A MICROPROCESSOR-BASED RESIDENT MONITOR AND
TEXT-EDITING WORD PROCESSOR

by

Ralph Huxsol McCartney

A Thesis Submitted to the Faculty of the
DEPARTMENT OF ELECTRICAL ENGINEERING
In Partial Fulfillment of the Requirements
For the Degree of
MASTER OF SCIENCE
In the Graduate College
THE UNIVERSITY OF ARIZONA

1978
STATEMENT BY AUTHOR

This thesis has been submitted in partial fulfillment of requirements for an advanced degree at The University of Arizona and is deposited in the University Library to be made available to borrowers under rules of the Library.

Brief quotations from this thesis are allowable without special permission, provided that accurate acknowledgment of source is made. Requests for permission for extended quotation from or reproduction of this manuscript in whole or in part may be granted by the head of the major department or the Dean of the Graduate College when in his judgment the proposed use of the material is in the interests of scholarship. In all other instances, however, permission must be obtained from the author.

SIGNED: Ralph Harrod

APPROVAL BY THESIS DIRECTOR

This thesis has been approved on the date shown below:

T. L. WILLIAMS

Date

Professor of Electrical Engineering
TABLE OF CONTENTS

LIST OF ILLUSTRATIONS ........................................ v
LIST OF TABLES ............................................... viii
ABSTRACT .................................................. ix

CHAPTER

1. INTRODUCTION .............................................. 1
2. HARDWARE ................................................. 6
3. SOFTWARE ................................................. 34

3.1 Overview .............................................. 35
3.2 Functional Symbols ...................................... 38
3.3 Macro Definitions ........................................ 41
3.4 Master Utilities .......................................... 42
3.4.1 PUSH .................................................. 43
3.5 Monitor Routine .......................................... 56
3.5.1 Initialization ......................................... 58
3.5.2 Decoding ............................................... 60
3.6 Major Disc Subroutines ................................... 66
3.6.1 DISKC ................................................ 71
3.6.2 TRK .................................................... 71
3.6.3 SEEK .................................................. 73
3.6.4 CMMD .................................................. 73
3.6.5 TSTST ................................................ 77
3.6.6 CLDSK ............................................... 83
3.6.7 SRCHT ............................................... 85
3.6.8 READ .................................................. 87
3.6.9 WRITE ............................................... 89
3.6.10 FORMT .............................................. 89
3.6.11 DC .................................................... 92
3.6.12 LOAD ................................................ 95
3.6.13 STORE .............................................. 96
3.6.14 KFILE .............................................. 98
3.6.15 RNAME .............................................. 98
3.6.16 GTNAM ............................................... 101
3.6.17 ABSL ............................................... 105
3.6.18 SCTR ............................................... 106
3.7 Editor Package ........................................... 106
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.8 Cursor Utility Subroutines</td>
<td></td>
</tr>
<tr>
<td>3.8.1 CSRU</td>
<td>113</td>
</tr>
<tr>
<td>3.8.2 CSRD</td>
<td>113</td>
</tr>
<tr>
<td>3.8.3 CSRL</td>
<td>115</td>
</tr>
<tr>
<td>3.8.4 CSRL5</td>
<td>118</td>
</tr>
<tr>
<td>3.8.5 CSRR</td>
<td>118</td>
</tr>
<tr>
<td>3.8.6 CSRR5</td>
<td>120</td>
</tr>
<tr>
<td>3.8.7 DELETE</td>
<td>123</td>
</tr>
<tr>
<td>3.8.8 RSTSC</td>
<td>125</td>
</tr>
<tr>
<td>3.8.9 INSRT</td>
<td>129</td>
</tr>
<tr>
<td>3.8.10 EFILE</td>
<td>129</td>
</tr>
<tr>
<td>3.8.11 PRINT</td>
<td>132</td>
</tr>
<tr>
<td>4. MACROPROCESSING AND ASSEMBLY</td>
<td>135</td>
</tr>
<tr>
<td>4.1 Macroprocessor</td>
<td>135</td>
</tr>
<tr>
<td>4.2 Assembler</td>
<td>137</td>
</tr>
<tr>
<td>4.3 Operation</td>
<td>138</td>
</tr>
<tr>
<td>5. DOWNLOAD OPERATION</td>
<td>141</td>
</tr>
<tr>
<td>6. PROPOSED DESIGN</td>
<td>146</td>
</tr>
<tr>
<td>7. FUTURE APPLICATIONS AND EXPANSION</td>
<td>151</td>
</tr>
<tr>
<td>APPENDIX A. SOFTWARE SOURCE CODE LISTINGS: MOS,6502 MICROPROCESSOR ASSEMBLY LANGUAGE</td>
<td>157</td>
</tr>
<tr>
<td>APPENDIX B. OPERATING INSTRUCTIONS FOR THE PROTOTYPE RESIDENT MONITOR AND TEXT EDITOR</td>
<td>187</td>
</tr>
<tr>
<td>APPENDIX C. THE DOWNLOADING PROCESS: PDP-11 SOURCE CODE, LINKED SUBROUTINES, AND KIM-1 PROGRAMS</td>
<td>188</td>
</tr>
<tr>
<td>APPENDIX D. DISKETTE DATA FORMAT AND FILE STRUCTURE</td>
<td>197</td>
</tr>
<tr>
<td>APPENDIX E. EPROM CONTENTS LISTINGS</td>
<td>198</td>
</tr>
<tr>
<td>SELECTED BIBLIOGRAPHY</td>
<td>202</td>
</tr>
</tbody>
</table>
# LIST OF ILLUSTRATIONS

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>EPROM Delay Circuit</td>
<td>11</td>
</tr>
<tr>
<td>2.</td>
<td>EPROM Delay Timing</td>
<td>11</td>
</tr>
<tr>
<td>3.</td>
<td>TTY/KIM-1 Wiring Diagram</td>
<td>14</td>
</tr>
<tr>
<td>4.</td>
<td>CRT/KIM-1 Wiring Diagram</td>
<td>18</td>
</tr>
<tr>
<td>5.</td>
<td>Disc Controller Address Decoding</td>
<td>21</td>
</tr>
<tr>
<td>6.</td>
<td>Disc Controller/Drive Interfacing Circuit</td>
<td>27</td>
</tr>
<tr>
<td>7.</td>
<td>Host/Controller Interface Ports</td>
<td>28</td>
</tr>
<tr>
<td>8.</td>
<td>EPROM -9 Volt Supply</td>
<td>29</td>
</tr>
<tr>
<td>9.</td>
<td>Host/Controller Interface Board</td>
<td>30</td>
</tr>
<tr>
<td>10.</td>
<td>Disc Controller Socket Positions</td>
<td>31</td>
</tr>
<tr>
<td>11.</td>
<td>Two-Byte &quot;PUSH&quot; Flow Chart</td>
<td>44</td>
</tr>
<tr>
<td>12.</td>
<td>Two-Byte &quot;PULL&quot; and &quot;ERROR&quot; Flow Charts</td>
<td>46</td>
</tr>
<tr>
<td>13.</td>
<td>Two-Byte &quot;DOWN&quot; Flow Chart</td>
<td>47</td>
</tr>
<tr>
<td>14.</td>
<td>&quot;RCV96&quot; Serial Data Reception Flow Chart</td>
<td>49</td>
</tr>
<tr>
<td>15.</td>
<td>&quot;SMT11&quot; Serial Data Transmission Flow Chart</td>
<td>51</td>
</tr>
<tr>
<td>16.</td>
<td>&quot;SETSP&quot; and &quot;DELAY&quot; Service Routines</td>
<td>53</td>
</tr>
<tr>
<td>17.</td>
<td>&quot;XMT96&quot; Serial Data Transmission Flow Chart</td>
<td>54</td>
</tr>
<tr>
<td>18.</td>
<td>&quot;SAVSP&quot; Two-Byte Stack Pointer Save Operation</td>
<td>55</td>
</tr>
<tr>
<td>19.</td>
<td>&quot;RSTSP&quot; Restore Two-Byte Pointer Routine</td>
<td>57</td>
</tr>
<tr>
<td>20.</td>
<td>&quot;XMTLF&quot; Transmit a LF to CRT Flow Chart</td>
<td>57</td>
</tr>
<tr>
<td>21.</td>
<td>Initialization Flow Chart</td>
<td>59</td>
</tr>
<tr>
<td>22.</td>
<td>Monitor Routine</td>
<td>61</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>23.</td>
<td>Select Disc Choice &quot;DISKC&quot; Operation</td>
<td>72</td>
</tr>
<tr>
<td>24.</td>
<td>Decode Absolute Track Address Flow Chart</td>
<td>74</td>
</tr>
<tr>
<td>25.</td>
<td>Position Disc Head &quot;SEEK&quot; Subroutine</td>
<td>75</td>
</tr>
<tr>
<td>26.</td>
<td>Disc Controller Command Timing Diagram</td>
<td>76</td>
</tr>
<tr>
<td>27.</td>
<td>Host-to-Disc Controller Command Subroutine</td>
<td>78</td>
</tr>
<tr>
<td>28.</td>
<td>&quot;TSTST&quot; Test Disc Status Routine</td>
<td>81</td>
</tr>
<tr>
<td>29.</td>
<td>Controller Clearing Utility Routine &quot;CLDSK&quot;</td>
<td>84</td>
</tr>
<tr>
<td>30.</td>
<td>Search Disc Directory &quot;SRCHT&quot; Subroutine</td>
<td>86</td>
</tr>
<tr>
<td>31.</td>
<td>Disc Read Flow Chart</td>
<td>88</td>
</tr>
<tr>
<td>32.</td>
<td>Disc Write Flow Chart</td>
<td>90</td>
</tr>
<tr>
<td>33.</td>
<td>Disc Formatting Routine</td>
<td>93</td>
</tr>
<tr>
<td>34.</td>
<td>Display the Disc Directory Contents Routine</td>
<td>94</td>
</tr>
<tr>
<td>35.</td>
<td>Load a Disc File Flow Chart</td>
<td>97</td>
</tr>
<tr>
<td>36.</td>
<td>Store a Disc File Chart</td>
<td>99</td>
</tr>
<tr>
<td>37.</td>
<td>Delete Disc File Subroutine</td>
<td>100</td>
</tr>
<tr>
<td>38.</td>
<td>Rename Disc File &quot;RNAME&quot; Routine</td>
<td>102</td>
</tr>
<tr>
<td>39.</td>
<td>Fetch a Filename from CRT Flow Chart</td>
<td>104</td>
</tr>
<tr>
<td>40.</td>
<td>Absolute Disc Addressing</td>
<td>107</td>
</tr>
<tr>
<td>41.</td>
<td>Editor Instruction Recoding Flow Chart</td>
<td>110</td>
</tr>
<tr>
<td>42.</td>
<td>Cursor-Up Command Flow Chart</td>
<td>114</td>
</tr>
<tr>
<td>43.</td>
<td>Cursor-Down Command Routine</td>
<td>116</td>
</tr>
<tr>
<td>44.</td>
<td>Cursor-Left Command Routine</td>
<td>117</td>
</tr>
<tr>
<td>45.</td>
<td>Cursor-Left Five Times Logic Diagram</td>
<td>119</td>
</tr>
<tr>
<td>46.</td>
<td>Cursor-Right Command Routine</td>
<td>121</td>
</tr>
<tr>
<td>Figure</td>
<td>Description</td>
<td>Page</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>47.</td>
<td>Cursor-Right Five Times Logic Diagram</td>
<td>122</td>
</tr>
<tr>
<td>48.</td>
<td>Delete-Character Operation</td>
<td>124</td>
</tr>
<tr>
<td>49.</td>
<td>Reset the CRT Screen Operation</td>
<td>126</td>
</tr>
<tr>
<td>50.</td>
<td>Insert a Character Subroutine</td>
<td>130</td>
</tr>
<tr>
<td>51.</td>
<td>Erase File Contents Utility</td>
<td>131</td>
</tr>
<tr>
<td>52.</td>
<td>TTY Print Service Routine</td>
<td>134</td>
</tr>
<tr>
<td>53.</td>
<td>Download Timing and Pinout Connections</td>
<td>145</td>
</tr>
<tr>
<td>54.</td>
<td>Proposed Design Logic Schematic</td>
<td>147</td>
</tr>
<tr>
<td>55.</td>
<td>Proposed String Search Subroutine</td>
<td>152</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. KIM-1 Expansion Adapter Pinouts (PPS Bufferboard)</td>
<td>9</td>
</tr>
<tr>
<td>2. CRT User-Defined Options</td>
<td>16</td>
</tr>
<tr>
<td>3. SMS-305 Software Command Listings</td>
<td>19</td>
</tr>
<tr>
<td>4. Absolute Addressing for Peripheral Devices</td>
<td>22</td>
</tr>
<tr>
<td>5. CalComp 142M Disc Drive Characteristics</td>
<td>24</td>
</tr>
<tr>
<td>6. CalComp 142M Disc Drive User--Selectable Functions</td>
<td>25</td>
</tr>
<tr>
<td>7. CalComp 142M Disc Drive Signal Line Pinouts</td>
<td>26</td>
</tr>
<tr>
<td>8. Flexible Disc Controller Wire-Wrap Connections at Connector J106</td>
<td>33</td>
</tr>
<tr>
<td>9. Software Memory Map</td>
<td>39</td>
</tr>
<tr>
<td>10. KIM-1 Monitor Routine Commands</td>
<td>65</td>
</tr>
<tr>
<td>11. Disc Control Programs</td>
<td>68</td>
</tr>
<tr>
<td>12. Disc Track and Sector Organization</td>
<td>69</td>
</tr>
<tr>
<td>13. Diskette File Storage</td>
<td>70</td>
</tr>
<tr>
<td>14. SMS-100 Disc Controller STATUS Port: Bit Coding</td>
<td>79</td>
</tr>
<tr>
<td>15. MOS 6502 uP Assembly Code Addressing Mnemonics</td>
<td>139</td>
</tr>
<tr>
<td>16. Parts List and Projected Material Costs</td>
<td>150</td>
</tr>
</tbody>
</table>
ABSTRACT

