as well as the query itself. Also specified in the questionnaire is information about the structure and format of the file to be searched.

The query language used to specify retrieval requests allows searches for words, phrases, or expressions involving words and phrases, optionally restricted in scope to the level of sentence or paragraph. This language is described in more detail in the following section.

**SEARCH—Search file**

This command causes the query to be processed and the search to be initiated. The query is redisplayed at the terminal for verification, and as the search progresses the search monitor displays continuous hit data to reassure the user that the system is actually working.

The user controls the frequency of this output by the parameters specified in the query questionnaire. At the end of the search, the system displays the total number of documents searched as well as the number of hits.

*The system under which the Microtext software runs is OS/MVT with the Time Sharing Option (TSO).*

**DANSWER—Display answers**

This command creates a file of retrieved text and allows the user to browse through this file.

**System operation**

In order to understand the role of the microprogrammed processor in the Microtext system, consider first the operation of a hypothetical full-text retrieval system, as it might be driven by the set of commands described above. For this purpose, the system can be thought of as three primary modules: (1) a query translator, (2) a search monitor, and (3) a display processor.

From the user's point of view, the first two of these modules are not really thought of as separate components, and in the online terminal environment described above, they are lumped together under the single "SEARCH" command. Figure 1 gives a flow diagram of such a system.

User input is first validated by the query translator and is then translated into an internal form, which facilitates easy evaluation of subparts of the query. This internal representation and the text file are then input to the search monitor which produces, not documents, but pointers to text items matched during the search. This list of pointers, and the text file, are then input to the display processor which gives the user access to the retrieved items.

It is in support of the operation of the search monitor that it was decided to apply firmware components first. To see how this was done, let us break the search monitor down further into the following processing functions:

1. A data base interface, which is concerned with data access requirements and with the specifics
Figure 2—Retrieval operation with Microtext search machine

of file structure, and which has the function of preparing text blocks for searching;

(2) an evaluator, which drives the matching process, evaluates the query, and records hit information;

(3) and a character string searching algorithm, which performs the scanning and recognition involved in the retrieval process.

Figure 2 shows this same system, no longer hypothetical, redrawn to indicate how the Microtext search machine replaces the character string searching function of the search monitor. Here the three major components perform exactly the same functions as before, but the data flow is slightly altered. The internal form of the query, described in more detail in a later section, is in a tabular form, highly compacted to fit within the available core memory on the Microtext search machine. After the table is generated, it is sent over a high-speed interface to the Microtext machine. Control is then passed to the search monitor which accesses the text as before, but appeals to special functions which communicate directly with the search machine, through operating system I/O facilities. The results of individual matches are returned to the host machine and hit information is recorded for later use by the display processor.

The reader will note from Figure 2 that there are several other ways in which the operation of a full-text retrieval system might be shared between a host machine and a specialized, microprogrammed processor. One such way might be the inclusion in the Microtext machine of the data base interface function and the incorporation in the design of a direct connection between this machine and the data base. This would have had the obvious advantage of avoiding the extra I/O transfer of text first to the host machine and then to the search machine, but would have been inconsistent with the development goals described in the Introduction to this paper. In this initial version of the Microtext system, we wanted to separate as much as possible the well-defined problem of character string searching from such functions as the data base interface, which are more likely to be sensitive to particular application requirements. The final section of this paper presents this mode of operation, as well as other alternatives, as possible directions for future work.

THE SEARCH MACHINE

Brief description

In this section we will examine the Microtext search machine more closely. It is, of course, implemented in firmware, but before we can fully appreciate this aspect of the machine, we have to understand the driving algorithm, and the manner in which the input to that algorithm is generated.

A finite state approach to character string searching satisfies the two requirements of (1) improving the performance of the sequential search, and (2) not sacrificing in any way the user's ability to state search requests that reap the benefit of having the full text available. This is accomplished by first transforming the search request into a table using a software routine. The actual search is then performed on each section of text by a very simple microprogrammed algorithm which operates on the text, the table, and a register holding the "state" of the search. A section of text is by definition that portion of text submitted to an individual execution of the search algorithm. Its length is controlled by application software, and it could range from a sentence to a complete document. The search passes through the text section from beginning to end, using each successive character to transform the state by consulting the table. At the end of the section, the output of the resulting state indicates whether or not the section satisfies the search request. (In cases where the search request succeeds or fails before reaching the end of the section, the search stops immediately and restarts with the next section.)

