Implementing Multiple-Linked Lists in the Minicomputer-Microcomputer String Processor, C. String

1976

John H. Firestone
University of Central Florida

Find similar works at: http://stars.library.ucf.edu/rtd

University of Central Florida Libraries http://library.ucf.edu

Part of the Engineering Commons

STARS Citation

http://stars.library.ucf.edu/rtd/216

This Masters Thesis (Open Access) is brought to you for free and open access by STARS. It has been accepted for inclusion in Retrospective Theses and Dissertations by an authorized administrator of STARS. For more information, please contact leedotson@ucf.edu.
IMPLEMENTING MULTIPLE-LINKED LISTS IN THE MINICOMPUTER-MICROCOMPUTER STRING PROCESSOR, C. STRING

BY

JOHN H. FIRESTONE
B.S.E.E., University of New Hampshire, 1955

RESEARCH REPORT

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering in the Graduate Studies Program of Florida Technological University

Orlando, Florida
1976
ABSTRACT

IMPLEMENTING MULTIPLE-LINKED LISTS IN THE MINICOMPUTER-MICROCOMPUTER STRING PROCESSOR, C. STRING

BY

JOHN H. FIRESTONE

This report will demonstrate how to implement a basic linked-list data structure in RAM for C. STRING. The result of this implementation is Memory Allocation or Data Management which obtains or releases memory space as required in C. STRING.

The basic concepts of data structures such as strings, lists and stacks are discussed and the algorithm for allocation of space is developed. The C. STRING user language TOSCL, and the TOSCL parse algorithm with Data Management is described. Finally, the INTEL's Schottky Bipolar LSI microprocessor is microcoded to implement Data Management.

[Signature]
Research Report Director
# TABLE OF CONTENTS

**INTRODUCTION** .................................................. 1

**CHAPTER** .......................................................... 3

1. DATA STRUCTURES FOR C. STRING .................................. 3

2. C. STRING INTERPRETATION OF USER LANGUAGES .................. 10

3. C. STRING INTERPRETATION ALGORITHM .......................... 20

4. C. STRING SYSTEM OPERATION .................................... 30

5. P3000 MICROCODING FOR IMPLEMENTING DATA MANAGEMENT .... 36

**CONCLUSION** ..................................................... 45

**GLOSSARY** ....................................................... 47

**LIST OF REFERENCES** ............................................ 49
INTRODUCTION

C. STRING is a minicomputer-microcomputer system which is based on string and list processing. String and list processing may be defined as the act of operating on an input string to produce an output string as specified by an input program string.

The goal of C. STRING is to investigate string and list processing based user languages and to provide the facilities for these investigations with extensible microprocessor systems.

A major drawback, presently in C. STRING is that strings are stored in contiguous locations in RAM, permanently assigned before run time to the program processors. String processing is characterized by dynamic patterns of storage requirements since the program and objects of computation can be one in the same and can vary dramatically in length. Because the user has no sure way of telling ahead of time how much space and where it should be allotted, much assigned memory space is not used or available where needed and pro-
grams can and do needlessly run out of memory space.

The intent of this report is to make the allocation of computer space the task of a microprocessor that answers requests to release and to obtain memory space as required by other processing programs. The release process involves putting a list link element into a free link element list. When a request to obtain space is made, a free list link element from the free link element list is supplied to the requesting processor.

Since only one microprocessor (an INTEL 3000 system) has been implemented to date, this system (P3000) is used to implement the first Data Management processing program.

The INTEL 3000 Microcomputer System is interfaced to the Data General minicomputer and the 3000 parsing processor operates at ten times the maximum data channel rate of the Data General system. The INTEL 3000 system was chosen for its speed and it significantly upgrades the performance of former software implemented processing functions.
CHAPTER 1
DATA STRUCTURES FOR C. STRING

1.0 C. STRING strings

C. STRING is a minicomputer-microcomputer system that operates on input strings to produce an output string as specified by an input program string. The strings can be text material or executable procedures or microcode for implementing system facilities. String handling facilities are a desirable base for a system designed to interpret user languages and to investigate multiple processor systems (Petrasko, 1976).

In C. STRING, strings are to be implemented as a list of multiple linked segments where each segment contains up to 254 string data elements and two link elements.

1.1 BASIC DATA STRUCTURE

For problems involving the storage of large quantities of data, it becomes apparent that computer system designers, as well as programmers must organize the problem solving procedures so as to economize on
computer operating time and storage space. The application of most data structures then involves the choice of space versus time tradeoffs in hardware, firmware and software design. For example, the choice of fast running programs that use lots of memory space versus slower running programs that use less memory.

Dynamic data structures (that grow or shrink in a given boundary) require many links, lists, and bookkeeping to keep track of current locations of data and available space. This is contrasted with a static structure, which may require only a pointer to designate the beginning address of a data area and a value to denote its length.

1.2 LISTS, STRINGS, AND LINKS

Most computers have a sequentially ordered memory with each location having a unique address. Storage of data in sequential or contiguous locations of memory is a data structure called a list.