The prototype of an office-oriented word processing system is presented. Based upon an eight bit microprocessor, hardware considerations include extended memory, high speed operator control, flexible disc storage media, and paper printing capability. Software was written in assembly-level language and contains a text editor and a resident monitor. The editor has scroll, deletion, and active cursor characteristics. The monitor interprets operator commands and includes a disc control package for storage and retrieval of data files. Macroprocessing, assembly, and downloading techniques are included.
CHAPTER 1

INTRODUCTION

This is an applications oriented thesis designed to fulfill requirements for electrical engineering course #410 which is part of the curriculum requirements for a Master of Science degree in Electrical Engineering as required by the Graduate College at The University of Arizona, Tucson. The goal of this thesis is to define, develop, and implement a modern, electronic office-oriented word processor. Modern word processors of this nature are used in office environments where large amounts of text-oriented communication are generated. By developing an electronic system whereby high speed, reliable text editing of typed or printed communications may be realized, production is greatly increased and resulting cost per transmitted word decreases. In addition, the ease of operation and high speed capability permit significantly improved reliability and flexibility of use.

To expedite the necessary requirements found in a successful text editing "intelligent typewriter," the project definition requires an intensive review of the necessary functions desired in the final system. To this end, the following criteria were determined at the beginning of this thesis program that would detail the necessary operations and lead to a developmental outline of the future hardware and software requirements. These generalized functions for a successful word processor are as follows:
1. Bulk data storage of text in file-oriented format.
2. Full text editing capability upon any given text file.
3. Ease of operation by untrained personnel.
4. Use of microprocessor based control logic to significantly reduce hardware cost.
5. Suitable hard copy output.

With the above criteria in mind, the design process now requires specific goals be defined to meet system requirements. Each of the five major points above is then broken down into subparts each pertaining to a different function or characteristic of the final desired configuration.

Mass storage requirements for bulk text involve a close examination of system trade-offs between such criteria as memory access time, cost per data unit, system interfacing difficulties, reliability, and system lifetime. The review of possible storage media was conducted and led to the selection of flexible disc storage. This review included charge coupled memory devices, magnetic bubble memory storage media, existing bipolar and $I^2L$ technology, flexible discs, and fixed and removable discs.

Text editing is the process whereby communication coded in the form of letters or other documents are generated, assembled, reviewed, and, if necessary, corrected. The desired functions for a successful text editor include ability to generate any standard typewritten symbol including all upper and lower case letters, all numerals, all
punctuation marks, and most required carriage-control characters such as line feed and carriage return.

The preceding criteria as defined by the project goals were implemented utilizing a number of components made available through the Department of Electrical Engineering at The University of Arizona. For printing output, it was found that the goals of this project could be met satisfactorily if the printer was supplied as an ASR-33 teletype. Use of this teletype permitted only the upper-case printing and minor restrictions upon punctuation and additional ASCII characters. The desire to use flexible diskettes as a bulk storage media led to the purchase of a pair of CalComp Model 142M flexible disc drives. Both drives were then to be controlled by a Scientific Microsystems Model FD 0305-2 Drive Controller Board. The implementation of the two drives and the controller board permits an extremely flexible, high speed data storage media whereby up to nearly one-half million bytes of data may be stored on-line for immediate retrieval. Off-line data storage is significantly greater due to the ability of changing diskettes between drives. The question of operator control was solved by the use of an available TEC cathode ray tube CRT and keyboard terminal. This unit permitted high speed serial interfacing to the designated logic control unit and offered not only a standard typewriter keyboard but, in addition, specialized key control functions. The logic unit required to control peripherals and to maintain control over the data files was delegated to a KIM-1 unit which had been previously purchased by The University of Arizona. This control unit, coupled with additional random access memory and programmable memory, provided a significant
test bed for sophisticated, extended program listings. The support functions for the cursor were placed in erasable/programmable read-only memory (EPROM).

Modern microprocessors have the unique features of generalized use, mass production (lowered costs), control versatility, and reliability. Since most commands and system software must be performed on a human time scale, the extremely high-speed functional requirements of modern main frame computers is not required. It was believed at the beginning of this project, and later after implementation it was shown to be true, that an 8-bit microprocessor operating at approximately \(2 \times 10^5\) instructions per second is entirely adequate on the human time scale to perform all necessary functions and system commands.

The final portion of the major criteria for project definition pertains to the final printed form of the generated text. If this project were intended to be incorporated into a design suitable for American industry, then the requirements imposed upon printing quality, speed, and mechanism reliability would be far more stringent than those called for in this thesis project. For purposes of this design, however, it was decided that a simple, "upper-case" only printing mechanism be employed to provide prototyping results for hard copy output of any given data file.

The major goals of this project thus are to create a microprocessor based word processing system complete with a resident monitor to control storage and command functions, a suitably powerful text editor to review and correct generated text data, to incorporate available hardware into an efficient prototype, to provide all necessary
commands for diskette storage of data files, and to provide sufficient documentation for future development should the need arise.
CHAPTER 2

HARDWARE

Hardware considerations for the microprocessor based word processor were dependent upon several considerations: available hardware from The University of Arizona, fiscal tradeoffs with available technology, system demands to be met, and future expansion considerat­ions. Between problem definition and completion date, a total of five months was available for project development and implementation. As a consequence, it was decided that equipment relegated to the relatively advanced state of the art development stage would be currently excluded from the project. The desire to obtain extremely high rates of file accession and deletion prompted the purchase of a pair of CalComp Model 142M diskette drives. This purchase was made in the light of an expected system expansion shortly after completion of this thesis project. The use of these drives in a storage application for the planned Microprocessor Development Laboratory permitted purchase of the equipment at this time. In other aspects, required equipment for the project could be obtained through equipment currently possessed by The University of Arizona.

The major subsections of hardware components may be broken down into the following nine categories:

1. KIM-1 central processing unit and memory module.

2. PPS buffer board for power buffering of expansion components.
3. RAM/EPROM memory board to expand the available system memory.
4. A 20 milliamp current loop for a teletype printer interface.
5. TEC cathode ray tube and keyboard.
7. Disc controller board.
8. Controller/KIM-1 interface.

These above major subsections of hardware were included for a specific user purpose or system function involving component communication or operation.

The KIM-1 microprocessor unit, manufactured by MOS Technology, Inc., of Mirristown, PA, was chosen as the heart of this word processor. Choice of this CPU was determined by (1) availability, (2) system compatibility with existing CPU equipment, (3) software support, and (4) the unit's versatility. With many ROM software packages currently installed in this central processing unit, the ability to utilize several of these subroutines for future expansion will be greatly simplified. In addition, several hardware options provided by the KIM-1 board facilitated implementation of several peripherals. As an example, the high current pin permitted the direct interfacing of the teletype printer to the processor unit.

In other aspects, the KIM-1 unit, which is based upon the MOS Technology 6502 microprocessor, permits a rather wide range of software functions through its various memory address modes. Use of these modes facilitates the relative flexibility of program control, and thus
provides user with a greater range of software-defined data stack techniques.

To effectively interface the KIM-1 unit to the outside world, a current source bufferboard must be provided to permit adequate signal buffering from the KIM-1 to an outside bus. This bufferboard, in addition to the KIM-1 unit, was again supplied by The University of Arizona as a previously constructed component. The bufferboard connects to the KIM-1 unit through two standard 44 pin connectors attached to the edge of the KIM-1 board. Through TTL buffering and suitable address decoding as noted in Williams (1976a), the bufferboard decodes address selection into eight 1024-byte increments and makes the selection of these blocks (K0-K8) on the general connection adapter which is used as bufferboard output to system peripherals. Table 1 lists the required interconnections and pinouts between the PPS/ bufferboard and the prototyping bus.

To effectively hold all user-defined software routines to be executed in this central processing unit, additional memory was required beyond the initial 1024 bytes made available in the KIM-1 units. This additional random access memory and erasable/programmable read-only memory was constructed on 0.10 inch on-center perfboard employing standard wire-wrap techniques. This memory extension module was provided to the author in a "rough" state of development, and that extensive circuit repair and analysis was performed prior to its implementation with the project. This random access memory and EPROM module was wired to occupy memory location /0400 through /0bff (2048 bytes of random access memory) and location /0c00 to location /1000
Table 1. KIM-1 Expansion Adapter Pinouts (PPS Bufferboard)

<table>
<thead>
<tr>
<th>Top Contacts</th>
<th>Bottom Contacts</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. +5v</td>
<td>A. +5v</td>
</tr>
<tr>
<td>2. --</td>
<td>B. --</td>
</tr>
<tr>
<td>3. GND</td>
<td>C. GND</td>
</tr>
<tr>
<td>4. CLK₂</td>
<td>D. CLK₂</td>
</tr>
<tr>
<td>5. CLK₁</td>
<td>E. CLK₁</td>
</tr>
<tr>
<td>6. AO</td>
<td>F. A1</td>
</tr>
<tr>
<td>7. A2</td>
<td>H. A3</td>
</tr>
<tr>
<td>8. A4</td>
<td>J. A5</td>
</tr>
<tr>
<td>9. A6</td>
<td>K. A7</td>
</tr>
<tr>
<td>10. A8</td>
<td>L. A9</td>
</tr>
<tr>
<td>11. A10</td>
<td>M. A11</td>
</tr>
<tr>
<td>12. A12</td>
<td>N. A13</td>
</tr>
<tr>
<td>13. A14</td>
<td>P. A15</td>
</tr>
<tr>
<td>14. K0</td>
<td>R. K1</td>
</tr>
<tr>
<td>15. K2</td>
<td>S. K3</td>
</tr>
<tr>
<td>17. K6</td>
<td>U. K7</td>
</tr>
<tr>
<td>18. R/W</td>
<td>V. RAM R/W</td>
</tr>
<tr>
<td>19. D0</td>
<td>W. D1</td>
</tr>
<tr>
<td>20. D2</td>
<td>X. D3</td>
</tr>
<tr>
<td>21. D4</td>
<td>Y. D5</td>
</tr>
<tr>
<td>22. D6</td>
<td>Z. D7</td>
</tr>
<tr>
<td>23. RDY</td>
<td>A. S0</td>
</tr>
<tr>
<td>24. NMI</td>
<td>B. IRQ</td>
</tr>
<tr>
<td>25. RST</td>
<td>C. spare</td>
</tr>
<tr>
<td>26. PA0</td>
<td>D. PA1</td>
</tr>
<tr>
<td>27. PA2</td>
<td>E. PA3</td>
</tr>
<tr>
<td>28. PA4</td>
<td>F. PA5</td>
</tr>
<tr>
<td>29. PA6</td>
<td>H. PA7</td>
</tr>
<tr>
<td>30. PB0</td>
<td>J. PB1</td>
</tr>
<tr>
<td>31. PB2</td>
<td>K. PB3</td>
</tr>
<tr>
<td>32. PB4</td>
<td>L. PB5</td>
</tr>
<tr>
<td>33. PB6</td>
<td>M. DECODE ENABLE</td>
</tr>
<tr>
<td>34. GND</td>
<td>N. GND</td>
</tr>
<tr>
<td>35. +12v</td>
<td>P. -12v</td>
</tr>
<tr>
<td>36. --</td>
<td>R. --</td>
</tr>
</tbody>
</table>
(1024 bytes of EPROM memory). A general schematic of the circuit memory may be found in Williams (1976d). The reference details the necessary address and data lines to and from all memory chips, and also includes the required control lines as decoded on board for the Enable and Chip Select functions. The 1K block provided by a total of four 2708 EPROM chips is controlled by the delay circuit in Figure 1. This design, due to the response time of 1.6 microseconds for the EPROM chips, includes a 1 microsecond delay through a standard J-K flip-flop. The operation and implementation of this delay is required due to the slow data access time of each EPROM chip. The operation of this circuitry is as follows: when the particular block of memory locations has been selected by the CPU (K3 goes low), the J-K flip-flop forces its output to cycle to the low state for 1 microsecond (1 clock cycle). Since the output of the flip-flop is directly connected to the input READY pin of the microprocessor, low state will indicate to the CPU that memory or peripherals must hold action temporarily and that the microprocessor logic must stop briefly. This permits a one microsecond addition to the data access time of each EPROM chip permitting the chip to cycle within 2 microseconds and supplying the resultant data on the output lines at the correct time. When the J-K flip-flop returns to its original high state, the microprocessor is allowed to proceed. The corresponding timing diagram is shown in Figure 2.