By choosing a query language abstractly equivalent to the regular expression language of Kleene, we can employ existing algorithms to construct a finite state recognizer for strings of characters satisfying the query.* At the same time, the regular expression

* We use "query" interchangeably with "search request."
language is powerful enough to support a query language at least as flexible as those designed for existing full-text systems.1,2

Various ways are then available for designing a table to direct the emulation, as it were, of the finite state machine. We have chosen the straightforward tactic of constructing a deterministic transition table. A nondeterministic version of a finite state machine is generally smaller and more easily found from the regular expression; a scheme for using it for string searches was suggested by Thompson.4 A nondeterministic search method, however, was thought less suitable for microprogrammed implementation because of the greater number of core references required per character.

It should be kept in mind that, while the table format (or choice of formats) is fixed by firmware, a new finite state machine to fill in the table must be constructed for each search request, preferably quickly enough not to discourage a user waiting at an interactive terminal.

The query language

Our present query language is regular expression notation modified for the convenience of the user. The precedence of operators has been changed to reduce the number of parentheses required in natural formulations of common search requests, and a number of standard abbreviations have been set up.

The following samples illustrate both the flavor of the present query language and the power of search requests based on regular expressions.

Query 1: /MICROPROGR/ & $/EMUL/
Query 2: /(# # (# ( U S) TROOP/ & / (WITHDR | PULL-OUT)/ & 'SENTENCE')/

Query 1 specifies a section of text about microprogramming but not emulation. The slashes indicate the embedding of the adjacent expressions in arbitrary text, and the blanks in contact with letters denote required punctuation or blanks. Thus, more literally, a section satisfies Query 1 if it contains a word beginning with "MICROPROG" but no word beginning with "EMUL."

Query 2 specifies a section mentioning the withdrawal of at least 100 U.S. troops. The number sign # stands for an arbitrary digit; the vertical bar is the "or" operator; the hyphen permits an arbitrary string of letters; and the angle brackets ( ) enclose an optional expression. In order to ensure that the "TROOP" mention is logically related to the "WITHDR" mention, they are required to be in the same sentence.

The queries are recognizable as regular expressions after the abbreviations have been expanded. For example, the slash / is translated to $ϕ. The 'SENTENCE' in quotes is an abbreviation for a moderately complicated regular expression characterizing the set of strings which can be sentences in the given database. The option brackets are expanded so that (expression) becomes an "or" between the null string and "expression" (the right bracket just becomes a right parenthesis). Incidentally, the digit sign # is not expanded into (0 | ... | 9), but is retained by the software as a single character-range symbol until the final construction of the table.

Query translation

Construction of the table from the query can be summarized in four steps:

1. Expansion of abbreviations
2. Infix-to-prefix translation
3. Production of the state graph
4. Table generation.

The Microtext implementation of this process is unusual in two ways: string manipulation techniques were used throughout (to simplify working space management and to anticipate the development of Phase II primitives), and several well-known algorithms were used in straightforward ways.

Expansion of abbreviations

While selecting the abbreviations requires some ingenuity, their expansion in the query is a simple table lookup. This is fortunate, because new abbreviations generally have to be designed for different data bases. For example, the fact that a data base may or may not have lower case letters affects the abbreviation for "arbitrary letter". Eventually Microtext software will have an associated data base descriptor file which will be used for, among other things, selecting the correct expansions. This will allow the user to express his queries in the same language, regardless of the structure of the data base he is searching.

* Regular expressions are built up from the character set using operators and two special symbols: phi (ϕ) and lambda or nil (λ). The symbol ϕ represents the empty set and ¬ϕ, therefore, represents the set of all strings. The symbol λ represents the null string.
Infix-to-prefix translation

The regular expression resulting from expanding the abbreviations is translated from its infix-operator form to a prefix form which is both more compact and easier to manipulate symbolically. Unions, intersections, and concatenations have any number of arguments and are thus parenthesized; complements and Kleene closures (stars) have one argument and have no bounding parentheses. Zero and one are used for \( \phi \) and \( \lambda \), respectively. A sample prefix regular expression is

\[
(\&\times(1 \text{ AB})(-\text{ 0B}))
\]

The infix-to-prefix translation is an instance of the classical use of a pushdown stack for this purpose. A transduction grammar of sixteen productions (exclusive of the replacement of the character set by a single non-terminal) was found and used in a simple syntax-directed translation using Lewis and Stearns' three stack algorithm.\(^4\)