When the data stored in a contiguous list is a series of characters it is commonly called a string. The characters handled in C. STRING are ASCII coded. A string that contains no characters is called a null
<table>
<thead>
<tr>
<th>ADDRESS or LOCATION</th>
<th>MEMORY CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>(BINARY) PAGE LINK-ADR.</td>
<td>(DECIMAL)</td>
</tr>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>64</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>500</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>001 1111111</td>
<td>513 F.L.</td>
</tr>
<tr>
<td>010 00000000</td>
<td>NULL HEADER</td>
</tr>
<tr>
<td></td>
<td>NULL</td>
</tr>
<tr>
<td>010 1111111</td>
<td>SEGMENT 1</td>
</tr>
<tr>
<td>011 00000000</td>
<td>769 F.L.</td>
</tr>
<tr>
<td></td>
<td>766 B.L.</td>
</tr>
<tr>
<td>011 1111111</td>
<td>SEGMENT 2</td>
</tr>
<tr>
<td>100 00000000</td>
<td>1025 F.L.</td>
</tr>
<tr>
<td></td>
<td>1022 B.L.</td>
</tr>
<tr>
<td>1278</td>
<td>SEGMENT 3</td>
</tr>
<tr>
<td>1279</td>
<td>1281 F.L.</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>512*(N-1)256</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>767*(N-1)256</td>
<td></td>
</tr>
</tbody>
</table>

FIG. 1. SEGMENTS, MULTI-LINKED LIST
Often times a sufficient amount of contiguous memory is not available to hold the entire list of characters. It is then necessary to segment the string and place it into two or more contiguous areas of memory. These areas can then be linked and the result is a linked list which can be made to appear as a string. As an example, in memory a string segment could occupy locations 512 through 767, a second segment could occupy locations 768 through 1023, ... and an Nth segment could occupy locations 512 + (N-1) 256 through 767 + (N-1) 256. See Figure 1.

If segments 1 through N above had 2 of the 256 locations to point to other segment locations rather than for storing characters, these locations would contain links (Pointers). Figure 1 shows the data structure we have described in memory and it is commonly called a doubly or multilinked list. The first and last locations of each segment contain links respectively linking backward to predecessor segments and forward to successor segments. An initial segment called HEADER and a final segment called TAIL could be added to the
list. These list elements may contain information about the string such as name, creation date, length, etc. The link at location 511 links forward to the first location in string segment 1 which may contain a character, while the link at location 500 is a null to indicate the end of the HEADER area. Segment 1 has a forward link (F.L.) at location 767 linking forward to location 769 of segment 2, while the backward link (B.L.) at location 512 is a null also, to indicate that this is the last string segment. Segment 2 contains a backward link (B.L.) linking to location 766 of segment 1. The list would end at the null link of Segment N. (Lewis, 1976).

The addresses in the figure are shown in both decimal and binary. Note that the eight least significant bits (lsb) of the binary addresses of F.L.'s and B.L.'s are all 1's ($\land\text{Link}(7:0) = 1$) and all 0's ($\lor\text{Link}(7:0) = 0$), respectively, where the notation in parenthesis describes the logical and/or of these 1's or 0's (Hill, 1973). This logical notation will be used later to describe the location of a Link. (ie, F.L. or B.L.) Also the decimal equivalents of the
remaining bits (msb) are called page numbers. For example, segment 3 is on page 4.

An additional pointer at the lower position in memory (location 64) can be used to point to the first information location available in HEADER or segment 1. This first pointer at location 64, is called a GLOBAL pointer, since any calling system of C. STRING can easily access page zero, location 64 directly. For example, the NOVA 1220 minicomputer instruction; LDA 2, 64, would cause the contents of location 64 to be loaded into the number 2 accumulator, ie, AC2 ← (loc. 64). AC2 contents are now a pointer to HEADER.

1.3 STACKS

The stack can best be described as a last in first out, LIFO, storage structure. In C. STRING the list structure we have described in Figure 1 is used as a stack, with the constraint that insertions and deletions of characters are restricted to one end of the list. Using the contents of an accumulator as a Pointer, the following algorithms illustrate reading out (POPPING) and storing (PUSHING) characters (єs) in a stack. See figure 1.
POP $\emptyset$: 1. Do one of the following:

(a) If Pointer points to a non-link element, go to 2.
(b) If Pointer points to a F.L., (Pointer) ← (Pointer-1), go to 1.
(c) If Pointer points to a B.L. and B.L. = null, exit (end of STACK).
(d) If Pointer points to a B.L., store B.L., then (Pointer+1) which is a F.L. is PUSHED to the Free link element list (STACK); Pointer ← B.L., go to 1.

2. Read out character, decrement Pointer, wait.

PUSH $\emptyset$: 1. Do one of the following:

(a) If Pointer points to a non-link element, go to 2.
(b) If Pointer points to a B.L., increment Pointer, go to 1.
(c) If Pointer points to a F.L. and the F.L. is null, exit (end of list).
(d) If Pointer points to a F.L., POP $\emptyset$FS from the Free link element STACK (FS), MEM (Pointer) ← $\emptyset$FS, TEMP ← (Pointer-1), Pointer ← $\emptyset$FS, decrement Pointer, MEM (Pointer) ← TEMP, increment Pointer, go to 1.

2. Write character, increment Pointer, wait.

In POP $\emptyset$, instruction 1. (d) illustrates the memory space release process where a list link element (F.L.) is pushed to a free link element list (STACK) or FS. See Glossary.