It was decided that extremely high quality printing would not be necessary. This is due in part to the requirements of a prototype system which dictates that only the bare necessities of system elements
Figure 1. EPROM Delay Circuit

Figure 2. EPROM Delay Timing
are required to prove actual operation. With this in view, an available ASR-33 teletype was used to represent printing output.

Most components applied to the prototype configurations were elements previously acquired by the Department of Electrical Engineering at The University of Arizona, and do not reflect available "state-of-the-art" equipment. Such a limitation resulted from funding constraints and the project's completion date. The purchase of new equipment would have presented a financial outlay far too great to justify the future use of a single-station automated text editor. Also, delivery of the necessary components in single-unit quantities would have added at least an additional four months to the development time.

The PPS bufferboard supplied to the author for memory extension contained several construction flaws, including an inactive address decoding chip, which were corrected prior to use of the additional memory. The bufferboard, a "two-times" reduction from artwork designed at The University of Arizona, often malfunctioned due to the accidentally small signal lines etched on the board. Adjustments and wire jumpers were installed to correct the errors.

Wiring mistakes with the address lines of the PPS memory board were also a source of delay. The random-access-memory (RAM) array consisted of 21LS02 memory chips in a 2048 by 8 bit arrangement. All signal and power lines were done using standard wire-wrap techniques. In all, some twenty-six wiring errors were located and corrected. No bipolar or LSI circuit chips were lost due to burnouts, however.

A final portion of the PPS expansion system, the analog input/output buffers, was removed to reduce power consumption and complexity.
The data format and timing considerations limit file or data printing to a rate of 10 characters per second. Connection of the teletype was made through the current loop pin outs made available by the KIM-1 unit in its original configuration. The wiring diagram is noted in Figure 3 and required two independent current loops: Printing loop and Reception loop. The output port of this TTY read/write circuit is at absolute address /1742 and requires 2 bits of that byte. It was decided that for system simplification, no use of the existing teletype routines provided by the KIM-1 unit would be made with this thesis. As a result, the actual KIM ROM routines were ignored during monitor use (see Section 3.5). Instead, a simple subroutine employing timing loops and the absolute memory location of this teletype was written to demonstrate serial data transmission and current loop interfacing to a TTL environment.

One major portion of this thesis project pertains to the subject of operator commands and interaction. This involves the ability of any given operator to conveniently communicate and employ the function of anticipated text-editing and monitor routines for successful use of this word processor. To further this end, two major design decisions were made.

The first decision involved the acceptance of the idea that no text-editing could be suitably performed on a paper printing medium. It would be preferred that the editing functions be performed on suitable cathode ray tube where large volumes of data may be quickly presented without use of large volumes of paper, and that changes in text may be quickly affected by simplified cursor commands.
Figure 3. TTY/KIM-1 Wiring Diagram
The second major design decision pertained to the purchase and future use of any possible equipment choice. Since it is anticipated that this thesis project will be expanded after completion, investment in a given piece of hardware at this time was considered appropriate. As a result, authorization was given for the application of a TEC Model 1400 CRT and keyboard for operator use. The major characteristics of this terminal are listed in Table 2 and denote the user options made available in the terminal design. For this particular installation, the user options chosen by the author are listed in the same table.

To facilitate extremely high speed screen update, it was decided that data transmission should be generated at as high a rate as made possible by the serial data electronics of both the central processing unit and the terminal itself. As a result, the terminal design selection of serial data rates ranging from 150 baud to 9,600 baud permitted the wide working range of data transmission rates. When it was determined that a software package could be written to provide data transmission in standard serial format consisting of one start bit, seven data bits, one parity bit and one stop bit, the terminal was switch-selected to accept the highest baud rate (9,600 baud) possible. This choice permits extremely fast screen update and high speed cursor corrections deemed necessary for comfortable operator use.

The actual transmission from the CPU unit (KIM-1) to both the teletype and CRT terminal was performed by generating serial data at TTL levels (via software commands). In addition, hardware requirements for additional MSI and LSI packages were minimized due to the extensive application of software to effect the necessary logic. As a result,
<table>
<thead>
<tr>
<th>Option</th>
<th>Characteristic(s)</th>
<th>Board and Switch Position</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baud Rate</td>
<td>9600 Baud</td>
<td>Rotary Switch on Back Panel</td>
</tr>
<tr>
<td>Duplex</td>
<td>Full Duplex</td>
<td>Toggle Switch at top of Back Panel</td>
</tr>
<tr>
<td>Signal Voltage</td>
<td>TTL levels</td>
<td>Board #5, Switch 2, Position 3</td>
</tr>
<tr>
<td>Word Length</td>
<td>10 bits</td>
<td>Board #5, Switch 1, Position 3</td>
</tr>
<tr>
<td>Start</td>
<td>1 bit</td>
<td>Automatic</td>
</tr>
<tr>
<td>Parity</td>
<td>Enable the Parity</td>
<td>Board #5, Switch 1, Position 3</td>
</tr>
<tr>
<td>Space Code</td>
<td>Disable space-code</td>
<td>Board #5, Switch 3 (turned left)</td>
</tr>
<tr>
<td>End-of-Line CR</td>
<td>Disable automatic CR</td>
<td>Board #5, Switch 4 (turned left)</td>
</tr>
<tr>
<td></td>
<td>char</td>
<td></td>
</tr>
<tr>
<td>Audible Signal</td>
<td>Enable signal at end</td>
<td>Board #3, Switch 1 (turned right)</td>
</tr>
<tr>
<td></td>
<td>of line</td>
<td></td>
</tr>
<tr>
<td>Roll-Up</td>
<td>Enable data roll-up</td>
<td>Board #3, Switch 2 (turned left)</td>
</tr>
<tr>
<td></td>
<td>on screen</td>
<td></td>
</tr>
<tr>
<td>60 Hz</td>
<td>Operate on a 60 Hz</td>
<td>Board #1, Switches 1 and 2 (turned right)</td>
</tr>
<tr>
<td></td>
<td>power source</td>
<td></td>
</tr>
</tbody>
</table>
very simple physical connections were required to connect both teletype and CRT terminal to the KIM-1 unit (see Figure 4).

The control of digital information onto and off of flexible disc drives was delegated to the Scientific Micro Systems Model FD 0305-2 disc controller board. The functions of this board are to receive data from the host unit (KIM-1), interpret the necessary commands generated by the host, and either read, write, or format information onto or off of a designated flexible disc drive. The SMS0305 board effects most of its communication with the host system through a total of eighteen signal lines and their respective grounds. These signal lines are divided into three groups: eight lines (1 byte) designated "data port," eight lines (1 byte) designated "status port," and two lines designated as "control." By the use of the two control lines CMT and ACK, the host is able to dictate the necessary controlling commands to the microcontroller board. Upon reception of the indication for a command (CMD goes high), the controller accepts the next two data bytes at the data port from the host as control bytes. These control bytes comprised a vocabulary of commands that may be generated by a host system to effect a given controller/disc operation. The resulting vocabulary of commands selected for use by the microcontroller board are listed in Table 3, and the corresponding functions are discussed next to each respective command (see Floppy Disc Drive Microcontroller: OEM Manual, 1976).

The general goal of this thesis has been to implement the controller by the use of the 128-byte buffer memory located on the controller board itself. Since data recovery rates or data write rates onto the diskette are designed and clocked at a rate of 1 bit per 4 microseconds, the
Figure 4. CRT/KIM-1 Wiring Diagram
### Table 3. SMS-305 Software Command Listings

<table>
<thead>
<tr>
<th>Command</th>
<th>Hexadecimal Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RESET</td>
<td>/00</td>
<td>Resets the FDC and all drives.</td>
</tr>
<tr>
<td>SEEK</td>
<td>/01</td>
<td>Moves the read/write head of a chosen drive to a desired track.</td>
</tr>
<tr>
<td>READD</td>
<td>/42</td>
<td>Reads a desired sector from disc to FDC buffer memory.</td>
</tr>
<tr>
<td>READB</td>
<td>/82</td>
<td>Reads the FDC buffer memory into the KIM-1 unit.</td>
</tr>
<tr>
<td>WRITD</td>
<td>/44</td>
<td>Writes the FDC buffer memory into a chosen disc sector.</td>
</tr>
<tr>
<td>WRITB</td>
<td>/84</td>
<td>Writes 128 bytes from the KIM-1 into the FDC buffer memory.</td>
</tr>
<tr>
<td>FMAT</td>
<td>/06</td>
<td>Formats an entire track according to IBM specifications.</td>
</tr>
<tr>
<td>STTS</td>
<td>/07</td>
<td>Obtains a single byte status of the desired disc drive.</td>
</tr>
</tbody>
</table>

Note: The desired disc drive is always inserted into bits #03 and #04 of the above hexadecimal codes (where MSB = Bit #00). Example for a READD command on disc drive #02, the resulting code is /52 or 01010010 in binary.
The 6502 microprocessor is incapable of supporting this speed and thus cannot be used to govern data read and data write functions for the disk storage requirements. If the controller were to simply accumulate one byte (of eight bits), each new data byte would be available or required by the controller every 32 microseconds. Since time constraints are extremely narrow between a possible data "supply" routine for the KIM-1 unit and the disc controller board, it was decided to implement the 128-byte buffer memory and execute data transmission onto and off of the diskette through this buffer. The general operation permits the following: for data read function, a disc to buffer command is executed and then, on a more leisurely time scale, a buffer to host transfer is performed. This permits the KIM-1 to be freed of any timing constraints imposed by the microcontroller and obtain organize, and if necessary decode data from the disc without being constrained by high speed operation limitations. The repaired write functions are performed in the same general manner with a host to buffer data transmission executed first, and a buffer to disc data transmission command executed secondly.

The addressing difficulties between the disc controller board and the host CPU were resolved with the circuit illustrated in Figure 5. The required eighteen control lines required by the controller board were tied to chosen pins of available I/O pins found on the KIM-1 board. Software was then dependent upon chosen addressing and was written to accommodate the required data direction changes found on the data port (see Table 4). The final major portion of this hardware discussion concerns itself with the bulk storage media chosen as the solution to be
Figure 5. Disc Controller Address Decoding
Table 4. Absolute Addressing for Peripheral Devices

<table>
<thead>
<tr>
<th>Device</th>
<th>Absolute Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>TTY Printer</td>
<td>/1742</td>
</tr>
<tr>
<td>SMS CONTROLLER</td>
<td></td>
</tr>
<tr>
<td>CONTROL</td>
<td>/1008 Bits #0 and #1</td>
</tr>
<tr>
<td>STATUS</td>
<td>/1010</td>
</tr>
<tr>
<td>DATA</td>
<td>/1700</td>
</tr>
<tr>
<td>Direction Register (for above port)</td>
<td>/1701</td>
</tr>
<tr>
<td>CRT</td>
<td></td>
</tr>
<tr>
<td>INPUT</td>
<td>/1702 Bit #0</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>/1702 Bit #7</td>
</tr>
<tr>
<td>Direction Register (for above ports)</td>
<td>/1703</td>
</tr>
<tr>
<td>PDP-11 DOWNLOADING</td>
<td></td>
</tr>
<tr>
<td>DATA</td>
<td>/1702 Bit #1</td>
</tr>
<tr>
<td>DATA STROBE</td>
<td>/1702 Bit #2</td>
</tr>
<tr>
<td>END-OF-WORD</td>
<td>/1702 Bit #3</td>
</tr>
<tr>
<td>EXPANDED RAM</td>
<td>/0400 to/OBFF</td>
</tr>
<tr>
<td>EPROM</td>
<td>/0C00 to/OPFF</td>
</tr>
</tbody>
</table>
implemented for this project. A pair of CalComp Model 142M flexible disc drives were purchased by the Department of Electrical Engineering at the University of Arizona for this project, and for later use as a developmental storage media, to provide large scale file storage of generated data. The characteristics of each disc are outlined in Table 5 and denote the common characteristics found in most IBM-compatible flexible disc formats. In addition, the model 142M provides several user selectable options. These options pertain to the operation of the individual drive and are listed in Table 6. The Model 142M flexible disc drive connects directly to the SMS micro controller board by means of a 40-pin, 0.10 inch O/C interface plug. This 40-pin plug contains a total of 12 signal lines and their respective grounds. These signal lines are designated as either status or function lines and are listed in Table 7. The interconnection required between disc drive and controller board involves the circuit of Figure 6. Since either the status or function lines of the interface between controller board and drive are open collector circuits a required "pull-up" termination circuit is required at each end. This termination circuit provides the termination voltage levels for successful interfacing to TTL-compatible data inputs.