Production of the state graph

Brzozowski's derivative method was implemented to produce the state graph.\(^4\) There is essentially only one other kind of method, based on Kleene's original proof that regular expressions can be recognized by finite state machines. It has two steps: generating a non-deterministic machine, and then converting it to a deterministic one. While this two-step method is fine in a batch system, such as the RWORD system for producing lexical processors, where it is followed by a state reduction phase, our early experiments in this direction were discouraging in speed of operation.\(^7\)

A number of far-reaching design choices were made here for reasons of efficiency.\(^*\) For example, the derivative algorithm generates a regular expression for each state, and these must be compared with previously generated ones and stored if they are new. Since even reasonable queries can give rise to large numbers of states, most state expressions are stored on disk, while a few are kept in a buffer according to a usage-age rule. Details of this and other strategies of the state graph production procedure constitute a paper in preparation.

The state graph is produced in the form of a list of transitions from each state. A transition comprises (1) an input character, (2) the next state after reading that input character, and (3) the next state output. The next state output is an indication of whether the text starting from the beginning of the section and ending with the current input character is recognized as satisfying the query. Before production of the state graph, the query is augmented slightly so that only complete sections satisfying the query are accepted by the finite state machine.

To cut down on the length of the list of transitions, certain characters are distinguished as significant for each state; the others share a default transition. In most states, only a few characters will be significant.

Table generation

The idea of distinguishing significant from default characters carries over into the design of the table used by the microprogrammed search algorithm. To explain the design of the table, let us shift our time frame from the preprocessing of the query to the execution of the search algorithm. During the search, the transition for the current character must be located among the set of transitions from the current state. This is done, in the table format described below, with a binary search among the significant characters with respect to the eight-bit unsigned value of the current character. Failure of this search causes the default transition to be taken.

For example, suppose state 1 has a transition to itself when the input is any letter, to state 2 on a blank, and to state 3 on any other input character. This portion of a hypothetical state graph is shown in Figure 3(a). The list of transitions from state 1 is shown in Figure 3(b). The list of transitions from state 1 is shown in Figure 3(b). The table generator identifies the intervals of characters (considered as eight-bit unsigned binary numbers) causing each transition. It produces a list like the one in Figure 3(c). The isolated characters in the list in Figure 3(c) are the ones against which the input character is compared in the binary search for the proper transition.

The search tree is shown in Figure 3(d). The order in which the comparisons are made is chosen by applying an easy modification of Huffman's algorithm to minimize the average search time (under the simplifying assumption that the successive characters in the text were chosen randomly and independently with given probabilities).\(^8\) The probabilities can be assigned proportionally to the relative frequencies of the individual characters in a representative sample of the data base, or the usual single-letter English probabilities can be used. This procedure, while not guaranteed optimal, should result in generally better performance than, say, choosing an arbitrary balanced tree on the same characters.

* Queries like those discussed in this section have been routinely processed and generate machines of about 50 states and 300 transitions. State graph production occurs at a rate of about 20 transitions per second of CPU time.
Figure 3(a). The locality of state 1 in a hypothesized state graph.

Figure 3(b). The transitions from state 1.

<table>
<thead>
<tr>
<th>FROM STATE</th>
<th>ON INPUT</th>
<th>TO STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>[x'00', blank)</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>blank</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>(blank, A)</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>(A, Z)</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Z</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>(Z, x'FF')</td>
<td>3</td>
</tr>
</tbody>
</table>

Figure 3(c). The transitions by EBCDIC character value. Intervals exclude the endpoints except at x'00' and x'FF'.

Figure 3(d). The search tree for state 1.

<table>
<thead>
<tr>
<th>CHARACTER</th>
<th>DISPLACEMENTS</th>
<th>ADDRESS OF TABLE SECTION</th>
<th>NEXT STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SMALLER LINK</td>
<td>FOR NEXT STATE</td>
<td>OUTPUT</td>
</tr>
<tr>
<td></td>
<td>GREATER LINK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>1</td>
<td>addr(1)</td>
<td>0</td>
</tr>
<tr>
<td>blank</td>
<td>2</td>
<td>addr(2)</td>
<td>1</td>
</tr>
<tr>
<td>Z</td>
<td>0</td>
<td>addr(1)</td>
<td>0</td>
</tr>
<tr>
<td>-</td>
<td>0</td>
<td>addr(3)</td>
<td>0</td>
</tr>
</tbody>
</table>