In PUSH $\emptyset$, instruction 1 (d) illustrates the
memory space retrieving process, where a list element (F.L.) is POPPED from the free link element list (FS) and pushed to another stack, eg. the Active Stack, AS. The B.L. of the AS is generated and stored in register TEMP. The pointer to the new segment (F.L.) is decremented and the stored B.L. is pushed to the AS. The Pointer is incremented and we go to 1. Since instruction 1. (a) is now applicable, we go to instruction 2.
CHAPTER 2

C. STRING INTERPRETATION OF USER LANGUAGES

2.0 USER LANGUAGES - TOSCL

TOSCL is the first user language that runs on C. STRING. TOSCL is an interpretively executed functional language which is based on TRACR (Text Reckoning and Compiling devised by Mooers and Deutsch). A TOSCL functional statement has the form of pre-fix POLISH notation and has three control symbols which govern function evaluation. The symbols are # (for active mode evaluation), ## (for neutral mode evaluation) and ( for quote mode evaluation (Mooers, 1966; Sammet, 1969).

In C. STRING an INTEL microprocessor (P3000) is used to parse TOSCL program strings. Parsing consists of identifying and marking function and argument sub-strings by the insertion of control characters. For parsing, program strings are first stored in the Active Stack (AS). After parsing, the modified or parsed strings are stored in the Neutral Stack (NS), as shown in Figure 2. For the strings on the left in Figure 2 the underlines represent a string of characters and the
commas divide the strings into arguments up to the closing parenthesis. The form of pre-fix POLISH notation is such that the first argument is the operator or primitive function identifier and the subsequent arguments are the required operands for the particular primitive.

**BEFORE PARSE**

TOSCL STRINGS IN AS

**AFTER PARSE**

PARSED STRING in NS

**MODE**

Active  

Neutral  

Quote  

Non

Control  

Non

Control

Restored

Fig. 2. Parsing Toscl Strings

The primitives in TOSCL are two-letter mnemonics that indicate the operation to be performed on a specified number of operands. For example, the string could be

```
## ( DS, XX, HELLO )
```
This means Define a String **HELLO** with the name assigned as **XX**. The PARSER (P3000) would parse ##( into one character $+$ indicating the beginning of a function which is to be evaluated in the Neutral mode. The PARSER would hash or encode the primitive Define String **DS** into an address, represented here as the hashed character $H$. The comma would be parsed into the character $+$. So far we have:

$+$ $H$ $+ $ $+$ $X$ $X$ $+$ $HELLO$

These and the subsequent operand strings and $+$ delimiters would then be passed to the Neutral Stack, **NS**. Finally, the parsed string in the **NS** would be:

$+$ $H$ $+$ $X$ $X$ $+$ $HELLO$

When the closing parenthesis was detected the PARSER would pass control to the **EVALUATOR**.

The **EVALUATOR** would search the **NS** for $+$ and detect this as the Neutral mode. Next, the **EVALUATOR** would detect $H$ (the hashing of the primitive) and go to a directory of primitives, then locate and execute the primitive operation.

The primitive Define **String DS** in the TOSCL language is a primitive that causes strings to be stored...
in a string storage area, Form Storage. The execution of DS causes the data string \texttt{HELLO} to be stored in Form Storage and the name string \texttt{XX} to be stored in the HEADER of the string.

2.1 \textbf{NEUTRAL MODE}

If subsequently another string in the AS were:

\[
\#\#(\texttt{CL, XX})
\]

(meaning \texttt{Call} or bring forth from Storage the data string named \texttt{XX}) It would become after parse in the NS:

\[
\#\#(\texttt{XX})
\]

When the closing parentheses was detected the PARSER would pass control to the EVALUATOR. The EVALUATOR would search the NS for the \# and detect this as the Neutral mode. Next, the EVALUATOR would detect \# (the hashing of the primitive) go to the table of primitives and then execute the primitive operation.

This specified operation would cause the action (Call) a string named \texttt{XX} from Form Storage) and produce the result string \texttt{HELLO} and cause it to be stored in the Neutral Stack. The PARSER deletes the substring in the AS that it parses and the EVALUATOR deletes the parced substring it evaluates in the NS, and the result
string of the evaluation is stored in the NS.

It can be seen that the execution of the function with DS produced an action (creating Form Storage) with no result string (null string). The execution of the function with CL produced an action (ie, called a string from Form Storage) and produced a result string HELLO.

The result strings of primitive functions evaluated in the Neutral mode are pushed to the Neutral Stack.

2.2 **QUOTE MODE**

In the quote mode, the strings within the matching parenthesis are passed by the PARSER to the NS unmodified. Hence, a function can be protected from evaluation. The following string in the AS:

```
(#(DS,XX, HELLO))
```

would be after parse:

```
#(DS,XX,HELLO)
```

in the Neutral Stack. Again the PARSER deletes the substring in the Active Stack that was parsed.

2.3 **ACTIVE MODE**

The result strings of primitive functions evaluated in the Active Mode are pushed to the Active Stack
For example, suppose we had in Form Storage both a string `HELLO` named `XX` and a string `##(CL,XX)` named `YY`. If we then had the following string in the AS:

```
##(CL,YY)
```

it would be after parse

```
∃ H \neq YY
```

in the NS.

When the closing parenthesis was detected, control would be passed to the EVALUATOR. The EVALUATOR would search for `∃` and next detect `H` (the hashing of the primitive `CL`), go to the table of primitives and execute the primitive operation in the Active Mode. This operation would produce a result string:

```
##(CL,XX)
```

This then is pushed to the AS and after parse would be

```
∃ H \neq XX
```

in the NS.