Additional circuit changes (provided by the manufacturer) were required for correct interfacing between host CPU and the Flexible Disc Controller (FDC). These are shown in Figure 7 for the three 1/10 ports STATUS, DATA, and CONTROL of the FDC. Odd-numbered pins of these connectors are ground lines.
Table 5. CalComp 142M Disc Drive Characteristics

<table>
<thead>
<tr>
<th>Characteristic</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Storage Capacity</td>
<td>292,944 bytes per disc</td>
</tr>
<tr>
<td></td>
<td>3,328 bytes per track</td>
</tr>
<tr>
<td></td>
<td>128 bytes per sector</td>
</tr>
<tr>
<td>Data Transfer Rate</td>
<td>250,000 bits per second</td>
</tr>
<tr>
<td>Access Time</td>
<td>6 milliseconds (track-to-track)</td>
</tr>
<tr>
<td>Head Stabilization Time</td>
<td>10 milliseconds</td>
</tr>
<tr>
<td>Disc Rotation Speed</td>
<td>360 RPM + 2.5%</td>
</tr>
<tr>
<td>Latency</td>
<td>166.7 milliseconds maximum</td>
</tr>
<tr>
<td></td>
<td>83.4 milliseconds average</td>
</tr>
<tr>
<td>Disc Media</td>
<td>IBM 3740-compatible</td>
</tr>
<tr>
<td>Index</td>
<td>1</td>
</tr>
<tr>
<td>Sectors</td>
<td>26 (soft-sectored)</td>
</tr>
<tr>
<td>Tracks</td>
<td>77</td>
</tr>
<tr>
<td>Bit Density</td>
<td>6,536 BPI (inside track)</td>
</tr>
<tr>
<td>Recording Method</td>
<td>Frequency Modulation</td>
</tr>
</tbody>
</table>

aIBM 3740 Standard Format.
Table 6. CalComp 142M Disc Drive User—Selectable Functions

<table>
<thead>
<tr>
<th>Switch Code</th>
<th>Switch Position</th>
<th>Corresponding Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>A and B</td>
<td>A = Closed</td>
<td>Separated Clock at J1-38 and separated data at J1-36.</td>
</tr>
<tr>
<td></td>
<td>B = Open</td>
<td></td>
</tr>
<tr>
<td></td>
<td>A = Open</td>
<td>Raw data at J1-38.</td>
</tr>
<tr>
<td></td>
<td>B = Closed</td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>C = Open</td>
<td>WRITE ENABLED is dependent upon drive select.</td>
</tr>
<tr>
<td></td>
<td>C = Closed</td>
<td>WRITE ENABLED is independent upon drive select.</td>
</tr>
<tr>
<td>D</td>
<td>D = Open</td>
<td>Soft-sectored data format.</td>
</tr>
<tr>
<td></td>
<td>D = Closed</td>
<td>Hard-sectored data format.</td>
</tr>
<tr>
<td>E</td>
<td>E = Open</td>
<td>Power to stepper motor is enabled when SELECT is active (low).</td>
</tr>
<tr>
<td>F</td>
<td>F = Open</td>
<td>READY is independent of SELECT.</td>
</tr>
<tr>
<td></td>
<td>F = Closed</td>
<td>READY is dependent upon SELECT.</td>
</tr>
</tbody>
</table>

Note: Switches listed above are arranged as SPST rocker switches in a Dual-In-Line (DIP) package located on the bottom of the drive's interface electronics card.
Table 7. CalComp 142M Disc Drive Signal Line Pinouts

<table>
<thead>
<tr>
<th>Signal Line Function</th>
<th>Signal Pin—Ground Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Above Track 43</td>
<td>23-24</td>
</tr>
<tr>
<td>Head Load</td>
<td>19-20</td>
</tr>
<tr>
<td>Index</td>
<td>25-26</td>
</tr>
<tr>
<td>Ready</td>
<td>21-22</td>
</tr>
<tr>
<td>Select</td>
<td>15-16</td>
</tr>
<tr>
<td>Direction</td>
<td>05-06</td>
</tr>
<tr>
<td>Step</td>
<td>07-08</td>
</tr>
<tr>
<td>Write Data</td>
<td>31-32</td>
</tr>
<tr>
<td>Track 00</td>
<td>17-18</td>
</tr>
<tr>
<td>Write Enabled</td>
<td>(Not Used)</td>
</tr>
<tr>
<td>Run Data</td>
<td>37-38</td>
</tr>
<tr>
<td>Write Enable</td>
<td>33-34</td>
</tr>
</tbody>
</table>

Note: This description relates pinouts to the 40-pin connection located at the rear of the 142M and is designated "J0 1."
Figure 6. Disc Controller/Drive Interfacing Circuit
Figure 7. Host/Controller Interface Ports

Odd-numbered pins above are grounded.
Unused signal pins (even numbers) are strapped to +5 volts.
Figure 8 shows the redesign of the -9 voltage regulator needed for operation of the PPS memory board EPROM's. The circuit supplies up to 50 milliamperes of current at -9 volts through the use of a Fairchild type μA79MG voltage regulator.

The circuitry for address decoding and data buffering to the disc controller was mounted as shown in Figure 9. The three controller ports, CONTROL, STATUS, and DATA were terminated in 20-pin connectors, while the PDP-11/KIM-1 download port required a 13-pin connector. The decoding and buffering packages were arranged as shown, with all identification pins (pin #1) positioned in the upper left corner.

The SMS Microcontroller socket positions are denoted in Figure 10. The controller-to-disc interfacing is accomplished by a 40-wire
Figure 9. Host/Controller Interface Board
Figure 10. Disc Controller Socket Positions
ribbon cable connected at socket J106. During the course of this project, wire-wrap connections had to be modified to accommodate the undocumented 142M disc drive. The changes, made between the pair of sockets adjacent to J106, are documented in Table 8.
Table 8. Flexible Disc Controller Wire-Wrap Connections at Connector J106 -- Key: Pin #1.

<table>
<thead>
<tr>
<th>Logic Connection (From)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>E05 E88</td>
<td>Direction</td>
</tr>
<tr>
<td>E07 E90</td>
<td>Step</td>
</tr>
<tr>
<td>E15 E78</td>
<td>Drive Select 1</td>
</tr>
<tr>
<td>E17 E68</td>
<td>Track 00</td>
</tr>
<tr>
<td>E21 E70</td>
<td>Ready</td>
</tr>
<tr>
<td>E23 E94</td>
<td>Low Current</td>
</tr>
<tr>
<td>E25 E66</td>
<td>Index</td>
</tr>
<tr>
<td>E29 E64</td>
<td>Write Protect</td>
</tr>
<tr>
<td>E31 E56</td>
<td>Write Data</td>
</tr>
<tr>
<td>E33 E96</td>
<td>Write Gate</td>
</tr>
<tr>
<td>E37 E58</td>
<td>Read Data</td>
</tr>
<tr>
<td>E39 E76</td>
<td>Drive Select 2</td>
</tr>
<tr>
<td>E45 (Not Used)</td>
<td>Drive Select 3</td>
</tr>
<tr>
<td>E47 (Not Used)</td>
<td>Drive Select 4</td>
</tr>
<tr>
<td>E06 E87</td>
<td>Ground Lines</td>
</tr>
<tr>
<td>E48 E73</td>
<td></td>
</tr>
</tbody>
</table>
CHAPTER 3

SOFTWARE

This section describes in detail the required software instructions written to implement the defined functions for successful completion of this application project. For ease of use, all software instructions were organized into five blocks of source code each individual block of which is devoted to a particular family of subfunctions. These five software blocks are as follows:

1. Master Utility Functions: A general file of system subroutines throughout the entire set of programs.
2. Monitor: A resident monitor routine that organizes system commands and executes requests made by the operator.
3. Editor: A general use text-editing routine designed to create and/or delete ASCII files.
4. Disc Access: A generalized collection of routines that operate under the "monitor" routine that enables the operator to store and retrieve data files.
5. Cursor Utilities: A generalized package of software subroutines that perform desired operator functions on a given data stack, and then duplicate said functions through manipulation of the output data seen on the CRT terminal.

Economy of memory has been stressed when organizing software code. Most instructions have been written to utilize subroutines as
often as possible to eliminate duplicated code. As a direct result, the above five software packages are highly subroutine-oriented with a total of 245 jumps to subroutine found in approximately 2,600 bytes of source code. This averages approximately one subroutine jump for every eleven bytes of machine code.

3.1 Overview

The "Master Utility Routines" include most generalized functions employed throughout the remaining blocks of source code. Thus, most generalized utility routines that are found common to the other portions of software were allocated to this initial block of instructions. This initial sequence of instructions includes not only the generalized routines used throughout the rest of the programming. In addition, it contains absolute addressing and definition of labels for the zero-page scratch pad variables, a listing that defines all labels assigned to absolute addresses, and the labels used as programming aids to implement the source code by use of mnemonics. Thus the master utility routine sets the stage for use of the zero-page as scratch pad memory, absolute addressing, establishing positions of object code blocks in memory, major label definitions, and the generalized routines.

The "Master Utility Routines" also contain all system macro definitions used throughout the rest of the software. These definitions are considered to be a programming aid designed by the author, and represent an attempt at higher level programming. An example of a macro definition is as follows:
This example of a macro definition shows the implementation as employed by the author's macroprocessor as noted in McCartney (1977b). The spacing involved with this format is of little consequence since the macroprocessor is character or string oriented rather than field oriented. Thus it is irrelevant where a strong character begins in a given line. The rules of the macroprocessor include one basic element: any definition must precede its macro call. Any macro call of a given definition where the definition follows the call cannot be processed. Thus to continue the example, if at some point in the source code the macro call:

```
RESET
```

were found, the four lines of code noted between the commands ".MACRO" and ",ENDM" will be inserted into the finalized output version of source code.

The "Monitor Routine" contains two major parts. The first includes initialization of all system parameters. Such items as the clearing of all counters, initialization of the stack pointer and stack, and the correct programming of bi-directional data ports are included. The second major portion of the monitor routine includes a combination polling and interpretation sequence whereby the input port from the operator's terminal is checked for transmission of a data byte. If received, the byte is placed in buffer and decoded according to tests
written in the immediately following sequence of instructions. In addition, the detailed explanation of monitor functions includes suggestions for future expansion concerning the use of monitor routines to correlate operations when running programs.

The third major package of software included in this thesis is the "Editor." This text-editing package of routines contains primarily a short section of test and decode operations whereby a given command from the operator's CRT terminal is registered in buffer by appropriate instructions and then decoded for correct operation. Once the correct operation has been determined, a corresponding jump-to-subroutine is executed. The Editor also includes the return command "control-C" that returns operator control from editor to the resident monitor.

The fourth major block of software instructions is titled "Disc Access." These routines are called from the main monitor program, and they provide all data communication and table operations with either of two flexible disc drives. Software expansion permits the added choice of as many disc drives as the future user would desire. However, hardware limitations of the floppy disc controller prevent more than four drives from being implemented in the same system. The major disc functions include such items as file storage and file retrieval from disc, deletion and renaming of any known file, and the ability to access any absolute address of a chosen disc without format constraint.

The final portion of these five software packages is titled "Cursor Control." This section of software provides all necessary control for execution of text-editing commands. The title derives itself from the cursor viewed by the operator at CRT terminal. This
package contains such items as ability to move cursor to the right or left one position on the CRT, the ability to rewrite all data on the CRT to a new format, and the ability to move through the data file forwards or backwards at a conveniently fast rate.

These five major packages of software instructions comprise several units of object code which have been broken down according to function and mapped in Table 9. The total required memory allocation for the functions mentioned above comprised slightly over 2,600 bytes of memory storage. This sequence of data blocks has been written to take advantage of the MOS 6502 microprocessor's ability at subroutine jumps and has attempted to streamline object code requirements by making the source code intensively oriented toward confining identical sections or functions into one subroutine. A more detailed explanation of each major package is now included.

3.2 Functional Symbols

A machine code assembler has been written to acknowledge absolute addresses and assign corresponding values to defined labels. The programmer's task of organizing machine instructions and data bytes is considerably reduced, and potential mistakes due to numeric operators may be avoided by insertion of mnemonic operators.