Figure 3(e)—The layout of the table section for state 1
The table section for a given state can be written directly from the binary search tree. Figure 3(e) shows the layout of the section of the table for the sample state discussed above. A binary search using computed addresses, although simple in concept, is a complicated algorithm by microprogramming standards; instead, to simplify next-address calculation, the displacements from each table entry to entries for greater- and smaller-valued characters are found in the table for each transition. The table is laid out so that all of the displacements are positive. A zero displacement forces the present transition. Thus, the default transition is signalled by zeros in both link fields. Also shown in Figure 3(e) is a special feature of this choice of table format: the zero in the smaller link field for the character "Z" indicates that not only is state 1 to be the next state for the letter "Z", but also for all letters smaller than "Z". After generating all the table sections for the states in the state graph, and therefore knowing the relative addresses of the table sections for each state, the table generator makes a final pass over the entire table replacing state numbers with addresses of the corresponding table sections. Finally, the table contains, for each transition, the output associated with the next state; the use of this field is described with the search algorithm below.

The internal format of a table entry, with bit addresses for each field, is shown below.

```
  0  7  8  11  12  15  0
  +--character S G address of next state 0+
     |            +--greater link
     |            |                  +--next state
     +--smaller link

Note that a table entry requires two 16-bit words, and that therefore the address of a table entry is always even. Thus, the last bit of the next state address is always zero, permitting the last bit position to store the output for the next state.

The search algorithm

Figure 4 shows a flowchart of the firmware search algorithm which accesses the above table format. The algorithm has three inputs, as previously mentioned: the text section, the table, and the 16-bit current state value, which is maintained in a microregister during the search. The operation of the algorithm is quite straightforward. Note that the algorithm can terminate in either of two ways: (1) if the input is exhausted before a state is encountered with an output of one, or (2) if a state with an output of one is encountered first. Although, in the state graph, the output is one only at the end of a complete section, the list of transitions is inspected before table generation for states which, once entered, cannot be left until the end of the section. Their outputs are set to one so that the search will stop there and the remainder of the section can be skipped.

Machine architecture

The architecture of the Microtext search machine is shown in Figure 5. Some basic statistics about the size of the machine are indicated in that figure; the microprogram occupies 243 microinstructions, of which approximately 20 percent are for the search algorithm itself, the rest being required for system and interface transfer control.

The device is initialized by writing the table and the initial state into the machine's core memory. The search command is then sent to the Meta-4. The search logic loads the initial state into a local store register where it remains for the duration of the search, and the search begins. As each character is accepted from the 370/155,
the count is incremented and a transition is taken by lookup in the table in core memory. As each new state replaces the current state in local store, it is inspected to see if its output is one. If so, the search terminates immediately by presentation of ending status to the 370/155 channel. Otherwise, the search proceeds until the channel stops sending data. When termination occurs, the search logic stores the updated state and count in core memory and the control logic takes over. On the 370/155, the channel then reads back the state and count from the machine's core memory, software records hit information if a match occurred, and the search is continued with the next block of text.

**Performance data**

It is instructive to compare the performance of a highly specialized, microcoded algorithm of this kind to a similar approach implemented in software. In this case the most appropriate comparison for the Microtext search algorithm would be to a machine language implementation of the algorithm for representative System/360 and System/370 machines, where the search would be performed on text in a buffer in the machine's main memory. The table below compares, for each method, the minimum time in microseconds to process a single character (that is, the case where the character matches the first table entry inspected by the algorithm), and the additional time in microseconds required for each subsequent probe in the table, if previous probes do not result in a match or default condition.

<table>
<thead>
<tr>
<th>Method</th>
<th>Minimum probe time</th>
<th>Additional probe time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Microcode</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Meta-4</td>
<td>4.5</td>
<td>.9</td>
</tr>
<tr>
<td>Software</td>
<td></td>
<td></td>
</tr>
<tr>
<td>370/155</td>
<td>8.9</td>
<td>8.9</td>
</tr>
<tr>
<td>370/145</td>
<td>19.6</td>
<td>19.3</td>
</tr>
<tr>
<td>360/50</td>
<td>46.8</td>
<td>47.5</td>
</tr>
</tbody>
</table>

From the figures, it can be seen that the microcoded implementation averages several times faster than the software implementation on the 370/155. It should be noted that the large difference between minimum and probe times for the Meta-4 is due to the overhead for I/O interface transfer; the software implementations need only perform an Insert Character instruction.