When the closing parenthesis is detected, control would be passed to the EVALUATOR. The EVALUATOR would search for `∃` and next detect `H`, go to the table of primitives and execute the primitive operation in the Neutral Mode. This operation would produce a result
string that would be pushed to the Neutral Stack.

2.4 **NON CONTROL - NON CONTROL RESTORE**

When the PARSER detects a string of characters with no lead control characters, there are no modes or functions to be evaluated. The strings are passed unmodified to the Neutral Stack.

If a # or ##...# is detected in a string but is not followed by a ( to identify an Active or Neutral mode function, the PARSER passes control to RESTORE which is actually another PARSER state. The PARSER (with RESTORE) thus insures that the following sub-string starting in the AS;

```
#\#\#\#(DS,XX,HELLO)
```

would be after parse

```
##\#\# XX HELLO
```

in the Neutral Stack.

2.4 **NESTING**

Arguments in TOSCL can be function strings. This provides for unlimited nesting. TOSCL strings are evaluated left to right and innermost out.

For example, another of the numerous primitives in TOSCL is **Print String.** A program with PS can illustrate
what happens in the interpretation of TOSCL with a typical string:

\[ \#(PS, \#(DS, XX, HELLO), \#(CL, XX)) \]

The first closing parenthesis is that of function 1. Evaluation of function 1 produces a null string and creates Form Storage as mentioned before. The next closing parenthesis is that of function 2. Evaluation of function 2 produces the result string `HELLO`. At this point we have essentially

\[ \#(PS, HELLO) \]

The next closing parenthesis is that of function 3. The primitive PS evaluation causes an output device to print out the operands, and the result string of function 3 is null.

2.5 **IDLING PROGRAM - READ STRING, PRINT STRING**

The initializing TOSCL program for C. STRING is the Idling program. This program sends the string `\#(PS, \#(RS))` to the empty Active Stack. The initial evaluation of `\#(RS)` will result in a question mark printed by an output device. This is the call to the user to input a TOSCL program. If the user inputs a
program terminated by a special character (meta), the program will be stored in the AS replacing #(RS) and will be parsed and evaluated. The result string will be printed out on the output device. The AS will become empty and will again receive #(PS,#(RS)) and the question mark will again be printed signaling that C. STRING is ready for another program input. The Idling program thus provides for communication between C. STRING and the user.

A user can enter the following example subprograms that will provide for communication between C. STRING and the user during a program:

```c
###(DS,XX,#(RS)) ; What is read in is evaluated and then stored.
###(DS,XX##(RS)) ; What is read in is not evaluated, but stored.
###(DS,XX,(#(RS)) ; What is stored is the function string #(RS).
```
CHAPTER 3

C. STRING INTERPRETATION ALGORITHM

3.0 PARSE - EVALUATE CYCLE

C. STRING interprets TOSCL program strings by alternating actions of a PARSER and an EVALUATOR. The PARSER examines input program strings for control and data structure information. If a control character is detected, the strings are parsed as was illustrated in Figure 2. If an operation requiring an evaluation is detected, control passes to the EVALUATOR, which may produce an action or produce result strings for the AS or NS. Then control is passed back to the PARSER.

In the following section the POPPING and PUSHING algorithms developed in Chapter 2, are integrated with the parser-evaluate cycle. See figure 3A.

3.1 PARSER (P3000)

The PARSER P3000 pops the active stack (POPAS), then examines the stack element for control and data structure information (links).

The algorithm shown in Figure 3A, accomplishes string maintenance (Data structure Management) through
FIG. 3A. P3000 ALGORITHM
FIG. 3B. AS DATA MANAGEMENT

FIG. 3C. NS DATA MANAGEMENT
link detection, and implies the parsing and evaluation operation through control character recognition. If the PARSER detects an AS link, control is passed to AS Data Management.

In Figure 3B, the AS Data Management algorithm detects the °AS; if it is null, this means we are at the end of the AS and control is then passed to the Idle program and the PARSER Halts. If °AS isn't null, then a list link element (F.L.) is pushed or returned to the free link element list (Free Stack) which essentially returns a list element (segment) for use elsewhere. The pointers are updated and AS Data Management returns control to the PARSER.

If a non-control character ∪ is detected, ∪ is pushed to the NS. If the PARSER detects a NS link, control is passed to NS Data Management. See Figure 3C. NS Data Management performs a series of pops and pushes. The first is popping the Free Stack (FS). If the link supplied by the FS is a null, this could result in a Wait (in the case of parallel processing) or a signal to indicate "out of space".

If the FS link isn't a null, then the algorithm
pushes a F.L., a B.L., and a \( \land \) character, to the NS. Since the Free Stack (FS) is popped in the process, NS Data Management has retrieved a segment of memory space. NS Data Management updates the pointers and returns control to the PARSER.

If instead of \( \land \), a control character was detected, then control passes to the control character microprograms of the PARSER. The processing of each function requires popping the AS and pushing the NS along with the attendant requirements of Stack Data Management. These actions are illustrated in the state diagram discussed in the following section.

### 3.2 STATE DIAGRAM - P3000

The State diagrams of Figure 4A and 4B present the effective states of the P3000 system. In STATE 0, initializing consists mainly of storing the pointers of the Active, Neutral and Free Stacks, (ASP, NSP, and FSP), and various constants. If the Idle program (STATE 3) for example, was returning control to (STATE 1) the current pointers, ASP and NSP would be initialized to starting addresses.