As a result, the following five major assembler commands had been written for use by the author. These commands and their functions are as follows:

1. .BYTE: Reserves a position in memory for data. This position may be assigned a label.
Table 9. Software Memory Map

<table>
<thead>
<tr>
<th>Absolute Address</th>
<th>Allocated Function(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>/0F00 to /0FE9</td>
<td>Editor Package</td>
</tr>
<tr>
<td>/0C00 to /0EDE</td>
<td>General Cursor Routines: CSRU, CSRD, CSRL, CSRL5, CSRR, CSRR5, DLETE, RSTSC, INSRT, EFILE, PRINT</td>
</tr>
<tr>
<td>/05AF to /0BC5</td>
<td>General Disc Routine: DISC, TRK, SEEK, COMMD, TSTST, CLDSK, SRCHT, READ, WRITE, FORMAT, DC, LOAD, STORE, KFILE, RNAME, GTNAME</td>
</tr>
<tr>
<td>/04B7 to /05AE</td>
<td>Monitor Package</td>
</tr>
<tr>
<td>/0481 to /04B6</td>
<td>Utilization Program</td>
</tr>
<tr>
<td>/0401 to /0480</td>
<td>Disc Buffer Region</td>
</tr>
<tr>
<td>/0200 to /0400</td>
<td>Data File of 512 bytes</td>
</tr>
<tr>
<td>/0177 to /01FF</td>
<td>Processor stack region</td>
</tr>
<tr>
<td>/0158 to /0176</td>
<td>Utility Routines: SAVSP, RSTSP, XMTLF</td>
</tr>
<tr>
<td>/0150 to /015A</td>
<td>Filename Buffer Location</td>
</tr>
<tr>
<td>/0100 to /014F</td>
<td>Utility Routines: SETSP, DELAY, XMT96</td>
</tr>
<tr>
<td>/00EF to /00FF</td>
<td>Reserved for KIM-1 ROM Routines</td>
</tr>
<tr>
<td>/0012 to /00EE</td>
<td>Utility Routines: PUSH, PULL, DOWN, XMT11</td>
</tr>
<tr>
<td>/000 to /0011</td>
<td>Zero Page scratchpad variables</td>
</tr>
</tbody>
</table>
2. `.EQU`: Assigns an absolute value to the given label.

3. `.ORG`: Establishes either a beginning or a new address value.

4. `.BLKB`: Reserves a section of bytes in memory for data or table use. This may or may not contain a label.

5. `.END`: Denotes the termination of a given source file.

In addition, several other assembly rules exist that assist the programmer in organizing source mode. The majority of these rules may be found in McCartney (1977a).

Thus, the "Master System Utilities" (MSU) data file begins with an introduction and the definition of the first memory location through the `.ORG` assembly command. The first major function executed by MSU is to reserve a total of 18 bytes on the zero-page of memory for scratch pad and variable use in later programming. All bytes are noted with a corresponding label, and all bytes are initialized to the value `/00`.

The next sequence of commands given to the assembler assigns values to labels denoting absolute addresses. All data and control ports are defined for the assembler and assigned mnemonic labels for future use by the author. The section immediately following this defines the starting and ending positions of the data stack and disc transfer buffer.

The final major portion of commands written for the assembler preparation are for `.EQU` commands establishing a table of labels providing mnemonic correlation between an ASCII character and its corresponding numeric value. As an example the list begins with:

```
LF:  .EQU   /OA    ;Line feed character code,
```
This line of source code denotes the characters "LF" as a mnemonic label that represents the hexadecimal value /OA. A total of 38 ASCII characters are assigned user labels for future ease of programming. Of these characters, two are non-standard and have been diverted from normal use by the author for specialized functions. The character "EOT" was assigned the numeric value /OE and has been designated the end of text character for stack operations. This data byte denotes where the end of the stack exists. The second character is "ERR." This is merely an error byte whereby the first bit of the word is set and the remaining bits are cleared. The "ERR" data byte is employed generally when an error of execution results from an operation. This, then, communicates an error condition to higher portions of program control.

3.3 Macro Definitions

The macro definition, as described earlier in Section 3.1, provides the programmer with the ability to insert generalized blocks of code at a given point of a source code file. The ability to substitute large blocks of code for simple commands is a feature incorporated into this project to augment the programmer's productivity in generating suitable and efficient software.

A detailed listing of all rules and functions that must be followed for correct implementation of the macroprocessor may be found in McCartney (1977b). The abilities of this macroprocessor include the substitution of defined code for a macro call, the ability to pass labels, and concatenation of variables between macro definition and source code. These major points provide the programmer with extremely
powerful tools of software manipulation and assist in streamlining the finalized code. A brief description of each of the macro definitions may be found in Appendix A. The table format is used in this thesis to provide additional space for more pertinent comments and permits a more organized data format to the reader. An example of implementation of a macro definition is as follows:

Given the following macro definition:

```
.MACRO SP-SET HIGH, LOW
LDAIM HIGH
LDYIM LOW
JSR SETSP
.ENDM
```

Were the definition called in the following fashion

```
STAIM LINE
.SP-SET SP-H, SP-L
LDYZP CNTR
```

Then the resulting processed output will appear as follows:

```
STAIM LINE
LDAIM SP-H
LDYIM SP-L
JSR SETSP
LDYZP CNTR
```

This demonstrates the use of a macro call, concatenation of variables, and correct insertion of code.

### 3.4 Master Utilities

The first section of software routines includes all generalized subroutines acquired in later portions of the software for this project. Most common utilities that are required between the remaining four major packages of code were located in this portion of the software package as a general convenience to the programmer. This package
contains a total of 10 utility subroutines employed throughout the remaining software. A detailed description now follows for each individual routine.

3.4.1 PUSH

To create a 2-byte stack pointer, it is necessary to rewrite source code to accommodate a 2-byte addition or subtraction utility and tests to determine if a given range of memory locations has been exceeded by the pointer. The 2-byte stack pointer located in "SP-L" and "SP-H" is used for data operations upon a given data stack (see Figure 11).

The ".PUSH" macro definition in effect creates the operation of the "PSH" mnemonic instruction provided with the MOS Technology 6502 Microprocessor. This 2-byte push instruction performs the following operations:

1. Performs a general test to determine if the pointer has exceeded bottom of the designated data file.
2. Fetches the data byte located in variable "CHAR."
3. Stores the data byte through deferred-Y addressing at the 2-byte absolute held in the stack pointer locations.
4. Performs a 2-byte increment on the stack pointer bytes.
5. Effects return from subroutine.

The next major macro definitions used with the 2-byte stack pointer operation is ".PULL." This instruction, similar to the "PUL" mnemonic instruction supplied by the MOS Technology Corporation, is a 2-byte pointer operation. Again, the stack pointer is found at locations "SP-L" and "SP-H." The general effect of this routine is to
Figure 11. Two-Byte "PUSH" Flow Chart
increment the stack pointer by one location and then place the contents of the memory location at which the pointer indicates into the variable titled "CHAR". A flow chart for this routine is given in Figure 12.

The software performs the following functions:

1. Performs a test to determine if the top of the stack has been reached by the stack pointer.
2. Performs a 2-byte increment on the stack pointer.
3. Fetches the data byte from the absolute location indicated by the stack pointer and places it in the variable "CHAR".
4. Effects a return from subroutine.

The third major operation employing the 2-byte stack pointer is the "DOWN" macro definition (Figure 13). This definition provides the programmer with a departure from the standard mnemonic instructions provided in the 6502 microprocessor software. The instruction performs a data byte transfer from the given stack pointer location into the variable "CHAR". The macro definition then performs a 2-byte decrement upon the stack pointer. This may be viewed as a logical inversion of the "PUSH" macro definition. The following major points are performed by the software:

1. A test is performed to determine if the stack pointer has reached the bottom of the data stack.
2. Fetches the data byte pointed at by the stack pointer and places it in the variable "CHAR".
3. Performs a 2-byte decrement.
4. Effects a return from subroutine.
Figure 12. Two-Byte "PULL" and "ERROR" Flow Charts
IS THE SP POINTING AT THE BOTTOM OF THE DATA STACK?

NO

LOAD THE Y-REG WITH ZERO TO AVOID OFFSETS

MOVE THE CONTENTS OF THE MEMORY LOCATION POINTED AT BY THE SP INTO 'CHAR'.

PERFORM A 2-BYTE SUB ON THE SP BYTE.

RTS

Figure 13. Two-Byte "DOWN" Flow Chart
Note: In the previous three subroutine operations, if any single test to determine if the stack pointer is currently out of range, an error flag will be created in response to this condition. This error flag consists of placing the data byte "ERR" into variable location "CHAR". It will be the programmer's responsibility to detect this error flag in the subsequent higher program routines.

Data reception of ASCII characters from the operator's terminal is done by a routine designed to receive serial data at 9,600 baud. The choice of software control of serial data transmission and reception was chosen over additional hardware packages because in prototyping analysis it was decided the software package would be more expedient to develop than implementation of a standard UART and the associated support hardware.

The program that comprises the data reception at 9,600 baud is titled "RCV96". This routine accepts serial data which are timed at 104 microseconds per bit for reception by the KIM-1 unit. Figure 14 illustrates the flow chart and logic flow that was required to execute this routine. The data format required for successful reception of serial data is one start bit, 7 data bits, 1 parity bit (which is ignored) and one stop bit. Upon completion of this routine, the assembled 7-bit ASCII data byte is placed in memory location "CHAR".

For serial reception purposes, the least significant bit of the bidirectional data port "CRT" was dedicated to receive serial data from the CRT terminal. This bit location was designated a read-only bit by clearing the corresponding bit in the data direction register "DDR3B".
Figure 14. "RCV96" Serial Data Reception Flow Chart
The correct use of this subroutine by a higher level program requires that the accumulator and the Y-register be made available for use. In addition, the memory locations "CHAR", "CNTRL", "CRT", and "DDR3B" are used in this routine. For additional information on timing constraints or use of number of cycles for timing purposes, please refer to Appendix A for the source code listing.

The next major routine written to provide control of the TTY printer is titled "XMTll". This is a routine that generates a 110 baud transmission of data characters from the KIM-1 to the teletype printer. Data format consists of 1 start bit, 7 data bits, 1 parity bit (which is ignored), and 2 stop bits. With an effective rate of 110 baud, the resulting character transmission rate is exactly 10 characters per second. Figure 15 denotes the flow chart and logic diagram of this routine. Use is made of the high current TTL logic gate supplied by the KIM-1 unit to interface correctly with the loop-current connection required by the teletype. Software control creates a transmitted character by directly operating upon memory location "TTY" (absolute address /1742). In addition, the software requires the use of the accumulator, the memory location "BITS", and the use of the subroutine "DELAY". No use of either the X- or Y-registers was made during this routine. The subroutine called "DELAY" was required due to the long (nine millisecond) delay required for each data bit.

The "DELAY" subroutine provides the necessary delay between changes of data bits based on the output port "TTY" for serial teletype data communications. This routine requires the use of the accumulator
INITIALIZE THE DATA DIRECTION REGISTER, THE XMTTD BIT (0), PREP THE CONTENTS OF 'CHAR', & SET THE COUNTER 'BITS' TO COUNT 8 BITS.

PLACE THE NEW BYTE AT THE TTY OUTPUT PORT.

SHIFT THE 'CHAR' BYTE LEFT ONCE & DECREASE 'BITS'?

HAS 'BITS' COUNTED 8 XMTTD BITS?

DELAY FOR PARITY & 2 STOP BITS

Figure 15. "XMT11" Serial Data Transmission Flow Chart
and two counters located at "CNTR" and "CNTRL". The logic flow chart for this routine may be found in Figure 16.

To create a serial communication link from the KIM-1 to the user CRT, a routine titled "XMT96" was written to give transmission at 9,600 baud. This routine is preceded by an entry point at "XMT95" for special use by the macro definition ".XMT". The desired character to be transmitted to the CRT resides originally in location "CHAR" and then is clocked out at the rate of 1 data bit per 104 microseconds. The generalized format of this transmission consists of 1 start bit, 7 data bits, 1 parity bit (which is not checked) and 1 stop bit. The flow chart for this utility is presented in Figure 17.

The required data locations include use of "BITS" and "CHAR". These serial data bits are timed by use of the known cycle times of the software instructions and are transmitted through the most-significant-bit of the data port labeled "CRT" (absolute address/1702). Both the accumulator and the Y-register must be available for use. Contents of these registers are lost during program execution. The X-register is not affected.

Two short, simple routines written as programming aids for the 2-byte stack pointer operation now follow. The first is titled "SAVSP" and performs a 2-byte save operation of the stack pointer. The contents of each of the locations "SP-1" and "SP-H" are stored in the temporary stack pointer byte locations "TSP-L" and "TSP-H" (Figure 18). The second routine is a stack pointer reset operation, "RSTP" returns the stored values of the previous stack pointer location (held at "TSP-L" and "TSP-H") and places them in the operational stack pointer location.
Figure 16. "SETSP" and "DELAY" Service Routines
Figure 17. "XMT96" Serial Data Transmission Flow Chart
Figure 18. "SAVSP" Two-Byte Stack Pointer Save Operation
Only the accumulator and the respective storage locations mentioned above are utilized during either of the two routines. The X- and Y-registers are unaffected (Figure 19).

It was noted during operation of the CRT that toward the bottom of the screen, a delay was met in the transmission of fine feed characters. As a result, with the terminal occupied with "household duties," it was incapable of receiving all transmitted characters while occupied with screen update functions. Several characters were lost toward the bottom of the screen shortly after the transmission of a line-feed character. To overcome this difficulty, the subroutine "XMTLF" was written to provide sufficient buffer delay for the terminal. Thus, in future software programs, the programmer need only write ".XMTLF" as a macro definition. A line-feed character would then be sent to the CRT and a 30 millisecond delay performed. This additional 30 milliseconds provides the CRT with screen update time and permits any necessary "roll-up" required (Figure 20).