At its best, the Meta-4 microprogrammed implementation can scan text at roughly 220,000 characters per second, with the rate degrading to roughly 140,000 characters per second when three additional table probes are required to locate the character being processed. The lower rate is still five times faster than software on the 370/155 and 25 times faster than the 360/50 for the same case.

**FUTURE PLANS**

Future plans for the Microtext activity include completion of and refinements to the Phase I system, as well as analysis of the operation and application of the Phase I system as part of the design work for the next Phase.

**Completion of Phase I**

The goal of this activity is to bring the system to full operational standing, where its development can be frozen, and emphasis can be placed on its use and application. Extensions to the current software support are planned to make the system easier to use, and to improve the performance and capacity of the system in the translation of very complex queries. The micro-coded search algorithm has remained stable since its implementation and no further changes to it are planned.

**Planning for Phase II**

We feel very strongly that the development of application-oriented systems should proceed in parallel with the application of initial, or prototype, versions of such systems. It is only through experience in the solution of real problems, and through user feedback, that truly useful automated systems can be developed. Of necessity, then, we can at best suggest possible future directions for Microtext, with specific plans waiting...
until we have had the benefit of this application experience.

One possible future direction is toward a version of the system which would take over query translation as well as searching responsibilities from application software. The user query would be sent directly to the Microtext processor, where it would be translated into the tabular finite state machine description. The host machine would then be notified that the processor is ready and the search would begin. This version of the Microtext processor would be able to take full advantage of the experiences of developing the query translation software for the Phase I machine. We would expect that many of the basic modules in this software would become microcoded instructions in the query translation machine, with the top level of this software becoming the Phase II machine language.

A second possibility under consideration is to implement the Microtext searching capability as an adjunct to the basic control mechanism of a disk file subsystem on a general-purpose computer. This approach, which has promise for heavily I/O-bound installations, would augment the primitive sequential and keyed lookup capabilities of such devices with a facility for, say, reading only those records which match specific patterns. Rather than transfer the retrieved records directly, this extended control mechanism could perform the entire search automatically, accumulating hit data in a separate file on disk. When the search completes, the application software could then access this file as an index into the original text file searched.

CONCLUSION

This paper has described the design and implementation of a specialized microprogrammed processor which performs character string searching for full-text retrieval applications. The activity has been successful in proving the feasibility of the approach, in identifying the basic requirements of such a system, and has pointed out areas for future work. Conclusions about the utility of the system we have developed must wait until we have had the opportunity to apply the system in solution of real problems and until we have had the benefit of user feedback.

Although the Microtext project did not set out with this particular goal in mind, we feel that the success of our work to date demonstrates the utility of firmware as a tool for application system design. In recent years, microprogramming has largely been undertaken only by computer manufacturers, universities, and some research organizations, such as MITRE. Part of the reason for this is that inexpensive microprogrammable computers have not been available for experimentation, and few guidelines have been developed for the methodology of applying microprogramming in systems design. We believe that this situation is changing, and we hope that reports of practical experience such as ours with the development of Microtext will contribute to this body of knowledge.

ACKNOWLEDGMENTS

The authors wish to acknowledge the financial support of the MITRE Independent Research Program and the contributions to the design and implementation of the Microtext machine made by H. A. Bayard, E. L. Burke, G. E. DeAgazio, R. J. Fleischer, and W. L. Schiller. Additional acknowledgment for moral support and encouragement is due J. A. Clapp, E. L. Lafferty, and C. M. Sheehan.

REFERENCES

1. R. S. Glantz
   SHOEBOX—A personal file handling system for textual data
   AFIPS Conference Proceedings Fall Joint Computer Conference Vol 37 1970
   An information retrieval language for legal studies
   Communications of the Association for Computing Machinery Vol 4 1961
3. B. Liskov
   The design of the VENUS operating system
   Communications of the Association for Computing Machinery Vol 15 1972
4. K. Thompson
   Regular expression search algorithm
   Communications of the Association for Computing Machinery Vol 11 1968
5. M. J. Lewis II, P. E. Stearns
   Syntax directed translation
   Journal of the Association for Computing Machinery Vol 15 1968
6. J. A. Brzozowski
   Derivatives of regular expressions
   Journal of the Association for Computing Machinery Vol 11 1964
   Automatic generation of efficient lexical processors using finite-state techniques
   Communications of the Association for Computing Machinery Vol 11 1968
8. D. A. Huffman
   A method for the construction of minimum redundancy codes
   Proceedings IRE Vol 40 1952

From the collection of the Computer History Museum (www.computerhistory.org)