Figure 4A illustrates the Data Management states.
FIG. 4A. STATE DIAGRAM, P3000
FIG. 4B. STATE DIAGRAM, P3000
In STATE 1, the AS is popped (POPAS). If an AS link is detected (ie, ∨/ASP(7:0)=0) we move to STATE 2 with outputs $\mathcal{C}AS$, ASP, and FSP from STATE 1. If the AS character popped ($\mathcal{C}AS$) was a null (ie, $\nabla/\mathcal{C}AS (15:0)=0$), we move to State 3, then to State 4. The Idle program does not need pointers, since this means the AS is empty. Also, since #(PS,$\#(RS))$ length is only eleven characters, Idle has no need of AS Data Management.

If in STATE 2, $\mathcal{C}AS$ isn't null, then we move to State 5 with the outputs $\mathcal{C}AS$, ASP and FSP of STATE 2. STATE 5 performs the AS Data Management algorithm of Figure 3B, and then returns to STATE 1 with new pointers ASP and FSP.

It can be seen that STATES 1 through 5 represent the complete AS Data Management loop. This loop is indicated in Figure 4B by the mnemonic Pop Active (PA) in STATE 1.

In Figure 4A we move from STATE 1 to STATE 6, if there isn't an AS link detected. If the $\mathcal{C}AS$ is a non control character $\chi$ we move from STATE 6 to STATE 13. In STATE 13 the $\chi$ is pushed to the NS (PUSHNS). If an NS link is now detected (ie, $\Lambda/\text{NSP}(7:0)=1$) we move to
STATE 14 NS Data Management. STATE 14 (and STATE 15) performs the algorithm of Figure 3C.

From STATE 14 we move to STATE 15 if the link popped from the Free Stack is a null (ie, \( \forall_{\text{FS}}(15:0)=0 \)). The wait mode is illustrated for the case where parallel processing may free up a segment and we return to STATE 14 with an F.L. and FSP. If we have our F.L. (and not null) we move from STATE 14 back to STATE 1 with a new NSP and FSP.

The STATES 13 through 15 represent the complete NS Data Management loop and this is indicated in Figure 4B by the mnemonic Push Neutral (PN) in STATE 13.

In Figure 4A it has been shown that we move to STATE 6 when a control character is popped from the AS and there is no AS link. Figure 4B describes micro-code for processing control characters (Ref: Fig. 3A). We move from STATE 6 to the state dictated by the control character. If a comma is detected, the parser moves to STATE 7 and \( \# \) is pushed to the NS with attending NS Data Management as indicated by PN.

If an opening ( is detected we move from STATE 6 to STATE 11 Quote Mode. In this STATE the PARSER per-
forms a sequential series of PS's and PN's indicated until the matching closing ) is detected, at which point we return to STATE 1.

If an opening # is detected and we are not in STATE 11 we move to STATE 8 Begin Function Search. If the opening # is followed by ( or #( we move to STATE 9 to parse an Active or a Neutral mode function. In STATE 9, the PARSER performs two more PA's in order to obtain the primitive identifiers. It performs a hash and sends the function mark and primitive symbol (hashed) to the NS.

When the # or ## is not followed by a ( we move to STATE 10 Non Control Restore (also see Figure 2). In STATE 10 the characters # or ## are pushed to NS as indicated by the PN's.

When the closing ) is detected, a function has been parsed and control passes to the EVALUATOR (Cotton, 1974).
CHAPTER 4

C. STRING SYSTEM OPERATION

4.0 C. STRING

C. STRING is a string processing system being built for specific goals. One is to provide for firmware parsing of diverse input languages, TOSCL being the first. The parsing is performed by the INTEL microcomputer set (P3000). The second goal is to provide hardware implementation for extensibility in string processing languages. The DATA GENERAL 1220 minicomputer system is the host and provides facilities including a subset of the standard TRAC R language.

Supporting hardware and software include a microcomputer - Data General interface, a 60 nsec Control RAM, an octal editor, an assembler for a subset of Intel's microcomputer instruction set, and a microprogram debug program.

The current parser operates at ten times the maximum data channel rate of the Data General minicomputer system and it significantly upgrades the performance of the software-firmware implemented language proccessor.
4.1 PROCESSOR P3000 DATA PATHS

Figure 5 illustrates the parser of the C. STRING system, the microprocessor P3000. It is composed of a Control RAM called CRAM, an INTEL 3001 (MCU) chip and eight INTEL 3002 (CPE) chips.

The CRAM is tied directly to the Data General 1220 minicomputer (D.G.) Data Bus and is also accessed by the MCU. The SCU selects either the CRAM output word or the D.G. control instruction word and is latched in the pipeline register L1.

P3000 address and control information is converted to 3001 function information by the logic network D1. For this application this input to D1 is the 8 lsb bits of the control word. SCU and CPE inputs are provided by the logic network D2. The input to D2 is the 8 msb bits of the control word.

The microcomputer address and data buses are tied to the D.G. I/O bus. The address (A) outputs from the CPE provide the ASP, NSP, or FSP which are used to access the AS, NS and FS in the main memory of the minicomputer. The characters that are read from main memory are deposited in register L2.
FIG. 5. C. STRING SYSTEM DATA PATHS
The actions following a read command (issued by the SCU) brings a character (CAS) into L2 and the Ready command (issued by the minicomputer) enables D3. D3 decodes CAS and the address (on the A lines) and puts the result into latch L3. The output of L3 is the input to the PX lines of the MCU. The information in latch L3 is accessed by using the MCU conditional jump instruction, JPX, in the P3000 control word.