3.5 Monitor Routine

The monitor package as supplied by the author for this project includes two major parts. The first pertains to the system initialization and performs all necessary reset commands, data organization, and vector loadings. The second portion contains the reception and decoding of monitor commands coupled with the necessary subroutine jumps to execute the desired operation.
Figure 19. "RSTSP" Restore Two-Byte Pointer Routine

Figure 20. "XMTLF" Transmit a LF to CRT Flow Chart
3.5.1 Initialization

The initialization process performs the necessary system organization for a boot-up operation. The first operation is to establish the correct machine stack pointer within the KIM-1 unit. The stack pointer is initialized to the value \( \text{FF} \) and is used in software operations to coordinate subroutine data. The next operation performed by the initialization is to establish the high- and low-vector bytes of the non-maskable interrupt vector. This interrupt vector, which is located at absolute address \( \text{17FA} \) and \( \text{17FB} \), provides the operator with the ability to reset the monitor routine and reorganize operations from an undesired operating mode by depressing the "ST" key on the KIM-1 unit. This interrupt will force the 6502 microprocessor to take the two data bytes found at the vector interrupt memory locations and commence program operations at the new address location. In this case the new address location will be designated "INIT". The next operation of the initialization procedure pertains to the 2-byte stack pointer used for text-editing and data transmission programs. The stack pointer in question is initialized to point at the top of the designated stack memory area (see Figure 21).

The stack itself is initialized with the "EOT" character (end-of-text) at the top of the stack, thus creating an "empty" data stack as viewed by the software. The final major operation of the initialization program is to reset the flexible disc controller board and the corresponding disc drive. The read/write heads of each disc are re-positioned at track \( \text{00} \). This organizes all bulk storage media for immediate use and prepares the controller board for host commands from
Figure 21. Initialization Flow Chart
the KLM-1. In addition, the serial data bits used by serial data sub-
routines are initialized to either read or write status depending upon
their respective uses. The final operation after the initialization
procedures have been completed is to effect an absolute jump from the
given program code to the initial byte locations of the second portion
of the monitor routine, the decoding and jump-to-operation instructions.

3.5.2 Decoding

The second portion of the monitor software package contains a
decoding scheme whereby commands issued from the user CRT are interpreted
by software. Upon correct interpretation, program control is transferred
from the monitor to the corresponding software package to satisfy user
commands. Figure 22 denotes the resulting flow chart and logic diagram
employed by the author to effect most major monitor package operations.
The first portion of this flow chart includes the prompting given by
the KLM-1 unit to the operator. This consists of a simple period
transmitted to the CRT. The next portion involves the reception of a
command character from the CRT followed by the corresponding carriage
return which "authorizes" the execution of the command character.
Table 10 lists the control characters that are valid operation commands
for the monitor software. In addition to these commands their descrip-
tive code and a brief operational description is included. Upon
successful reception of a control character followed by a carriage
return, monitor operation proceeds at label "MON3" with the command
colorature currently residing in memory location "CHAR". A total of
eleven comparisons are made in an attempt to decode the character sent
Figure 22. Monitor Routine
Figure 22.—Continued Monitor Routine
'CHAR' = ASCII 'E'?  
YES

RELOCATE THE 2-BYTE SP AT THE TOP OF STACK, SET THE LINE COUNT TO ONE, CLEAR THE CRT & RESET THE CRT SCREEN CHARACTERS

JUMP TO 'EDIT'

'CHAR' = ASCII 'T'?  
YES

CLEAR THE CRT SCREEN & REPOSITION THE TTY TO THE LEFT MARGIN

RECEIVE A CHARACTER FROM THE CRT @ 9600 BAUD

'CHAR' = ASCII 'C'?  
YES

JUMP TO MONITOR

'CHAR' = ASCII 'C-UP'?  
YES

NO

ECHO 'CHAR' TO CRT & XMT IT TO TTY

Figure 22.--Continued Monitor Routine
Figure 22.--Continued Monitor Routine
Table 10. KIM-1 Monitor Routine Commands

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Enter the &quot;typewriter&quot; mode of operation.</td>
</tr>
<tr>
<td>D</td>
<td>Choose a disc for data storage uses.</td>
</tr>
<tr>
<td>L</td>
<td>Load a 1024-byte file from disc into the KIM.</td>
</tr>
<tr>
<td>S</td>
<td>Store a 1024-byte file onto a chosen disc.</td>
</tr>
<tr>
<td>R</td>
<td>Rename a desired file on disc.</td>
</tr>
<tr>
<td>K</td>
<td>Kill a file already stored on a common disc.</td>
</tr>
<tr>
<td>C</td>
<td>Display the disc contents on the TTY &amp; CRT.</td>
</tr>
<tr>
<td>F</td>
<td>Format a pre-selected disc.</td>
</tr>
<tr>
<td>A</td>
<td>Absolute read/write operation of a disc sector.</td>
</tr>
<tr>
<td>E</td>
<td>Enter the text editor.</td>
</tr>
<tr>
<td>P</td>
<td>Print contents of current stack on the TTY.</td>
</tr>
<tr>
<td>R</td>
<td>Run a program beginning at &quot;TOS&quot;.</td>
</tr>
</tbody>
</table>
from the CRT for possible execution. If the character is illegal and
cannot be matched with a desired monitor function, then after the
entire sequence of byte checks has been performed.

In most cases, the decoded character simply calls for a sub-
routine jump to a specific code or operation. In two cases, however,
decoding performs special functions for the programmer. The first
case is the transfer from monitor to text-editor routines. After
correct decoding of the "E" character, the following operations take
place:

1. The stack pointer is reset to the top of the stack.
2. The line count is initialized at line #1.
3. The CRT is completely cleared.
4. An absolute jump to the editor main routine is performed.

The second major operation that is nonstandard in the decoding
of monitor commands is the run command. The run command, which is
represented by the "R", simply causes the 6502 microprocessor to perform
an absolute jump to memory location "TOS". This location is the "Top-
Of-Stack," and in machine code operations contains the beginning address
of a machine code program. This run operation is a simplified version
of a more generalized technique, and further expansion could easily
include a more versatile approach to running machine code programs.
Future expansion of this monitor command is discussed in Chapter 7.

3.6 Major Disc Subroutines

The third major subroutine package contains the necessary
operations for complete control of all bulk data placed on or taken
off from the pair of flexible disc drives. These routines, as indicated in Table 11, perform all necessary searches, read and write functions, and host/controller interface functions for a simplified operator format.

It was decided that a simplified storage operation oriented toward a 1,024 byte data file would be the most appropriate form of bulk data storage for this project. As a consequence, the resulting disc file structure was greatly simplified. In addition, considerable efficiency in the use of data storage techniques resulted. This led to a total of 222 data files of 1,024 bytes each to be stored on one side of a single flexible disc.

The track and sector organization may be obtained from Table 12. This table denotes the tracks allocated to disc identification, disc content, and the actual data file. The table denoting the sector organization displays how each sector was organized according to active sectors in relation to desired data files. A total of 3 data files have been allocated for each track. Additional information concerning track and sector allocation may be obtained from the source code listing given in Appendix A. Table 13 denotes sector sequencing.

A total of 18 major software subroutines have been written to provide the adequate disc control necessary to offer all functions required for an intelligent word processor. We now begin a more detailed explanation of each of the 16 major disc routines, including a logic flow chart of each operation.
Table 11. Disc Control Programs

<table>
<thead>
<tr>
<th>Name</th>
<th>Memory Space</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISKC</td>
<td>48</td>
<td>Allows the operator to select a disc.</td>
</tr>
<tr>
<td>TRK</td>
<td>31</td>
<td>Computes the absolute track and sector from the absolute file number.</td>
</tr>
<tr>
<td>SCTR</td>
<td>16</td>
<td>Computes the next sequential sector.</td>
</tr>
<tr>
<td>SEEK</td>
<td>32</td>
<td>Causes the read/write disc head to position itself.</td>
</tr>
<tr>
<td>CMMD</td>
<td>100</td>
<td>Sends a two-byte command to the FDC.</td>
</tr>
<tr>
<td>TSTST</td>
<td>124</td>
<td>Tests the FDC status.</td>
</tr>
<tr>
<td>CLDSK</td>
<td>11</td>
<td>Clears the FDC after status is made available.</td>
</tr>
<tr>
<td>SRCHT</td>
<td>98</td>
<td>Searches a directory for a desired filename.</td>
</tr>
<tr>
<td>READ</td>
<td>62</td>
<td>Reads a 128-byte sector of data.</td>
</tr>
<tr>
<td>WRITE</td>
<td>57</td>
<td>Writes a 128-byte sector of data.</td>
</tr>
<tr>
<td>FORMT</td>
<td>120</td>
<td>Formats an entire diskette.</td>
</tr>
<tr>
<td>DC</td>
<td>165</td>
<td>Displays the contents of a diskette.</td>
</tr>
<tr>
<td>LOAD</td>
<td>89</td>
<td>Loads a 1024-byte file on diskette.</td>
</tr>
<tr>
<td>STORE</td>
<td>169</td>
<td>Loads a 1024-byte file on diskette.</td>
</tr>
<tr>
<td>KFILE</td>
<td>79</td>
<td>Kills a desired filename from the directory.</td>
</tr>
<tr>
<td>RNAME</td>
<td>75</td>
<td>Renames a desired file.</td>
</tr>
<tr>
<td>GTNAM</td>
<td>81</td>
<td>Utility that fetches a filename from the operator.</td>
</tr>
<tr>
<td>ABSL</td>
<td>168</td>
<td>Absolute sector read or write operation.</td>
</tr>
</tbody>
</table>
Table 12. Disc Track and Sector Organization

<table>
<thead>
<tr>
<th>Track #00</th>
<th>SECTOR #1: Diskette Identification/Special Messages.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>SECTORS #2-26: Reserved for future use (Bootstrap, Monitor, Editor, etc.)</td>
</tr>
<tr>
<td>Track #01</td>
<td>SECTOR #1: Reserved for Track Identification, etc.</td>
</tr>
<tr>
<td></td>
<td>SECTORS #2-25: Diskette Directory of the 192 file data slots.</td>
</tr>
<tr>
<td></td>
<td>SECTOR #26: Reserved for future expansion.</td>
</tr>
<tr>
<td>Track #02</td>
<td>SECTOR #1: Reserved for Track Identification, etc.</td>
</tr>
<tr>
<td></td>
<td>SECTORS #2, 8, 14, and 20: Diskette Directory of 30 data file slots.</td>
</tr>
<tr>
<td>Tracks #03 to 76</td>
<td>FILE STORAGE</td>
</tr>
</tbody>
</table>

The file format contains a fixed number of 1024 bytes per file and three files per track, occupying a total of 24 sectors on each track. The two empty sectors per track, #01 and #26, are reserved for future use.
Table 13. Diskette File Storage

<table>
<thead>
<tr>
<th>File Number</th>
<th>Sector Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 8, 14, 20, 3, 9, 15, 21</td>
</tr>
<tr>
<td>2</td>
<td>4, 10, 16, 22, 5, 11, 17, 23</td>
</tr>
<tr>
<td>3</td>
<td>6, 12, 18, 24, 7, 13, 19, 25</td>
</tr>
</tbody>
</table>

Reserved Sectors:

SECTOR #1: May contain Track Identification and special data.

SECTOR #26: Reserved for future expansion.
3.6.1 DISKC

This command routine is designed to permit the operator the direct choice of either of the two available flexible disc drives. After prompting the operator with "Disc #?", the operator may insert from the keyboard the choice of disc that is required for upcoming storage operations. Illegal characters transmitted from the CRT are decoded and ignored. A carriage return is not required to execute a legal character. Thus, in the prototype configuration, legal ASCII characters permitted by this routine to choose a disc number are "0" and "1" (see Figure 23).

After transmission of the legal character from the CRT to the KIM-1, the software routine, noting that the transmitted character is a legal disc choice, proceeds without additional prompting or carriage return to change the binary value located in memory location "DISC". This is done by subtracting the ASCII value /30 from the contents of location "CHAR" and then placing this result into location "DISC". A correct or legal disc choice is then echoed back to the CRT for operator conformation. A return from subroutine is then executed.

3.6.2 TRK

The organization of data files upon each floppy disc is dependent upon its file name location on the disc table. When a search routine locates the matching file name desired by the operator (see Section 3.7), the relative number or rank of the file (from 1 to 222) must be converted into a valid track address and the file rank found in that specific track. Thus, if the absolute file number of a
Figure 23. Select Disc Choice "DISKC" Operation
desired data file on floppy disc was 86, the absolute track address would become $28 + 2 = 30$, and the relative data file on that track would be #2. Note: the absolute track address is incremented by 2 because data files are recorded on the disc beginning at track #3. Thus the subrouting "TRK" divides the absolute file number by 3, determines the relative file number on that particular track, and then places both into available memory locations (see Figure 24).