4.2 P3000 SUPPORTING SOFTWARE - CONTROL WORD

4.21 SOFTWARE

The P3000 system support software consists of an assembler (D3000), a debugger, a microcode loader and an octal editor.

The D3000 assembler allows the programmer to write INTEL programs using INTEL symbolic mnemonics as opposed to writing in machine language (direct octal). The Octal editor allows one to write and edit an octal program if desired. The D3000 assembler was constructed using the D. G. Macro-assembler. The need for efficient interaction with a sixteen bit minicomputer (DG1220) led to the decision to use a 16 bit control word for the P3000 system.
4.22 **CONTROL WORD FORMAT**

Each control word as shown below is divided into four fields:

1. The function or OP field which consists of an INTEL mnemonic code (Bits 0-2).

2. The Operand or register field which specifies the CPE register (Bits 3-6).

3. The flag field for the Carry out flip-flop (Bit 8), for the Carry in line (Bit 9).

4. The jump code and displacement field (Bits 10-15).

A line of source code consists on an INTEL mnemonic followed by up to two suffixes H and I. H is Bit 8 and if 1, specifies the Carry out flip-flop is to be held constant. If 0, the Carry out flip-flop is to be changed to the state of the Carry Out of the CPE. I is Bit 9 and if it is 1 Carry in (CI) is 1. If Bit 9 is 0, CI is zero. Following these are register selecters followed by the jump code with displacement.

An example code line is: **ILRHI, R4, JCR, 3**.

**Source Word:** ILR R4 HI JCR 3

**Control Word:**

<table>
<thead>
<tr>
<th>Field</th>
<th>000</th>
<th>0100</th>
<th>011</th>
<th>01</th>
<th>0011</th>
</tr>
</thead>
<tbody>
<tr>
<td>OP</td>
<td>REG.</td>
<td>KHI</td>
<td>JP</td>
<td>DISP.</td>
<td></td>
</tr>
</tbody>
</table>

The control word will cause the following action:
1. AC, R4 ← R4 + 1
2. Carry out flip-flop will be changed.
3. JMP in current row of CRAM to column 3.
CHAPTER 5

P3000 MICROCODING FOR IMPLEMENTING DATA MANAGEMENT

5.1 P3000 MICROPROGRAMS

The microprogram developed in this section is an interpretation of the algorithms of Figure 3 and the State diagram of Figure 4A. This microprogram is designed to be an extension of the current parsing microprogram in the P3000, which does not yet incorporate the Data Structure Management illustrated in the state diagram of Figure 4B.

5.2 P3000 AS DATA MANAGEMENT MICROCODING

Figure 6A shows the AS Data Management microprogram flowcharted using only P3000 OP and Register fields. Each instruction is numbered and the AHPL definitions are indicated in the box on the left of the instruction. (Hill, 1973). Along the right-hand upper side of Figure 6A are parts of the AS and FS along with the specific links and characters used as examples. These are added to illustrate the AS Data Management portion of P3000 Microcode.

In the P3000, MAR is the CPE address output reg-
**FIG. 6A. MICROPROGRAM FLOWCHART—AS DATA MANAGEMENT**
ister. AC is the accumulator and also the CPE data output register. The symbols ASP, FSP, NSP and TEMP are labels assigned to four of the eleven CPE registers. Conditional jumps are specified using jump code field symbols.

For our example, initially the FSP is pointing to location xxx250 in the FS and ASP is pointing to location 1024 of segment 3 with contents 766 in the AS. See the upper right-hand corner of Figure 6A showing these quantities.

Instruction 0 puts ASP in the CPE address output register MAR, ie, 1024. Instructions 1 through 3 get the $\mathcal{A}$S (766) from main memory and deposit it in the AC of the CPE (POPAS). The conditional jump (JPX of instruction 3) detects the location as a link.

Since we have a link location we move to instructions 4 and 5 which perform the test on the $\mathcal{A}$S (which is a B.L.) for null. The B.L. is added to a constant (all 1's). If there is an overflow in the computation, the Carryout of the CPE will be 1. This indicates the B.L. isn't null and we move to instruction 8. If there had been no overflow in the computation, Carry out would
have been zero (i.e., \( B.L. = \text{Null} \)) and we would have moved to instructions 6 and 7 to call IDLE and then the P3000 would halt. The conditional jump of JFL of instruction 5 detects the Carry Out of the CPE.

Instructions 8 and 9 increment \( ASP \) by 2 to get the \( F.L. (1025) \). In instruction 10 the \( FSP \) is incremented by 1 and \( FSP = 251 \). In steps 11, 12, and 13 the \( F.L. \) is written into the \( F.S. \) (\text{PUSHFS}). Finally in instructions 14 to 16, a new \( ASP \) is generated (i.e., \( ASP \text{ Final} = 766 \)) and we return to Instruction 0.