3.6.3 SEEK

In order to correctly obtain information from a desired area on a flexible disc, a read/write head must be positioned radially to the correct circular track. This is done with the use of the subroutine "SEEK". This subroutine performs the desired track seek by inserting the desired command bytes into memory locations and then calling for a disc-command subroutine. The read/write head is then correctly positioned by the upcoming subroutine titled "CMMD". The "SEEK" subroutine then returns control to the higher routines without an error flag upon successful completion (see Figure 25).

3.6.4 CMMD

Commands sent to the flexible disc controller (FDC) are sent by software with the use of this subroutine. The command timing is found in Figure 26 and consists of the transmission of two data bytes through the data port at absolute address/1008. The general data transfer operation for each of two command bytes is as follows:

1. The desired command byte is latched into the output port at absolute location /1700 ("DSK2").
CLEAR THE VARIABLE 'TRACK' & LOAD THE ACC WITH 'CNTR'

ACC < 3?

NO

SUB 3 FROM ACC
INCREMENT 'TRACK'

MOVE ACC INTO 'SECTR'

INCREMENT 'SECTR'
THEN MULT IT BY 2

INCREMENT 'TRACK' BY 3

RTS

Figure 24. Decode Absolute Track Address Flow Chart
ENTER (SEEK)

MOVE THE SEEK COMMAND INTO THE VARIABLE 'CMD1'

MOVE THE DESIRED TRACK ADDRESS INTO 'CMD2'

USE THE SUBRIN 'CMD0' TO ISSUE THE NEW HOST COMMAND TO THE FDC

RTS

Figure 25. Position Disc Head "SEEK" Subroutine
Figure 26. Disc Controller Command Timing Diagram

ACK [unused]
2. The transfer line "CMD" is raised by the KIM-1.
3. The FDC accepts the command byte at the data port.
4. The FDC, when finished, raises the status line "XFR".

The above operation is conducted for two data bytes as required by the software instructions of the Scientific Micro Systems Micro Controller #FD0305-2. In general, the first data byte contains the desired command, disc choice, and user options. The second byte generally contains the desired track or sector number. In addition, should the "RESET" command be attempted, at the end of the second command by transfer, no status information is made available to the host. In all other instances, status information is available to the host after completion of the operation (see Figure 27).

3.6.5 TSTST

This utility contains two major portions designed to test and report, if in error mode, the status of the flexible disc controller board. Status information is available at the end of the transfer of the second command byte of a desired command. The only exception to this rule is the command "RESET" which provides no status information after the completion of the command.

Status is obtained by reading the input-only data port located at absolute address /1010 (label "DSK3"). Table 14 lists the 8 data bits available at the status port and their respective functions. If the error bit (bit #1) is set, then the data port at absolute location /1700 contains "Secondary Status." The secondary status may be obtained only on occasion of error or incorrect host control of the FDC,
Figure 27. Host-to-Disc Controller Command Subroutine
Table 14. SMS-100 Disc Controller STATUS Port: Bit Coding

<table>
<thead>
<tr>
<th>Bit</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (MSB)</td>
<td>DCTRL</td>
<td>Defines the direction of the data port.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = data output.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = data input.</td>
</tr>
<tr>
<td>1</td>
<td>FDC ON</td>
<td>Defines if power is applied to the FDC.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1 = FDC is off.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = FDC is on.</td>
</tr>
<tr>
<td>2</td>
<td>ERROR</td>
<td>When set, indicates an error was found during FDC operation.</td>
</tr>
<tr>
<td>3/4</td>
<td>STATID</td>
<td>Provides the type of status at data port:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00 = No status available.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>01 = Command Status Error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>10 = Disc Drive Error.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>11 = Operator Status.</td>
</tr>
<tr>
<td>5</td>
<td>DONE</td>
<td>1 = An operation is complete.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Operation in progress.</td>
</tr>
<tr>
<td>6</td>
<td>BUSY</td>
<td>1 = Controller is active.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>0 = Controller is inactive.</td>
</tr>
<tr>
<td>7 (LSB)</td>
<td>XFR</td>
<td>Handshake signal for data exchange.</td>
</tr>
</tbody>
</table>
The table also denotes the possibilities that may be encountered by decoding the bits #3 and #4 of the status byte and correlating this number with the corresponding command, disc, or operation status byte on the data port. In addition the corresponding operator code (viewed on the CRT1) is given.

The operation of "TSTST" is given in the logical flow diagram of Figure 28. The first major portion of this routine is to test the current controller status and continue with operation if the controller had made status available to the host at the data port. If status is not available, then the corresponding controller mode is placed in the memory location "MODE," and an immediate return from this routine is executed.

If the secondary status is available from the flexible disc controller, program execution continues at label "TSTS2". Here, the error bit of the status byte is tested. If set, an error has occurred during controller operations, and it is desired that the exact nature of the error be made known to the operator. Machine operation continues at label "TSTS3". If no error has occurred, the status is unimportant to the operator, the machine code conducts a "clear" operation on the controller, and a return from subroutine is executed.

If status indicates an error was performed during microcontroller operations, the exact nature of the error is obtained from the data port, converted to ASCII format, and stored in the memory location "NUM2". The type of error (command, disc, or operational) is found from the status port, converted into ASCII format, and stored in location "NUM1".
Figure 28. "TSTST" Test Disc Status Routine
Figure 28.—Continued
The final portion of the "TSTST" routine begins at mnemonic label "TSTS6". The short section prints an error message to the operator on the CRT. The general format consists of starting at the left of the CRT, printing "*** ERROR", the type of error (command, disc, or operational), a period, the secondary status, and three additional asterisks. In addition to this, due to the error mode, the 6502 stack pointer is returned to its original value of /FF and a direct return to the monitor routine is made. Thus, if during disc operation an error occurs, the operation is completely halted, the operator is prompted with an error message, and machine execution continues back at the main monitor routine.

3.6.6 CLDSK

This is a short service subroutine designed to clear the disc controller at the end of a command. With the exception of a "reset" command, all commands provide status concerning the operation at the end of their execution. Whether or not an error occurred during execution of the given command, the status must then be controller before future commands may be given. Thus, as seen in Figure 29, the logic is identical for both cases and involves merely the raising and lowering of the signal line "ACK". Once this has been accomplished, the FDC has assumed that the host has received status concerning the previous command operation, and thus resets itself to accept a new command.
Figure 29. Controller Clearing Utility Routine "CLDSK"
3.6.7 SRCHT

To obtain or locate a given data file on a disc, the following routine was written to search and test each individual file name for a match with the desired file name. As seen in Figure 30, the logic flow involves a matching routine designed to quickly check the disc file table and determine if an identical file name is present. If a successful match has been made, the X-register points as an offset of the first character of the desired file name in the disc buffer. The Y-register upon return will contain the value /00. The memory location "FCNTR" will contain the sequential file number of the given data file. That is, "FCNTR" will contain a number from 1 to 222 denoting the absolute sequential location of that data file on the disc. The memory location "TRACK" will continue the track number and the memory location "SECTR" will contain the sector number.

If, during execution, no identical file name was found that matches the desired file name, then an error will be returned to the major program routine through "FCNTR". The error code will be the value /El entered into "FCNTR". It is the programmer's responsibility to correctly test "FCNTR" in the main program software for a potential error or unmatched file name.

The logic flow of this routine simply consists of reading each sequential sector allocated to the disc table, attempting to match the first character of the desired file name with the first character of eight file names in the table. If no match is made with this first character, the X-register is decremented by /10, and the next file name in the table is tested. If no match in the entire sector (eight file
Figure 30. Search Disc Directory "SRCHT" Subroutine
A new sector is fetched upon successful matching of the first character of a file name, machine operation continues at "SRCH6". Here the X-register is stored in "BITS". The software then sequentially tests both the table "FILNM" and "BUFFER" one location at a time. If at any time a character does not match, the X-register is restored to its original value and the major search continues as before. If the tenth byte is correctly matched with the desired file name, a file name has been completely matched, the registers and memory locations containing return information to the main program have already been correctly set, the X-register is restored to its original value, and a return from subroutine is executed.

3.6.8 READ

This routine reads a 128-byte sector of information from a previously designated disc drive. This operation proceeds in two portions: a disc-to-buffer read and a buffer-to-host transfer. The first portion requires that the correct command byte and sector address (assuming a "SEEK" has been executed prior to this routine) be transferred as command bytes to the FDC. Once accomplished, the corresponding software routines are called to execute a data transfer from the disc to the FDC. The second portion concludes the operation by the transfer of data, one byte at a time, through the data port from the FDC to the host (KIM-1).

The logic flow diagram may be seen for this subroutine in Figure 31. Again, as in the subroutine "CMDM", acknowledgment of data bytes is performed by raising and then lowering the signal line "ACK".
PLACE THE 'READ' COMMAND FOR THE DISK TO BUFFER IN 'CMD1'. THEN MOVE THE SECTOR CHOICE INTO 'CMD2'.

USE SUBRNT 'CMMD' TO EXECUTE THE READ

LOAD THE 'BUFFER-TO-HOST' COMMAND INTO 'CMD1' & CLEAR 'CMD2'

USE SUBRNT 'CMMD' TO EXECUTE THE READ, THEN SET 'LINE'=80 TO COUNT 128 BYTES

WAIT UNTIL THE LSB OF FDC STATUS PORT IS HI

LOAD BYTE FROM DATA PORT & 'PUSH' IT ONTO STACK

RAISE CONTROL LINE 'ACK' TO FDC & WAIT UNTIL THE STATUS LINE 'XFR' DROPS

LOWER CONTROL LINE 'ACK'

DECREMENT 'LINE'

IS 'LINE'= 0? NO

YES

RTS

Figure 31. Disc Read Flow Chart
At any time during a given read operation, the subroutines and utilities listed in the software may send an error incurred by the hardware. The subroutine "TSTST" contains the necessary recovery software that will return operation to the monitor routine and prompt the operator with a period. There is therefore no software requirements to include additional error recovery with this set of instructions.

After completion and testing of status (immediately after the last data byte has been transferred) a return from subroutine is executed.

3.6.9 WRITE

The following write routine operates in very close to the same fashion as the previous "READ" subroutine. The difference is that information is written onto the flexible disc instead of being transferred from the disc to the host. This "WRITE" subroutine again performs its operations in two major sections: a byte-by-byte transfer of 128 bytes from the host (KIM-1) to the FDC storage buffer, and then a buffer-to-disc write command. The logic flow for this command may be found in Figure 32. Again, as in the "READ" subroutine, error control is handled by the utility subroutines, and additional instructions for error recovery is not required. Upon successful completion of this routine, a return from subroutine is performed.

3.6.10 FORMT

The following section of source code permits the programmer to initialize a flexible disc. After the selection of the desired disc drive through "DISKC", the keyboard command "Control-F" will transfer
PLACE A 'HOST-TO-BUFFER' WRITE COMMAND IN 'CMD1' & CLR 'CMD2'.

USE SUBRTN 'CMMD' TO EXECUTE THE WRITE.

SET 'LINE' = 80 TO COUNT 128 BYTES.

WAIT UNTIL THE FDC SIGNALS 'READY'.

USE RTN 'DOWN' TO FETCH THE NEXT DATA BYTE.

PLACE DATA BYTE ON DATA PORT OF FDC.

RAISE & LOWER THE CONTROL LINE 'ACK'.

USE SUBRTN 'TSTST TO CHECK FDC STATUS.

IS FDC OPERATION DONE?

YES

DECREMENT 'LINE'.

IS 'LINE' = 0?

YES

PLACE THE COMMAND 'BUFFER TO-DISK' WRITE IN 'CMD1' & THE DESIRED SECTOR IN 'CMD2'.

USE SUBRTN 'CMMD' TO EXECUTE THE WRITE.

RTS.

Figure 32. Disc Write Flow Chart
machine execution from the main monitor routine to this formatting sequence.

Several major operations are conducted during this sequence. The first operation includes the correct address formation of all 77 disc tracks. This requires that a correct "SEEK" operation be conducted prior to a "FORMAT" command. Once the "FORMAT" command has been given to the FDC, all 26 sectors of the current track are formatted according to IBM standards. For further information concerning disc formatting, please turn to Appendix D.

The next portion of the source code involves the transfer of null-characters onto Tracks 0 through 2. This initializes the disc identification sector and the tracks pertaining to the disc table. After final initialization of three tracks of nulls, the last operation in this formatting sequence is conducted. This is to write a data sector of "/FF" characters to flag the end of the disc table. With the first sector of Track 00 allocated to the disc identification, remaining sectors on tracks 0, 1, and 2 provide sufficient space for the disc table. The format for file names in this disc table is as follows:

1. Eight file names per sector of 128 bytes.
2. A total of 16 bytes per file name.
3. The first ten bytes (number 0 through number 9) allocated to ASCII characters representing file name.
4. The remaining six bytes (/0A through /0F) are allocated for future expansion.
Upon completion of the writing of the end-of-table sector, a "RESET" command is given to the FDC, and a return from subroutine is executed. The logical flow chart for this program may be seen in Figure 33. For further information regarding this program, please refer to the source code listed in Appendix A.