The nature of the 3001 address control logic required that special attention be given to conditional jumps. (Hoff, 1975). The mapping or writing of an instruction in CRAM with the JFL conditional jump, for example, dictates that it be placed anywhere in column 1 (given a row). If the Carry Out is a 1, we will jump to column 3, if Carry Out is a 0 we will jump to column 2. See instruction 5 and note its placement in the memory map of Figure 7, with respect to possible subsequent instructions 6 and 8. Instruction 5 is in row 1, column 1 of CRAM.

The P3000 program for As Data Mgt. is shown in
FIG. 6B. MICROPROGRAM FLOWCHART-NS DATA MANAGEMENT
FIG. 7. MEMORY MAP, P3000
Figure 8 with the jump and displacement assignments formulated in the memory map of Figure 7. The JPX operation in Instruction 3 will lead to locations in CRAM determined by the PX inputs of the 3001 element. These inputs are shown decoded into control characters above the column numbers in Figure 7.

5.2 NS DATA MANAGEMENT

The P3000 program for NS Data Management (Instructions 17 through 43) is shown in Figure 8, and the microprogram flowchart if Figure 6B. The pops and pushes are marked in accord with the algorithm of Figure 3C. The memory mapping of NS Data Mgt. instructions 41, 42, and 42 (shown in Figure 7.) for example, appear in CRAM locations 64, 65, and 66 respectively, in the P3000 program.

5.3 C. STRING DATA MANAGEMENT

The P3000 source program shown in Figure 8, implements a basic linked-list data structure in the RAM of the D.G. minicomputer and provides C. STRING with Data Management.
<table>
<thead>
<tr>
<th>LOCATION (Decimal)</th>
<th>INSTRUCTION NUMBER</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>.LOC 0</td>
</tr>
<tr>
<td>16</td>
<td>1</td>
<td>LMI, ASP, JCC, 1</td>
</tr>
<tr>
<td>17</td>
<td>5</td>
<td>CSR, ASP, JCR, 5</td>
</tr>
<tr>
<td>18</td>
<td>6</td>
<td>ALR, TEMP, JFL, 1</td>
</tr>
<tr>
<td>19</td>
<td>8</td>
<td>INTERRUPT, JCC, 2</td>
</tr>
<tr>
<td>20</td>
<td>44</td>
<td>INRI, ASP, JCC, 2</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>NOP, JCR, 5</td>
</tr>
<tr>
<td>22</td>
<td>3</td>
<td>WAIT, JCR, 6</td>
</tr>
<tr>
<td>23</td>
<td>17</td>
<td>LMM, AC, JPX, 1</td>
</tr>
<tr>
<td>24</td>
<td>x</td>
<td>LMI, NSP, JCC, 2</td>
</tr>
<tr>
<td>25</td>
<td>x</td>
<td>, JMP /</td>
</tr>
<tr>
<td>26</td>
<td>x</td>
<td>, JMP F.L.</td>
</tr>
<tr>
<td>27</td>
<td>x</td>
<td>, JMP #</td>
</tr>
<tr>
<td>28</td>
<td>x</td>
<td>, JMP )</td>
</tr>
<tr>
<td>29</td>
<td>4</td>
<td>, JMP (</td>
</tr>
<tr>
<td>30</td>
<td>x</td>
<td>CSR, TEMP, JCR, 1</td>
</tr>
<tr>
<td>31</td>
<td>x</td>
<td>, JMP )</td>
</tr>
<tr>
<td>32</td>
<td>40</td>
<td>, JMP NC</td>
</tr>
<tr>
<td>33</td>
<td>39</td>
<td>ILR, T, CC, 4</td>
</tr>
<tr>
<td>34</td>
<td>7</td>
<td>INRI, NSP, JCRO</td>
</tr>
<tr>
<td>35</td>
<td>9</td>
<td>HALT,</td>
</tr>
<tr>
<td>36</td>
<td>10</td>
<td>INRI, ASP, JCC, 4</td>
</tr>
<tr>
<td>37</td>
<td>11</td>
<td>INRI, FSP, JCR, 5</td>
</tr>
<tr>
<td>38</td>
<td>12</td>
<td>ILR, ASP, JCR, 6</td>
</tr>
<tr>
<td>39</td>
<td>18</td>
<td>LMI, FSP, JCC, 3</td>
</tr>
<tr>
<td>40</td>
<td>31</td>
<td>WAIT, JCC, 3</td>
</tr>
<tr>
<td>41</td>
<td>28</td>
<td>WAIT, JCR, 14</td>
</tr>
<tr>
<td>42</td>
<td>29</td>
<td>ALR, TEMP, JFL, 3</td>
</tr>
<tr>
<td>43</td>
<td>30</td>
<td>INTERRUPT,</td>
</tr>
<tr>
<td></td>
<td></td>
<td>LMI, NSP, JCR, 8</td>
</tr>
</tbody>
</table>

Fig. 8 P3000 Program
<table>
<thead>
<tr>
<th>LOCATION (Decimal)</th>
<th>INSTRUCTION NUMBER</th>
<th>INSTRUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>44</td>
<td>27</td>
<td>CSR, TEMP, JCR, 9</td>
</tr>
<tr>
<td>45</td>
<td>26</td>
<td>LMM, AC, JCR, 12</td>
</tr>
<tr>
<td>46</td>
<td>32</td>
<td>SDR, R1, JCR, 15</td>
</tr>
<tr>
<td>47</td>
<td>33</td>
<td>CSR, NSP, JCC, 3</td>
</tr>
<tr>
<td>48</td>
<td>16</td>
<td>ALR, ASP, JCC, 0</td>
</tr>
<tr>
<td>49</td>
<td>38</td>
<td>SDRI, NSP, JCC, 2</td>
</tr>
<tr>
<td>50</td>
<td>37</td>
<td>ILR, R1, JCR, 1</td>
</tr>
<tr>
<td>51</td>
<td>36</td>
<td>WAIT, JCR, 2</td>
</tr>
<tr>
<td>52</td>
<td>15</td>
<td>CLR, ASP, JCR, 0</td>
</tr>
<tr>
<td>53</td>
<td>14</td>
<td>ILR, TEMP, JCR, 4</td>
</tr>
<tr>
<td>54</td>
<td>13</td>
<td>WAIT, JCR, 5</td>
</tr>
<tr>
<td>55</td>
<td>19</td>
<td>ILR, MSK, JCR, 8</td>
</tr>
<tr>
<td>56</td>
<td>20</td>
<td>ORR, AC, JCR, 9</td>
</tr>
<tr>
<td>57</td>
<td>21</td>
<td>ILRI, AC, JFL, 3</td>
</tr>
<tr>
<td>58</td>
<td>22</td>
<td>INRI, NSP, JZR, 0</td>
</tr>
<tr>
<td>59</td>
<td>23</td>
<td>LMI, FSP, JCR, 12</td>
</tr>
<tr>
<td>60</td>
<td>24</td>
<td>CSR, FSP, JCR, 13</td>
</tr>
<tr>
<td>61</td>
<td>25</td>
<td>WAIT, JCC, 2</td>
</tr>
<tr>
<td>62</td>
<td>35</td>
<td>LMI, R1, JCR, 3</td>
</tr>
<tr>
<td>63</td>
<td>34</td>
<td>ILR, NSP, JCR, 14</td>
</tr>
<tr>
<td>64</td>
<td>41</td>
<td>LMI, NSP, JCR, 1</td>
</tr>
<tr>
<td>65</td>
<td>42</td>
<td>INRI, NSP, JCR, 2</td>
</tr>
<tr>
<td>66</td>
<td>43</td>
<td>WAIT, JZR, 0</td>
</tr>
</tbody>
</table>

.END

Fig. 8 P3000 Program Cont.
CONCLUSION

The preceding chapters have demonstrated how to implement linked data structures for a string processor, C. STRING.

This implementation of Data Structure Management provides allocation of memory space as required and overcomes the problems associated with permanently assigned memory areas.

In order to understand the facilities needed by C. STRING for processing TOSCL with Data Structure Management, it is necessary to first comprehend data structures; next, the basic concept of the user language TOSCL, that programs consist of strings containing sequences of functions which can be nested; finally to utilize the INTEL 3000 microcomputer with additional hardware to implement these facilities.

The conclusions of this report are:

1.) C. STRING is easily adaptable to list and string processing utilizing multiprocessors such as the 3000 system.

2.) The use of firmware for the design of peripheral devices is a viable and cost effective
approach, and it upgrades the performance of language processing.

3.) The movement of other traditionally software functions can just as effectively be converted to firmware. This is in keeping with the extensible goal of C. STRING.
## Glossary

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AHPL</td>
<td>A Hardware Programming Language (Hill, 1973).</td>
</tr>
<tr>
<td>ASCII</td>
<td>TELETYPEx CODE</td>
</tr>
<tr>
<td>AS</td>
<td>Active String (Stack)</td>
</tr>
<tr>
<td>ASP</td>
<td>Active String (Stack) Pointer</td>
</tr>
<tr>
<td>B.L.</td>
<td>Backward Link (Pointer)</td>
</tr>
<tr>
<td>⦿</td>
<td>Character</td>
</tr>
<tr>
<td>⦿AS</td>
<td>Active String (Stack) Character</td>
</tr>
<tr>
<td>CPE</td>
<td>Intel Central Processor Element</td>
</tr>
<tr>
<td>F.L.</td>
<td>Forward Link (Pointer)</td>
</tr>
<tr>
<td>FS</td>
<td>Free Link Element List (Stack)</td>
</tr>
<tr>
<td>FSP</td>
<td>Free Link Element List (Stack) Pointer</td>
</tr>
<tr>
<td>Links</td>
<td>See B.L., F.L.</td>
</tr>
<tr>
<td>List</td>
<td>A data structure which defines a positional relationship between data elements in a group. This relationship can be implied by placing the elements in a contiguous area of memory or can be made explicitly by the use of Links.</td>
</tr>
<tr>
<td>MCU</td>
<td>Intel Microprogram Unit</td>
</tr>
<tr>
<td>NS</td>
<td>Neutral String (Stack)</td>
</tr>
<tr>
<td>NSP</td>
<td>Neutral String (Stack) Pointer</td>
</tr>
</tbody>
</table>
**Glossary**

(Cont.)

**PUSHAS**  Write $\emptyset$ in AS

**PUSHFS**  Write $\emptyset$ in FS

**PUSHNS**  Write $\emptyset$ in NS

**POPAS**   Read $\emptyset$ in AS

**POPFS**   Read $\emptyset$ in FS

**POPNS**   Read $\emptyset$ in NS

**RAM**     Random Access Memory

**SCU**     P3000 System Control Unit

**STACK**   A last in First Out (LIFO) Storage Structure

**STRING**  A series of characters stored in a contiguous area of memory.
LIST OF REFERENCES