3.6.11 DC

This portion of the source code was written to provide the operator with the ability to retrieve from the disc a copy of the disc table. This provides the operator with a record of all current files and provides bookkeeping functions for file organization.

The logical flow chart for this program may be seen in Figure 34. The operations begin with a jump to this routine from the main monitor routine which had decoded the command "C". The operator is prompted with the request for printing operation, "ENABLE PTR?", and must then transmit to the CPU a "P" character to enable the teletype printer. Should any other character be sent, no printing output will occur at the teletype (only the CRT terminal will be enabled).

The next portion of this routine positions the disc head at the correct track and begins reading at sector 2. Each sector is sequentially read from the disc, and the resulting ASCII characters are entered into the disc buffer in the KIM-1. Once returned, software decoding begins by isolating the ten-character file name for each file. A total of eight file names will be found per sector. Once isolated, the file name is sent to the CRT, one character at a time. At the end
Figure 33. Disc Formatting Routine
Figure 34. Display the Disc Directory Contents Routine
of a file name, a line feed and carriage return are transmitted to reset
the CRT to the next line.

At the end of each sector, the operator is prompted with a pair
of spaces and a dash to indicate a pause has been reached. This pause
was inserted to provide the operator with a more leisurely rate of data
display. Typing any key on the operator's keyboard will resume opera­
tion. Upon completion of the transmission of all possible file names
from the disc table, the program executes a return from subroutine.

3.6.12 LOAD

This subroutine is written to provide the ability to retrieve
a data file consisting of 1,024-bytes from a previously selected disc
drive. After transfer from the main monitor program by the command
key "L", the "LOAD" routine begins by prompting the operator for a file
name by typing "FILENAME". After use of the "GTNAME" subroutine to
obtain the desired file name (see GTNAME source code description follow­
ing shortly), the use of the "SRCHT" subroutine is executed with an
attempt to find the desired file name in the disc table. If no file is
determined, an error is printed on the CRT and a return from the sub­
routine is executed.

Should the file be found, the use of "TRK" is made to determine
the absolute track address and the relative file number on that track.
Once done, the two-byte stack pointer is repositioned to the top of the
data stack, and a seek operation is used to correctly relocate the
read/write head of the desired disc.
In the correct sequential format, a total of eight sectors are read from disc into the KIM-1. Once completed, the stack pointer is reset to the top of the data stack, a reset operation replaces the disc head to track number 00, and a return from subroutine is executed. Further information concerning this operation may be found from the logical flow chart in Figure 35, or from the source code listing found in Appendix A.

3.6.13 STORE

This service routine is designed to permit the operator to store 1,024-byte data file on a previously selected disc drive. After transfer of machine execution from the main monitor routine, the operator is prompted with the request for a desired file name: "FILENAME". After using the subroutine "GTNAME", the software then attempts a search of the disc table to determine if the file name existed previously. If it existed, program control continues at label "STOR5". If not, the program continues with the next statement.

If the file name was not found, the next step in the program logic is to search the disc table for a vacant data file position. Once found, this position is decoded into an absolute track address and sector, and the vacant slot on the disc table is filled with the desired file name. The disc drive head is then positioned at the new absolute track address through a "SEEK" command, and the data file is stored in the correct sector format at that track.

If the file name was found during the table search, then the relative file position is translated into an absolute track address and
LOAD1

SEND PROMPTING MESSAGE 'FILENAME?' TO OPERATOR

USE .GETNAME TO FETCH FILENAME FROM CRT

SEARCH THE DISC DIRECTORY FOR A LIKE FILENAME

WAS THE FILE FOUND?

NO

PUT CODE '4.0' INTO ERROR BUFFERS

LOAD2

DETERMINE THE TRACK & SECTOR ADDRESS FROM THE ABSOLUTE FILE NUMBER

LOAD3

POSITION R/W DISC HEAD TO DESIRED TRACK

SCTCT = 8
SET THE 2-BYTE POINTER TO TOP OF STACK

READ A SECTOR ONTO THE STACK

SELECT THE NEXT SECTOR & DECREMENT SCTCT

LOAD4

NO

IS SCTCT = 0?

YES

RESET THE STACK POINTER TO TOP OF DATA STACK

LOAD5

RTS

Figure 35. Load a Disc File Flow Chart
sector number, a "SEEK" command repositions the disc read/write head, and the required eight sectors for the 1,024-byte data file are then written. Upon completion of this write operation, the 2-byte stack pointer is repositioned to the top of the data stack, and a return from subroutine is executed. The logical flow chart for this operation may be seen in Figure 36.

3.6.14 KFILE

To delete a certain file from the contents of a disc table, the following routine was written. After receiving the correct keyboard command from the CRT terminal, program execution continues at the program label "KFILE".

We begin by having the operator prompted with the request for desired file name, "FILENAME". Using the "GTNAME" routine to obtain the desired file name, a subroutine search of the disc table is performed to locate the desired file.

If the search does not locate the desired file name, an error is presented to the operator and a return is executed. If, however, the file name is located, null characters are written over the original file name location now in "BUFFR". Once accomplished, the contents of "BUFFR" is returned to the original sector position in the disc table. Upon completion of this operation, a return from subroutine is executed. The logical flow chart for this short routine may be found in Figure 37.

3.6.15 RNAME

This routine is written to permit the operator to rename any desired file. Machine execution transfers from the main monitor routine
Figure 36. Store a Disc File Chart
Figure 37. Delete Disc File Subroutine
to the software label "RNAME" upon reception of the control character "R". The routine begins by requesting the operator for the old file name by transmitting "OLD FN". Once the old file name is received, an attempt to search the disc table is made, and if an error is located, an error flag is set and a return from subroutine is executed.

If the old file name is located in the disc table, a new line is established on the CRT, and the software then requests the new, desired filename from the operator. The prompting for this is "NEW FN". Again using the "GTNAME" subroutine to obtain a desired filename, the software then transposes the new file name over the old currently located in the KIM-1 buffer. Once overwritten, the two-byte stack pointer is set to the top of the disc buffer, and a disc write is performed. This write operation replaces the old sector information with the newly revised filename inserted in the sector data. A return from subrouting is then executed. The logical flow chart for this operation may be found in Figure 38.

3.6.16 GTNAM

This is a utility routine designed to allow the operator to enter filenames from the CRT into a short, ten-character filename buffer. The routine consists of four small sections: initialization, reception and decoding, insertion, and deletion.

The first portion, the initialization, begins by loading the filename buffer, "FILNM", with blank characters. Once ready, the X-register is employed as an offset pointer, and the program continues with reception and decoding.
ENTER (RNAME)

PROMPT THE OPERATOR WITH 'OLD?'

USE SUBRTN 'GTNAM' TO OBTAIN ORIGINAL FILE NAME

USE SUBRTN 'SRCHT' TO SEARCH THE DISK CONTENTS FOR THE DESIRED FILE

'CNTR' = /DE?

YES

PROMPT THE OPERATOR WITH 'NEW?'

USE SUBRTN 'GTNAM' TO OBTAIN DESIRED FILENAME

SET Y-REG = 9

TRANSFER THE BYTE IN ('FILNM'+Y) TO THE LOCATION ('BUFFER'+X)

DECREMENT X-REG
DECREMENT Y-REG

Y=0?

NO

RESET THE 2-BYTE SP TO THE TOP OF THE DISK BUFFER

USE SUBRTN 'WRITE' TO PLACE THE DATA BACK ON THE DISK

RTS (RNAME2)

Figure 38. Rename Disc File "RNAME" Routine
The reception and coding portion of this routine begins by the reception at 9,600 baud of an ASCII character from the CRT. Once received, tests are performed for a deletion character or a carriage return. If a rubout character is decoded, machine execution continues at software label "GTNA4". If a carriage return is decoded, a return from the subroutine is executed. Therefore, no carriage return nor rubout character may ever be legally used in a filename.

If neither of the two characters were detected, program operation would continue with the insertion section at label "GTNA3". Here, a test for the bounds of the X-register is performed. If more than 10 characters had been received, all additional characters are ignored and program continues with the reception portion of this routine. If less than ten characters have been received, the character is placed into the filename buffer "FILNM", and the X-register is repositioned down the stack one place.

If a rubout character is decoded, program execution at label "GTNA4" begins by testing if the X-register pointer has exceeded the bottom of the filename buffer. If true, then there are no additional characters to delete, and program execution resumes at the reception routine. If instead there are characters available for deletion, then the X-register is repositioned one place higher in the filename buffer, the position is written over with a blank character, and the CRT screen is reset. The routine then transfers back to the reception operation. The flow chart for this subrouting may be found in Figure 39,
Figure 39. Fetch a Filename from CRT Flow Chart
3.6.17 ABSL

This routine permits the operator to access any absolute track and sector address on a chosen disc. Once the desired location is found, the choice of reading or writing an entire sector of 128 bytes is given to the operator. This allows direct modification of any byte stored under IBM Data-recording format (IBM System 3740).

The routine begins by prompting the operator with "SECTOR?", asking for the desired absolute sector number (from 1 through 26). The reply must consist of two decimal digits (i.e., 08). When the ASCII characters are received, they are transformed into a single binary-coded byte (label "ABSL4" and following) which is stored at CNTR + X-register. The X-register is then incremented. The operator is then prompted on a new-line with "TRACK" asking for the desired track number (from 00 to 77). Again, two decimal digits are required to satisfy the decoding process. These digits are converted from ASCII format to a single binary-coded byte which is stored in CNTR + X-register.

At "ABSL7", the prompting routine asks the operator if this is to be a read or write operation. Upon receipt of an ASCII "R", program control resumes at "ABSLA" where the track and sector values having been preset earlier, now permit a disc head SEEK operation and a READ operation of 128 bytes. The new data are inserted onto the data stack beginning at the last known cursor location while in edit mode. If the data exceeds the data stack, the overflow is lost. In addition, previous data are overwritten and thus lost.

Should a write operation be requested by the operator by typing the keyboard letter "W", a SEEK operation to the correct track is
initially performed. Then a WRITE operation transfers 128 bytes from the data stack onto the desired sector. The data begins at the last known cursor location while in edit mode and includes the next 128 bytes down the stack. Should the end of the stack be reached, the remaining characters transferred to disc are ERR bytes (/80). After completion of either the READ or WRITE portion of ABSL, a return from subroutine is performed (Figure 40).

3.6.18 SCTR

This is a short algorithm added to the source code file when it became apparent that sequential sector read/write operations consumed too much time. By staggering the sequence of sectors, the resulting delay contributes to the processing time of each 128-byte block. An entire revolution by the disc need not be performed. Thus, two revolutions instead of eight are necessary, reducing software latency time from 1.3 seconds to about 0.3 seconds.

This algorithm starts by incrementing the sector value by six. A test to see if the result is greater than twenty-six is performed. If not, the sector value is valid, and a return from subroutine is executed (label "SCTR1"). If "SECTR" exceeds twenty-six, the value is invalid, and the value twenty-three is subtracted bringing the sector address to the next cyclic sequence that starts on the track. A return from subroutine is then executed.

3.7 Editor Package

The editor package is a collection of software instructions designed to correctly receive and decode text editing commands from
Figure 40. Absolute Disc Addressing
Figure 40.—Continued
the operator via CRT terminal. This routine consists of a reception section and a total of ten decoding comparison operations. Upon successful decoding of a given command, the editor package will cause machine execution to continue at the desired text-editing utility routine. These routines are to be found in Section 3.8.

Only ASCII characters with a value 020 or higher may be permitted as a legal text character. Other characters lower than this value, or the "rub-out" character (07F), are permitted control status. If a character within this group has no selected function as determined by the software, it will be ignored and control will return to the reception instructions of the editor package.

Logic for this section of code may be found in Figure 41. For further information, the source listing may be found in Appendix A.

3.8 Cursor Utility Subroutines

This final package of major utility subroutines comprises the necessary operations imposed upon the CRT cursor and resulting logical operations performed on the data stack. The use of the two-byte stack pointer, serial data transmission routines, and the editor package containing the main editor decoding routines are employed by this subroutine package. The package contains a total of eleven major utilities and provides operations for the cursor, deletion of characters at the cursor location, the ability to retype the CRT screen, the ability to insert a character at a given cursor location, the ability to erase a data file from the cursor location onward, and the ability to print data file on paper at the teletype.
Figure 41. Editor Instruction Recoding Flow Chart
Figure 41.—Continued Editor Instruction Recoding Flow Chart
Figure 41.—Continued Editor Instruction Recoding Flow Chart
3.8.1 CSRU

This utility allows the cursor to move upwards one line. For purposes of this project, it was defined that should the cursor be located at the very top line of the CRT, then operation would include a complete rewriting of the CRT screen and positioning of the cursor at the top of the previous page. Thus, if the cursor were located on the uppermost line of the CRT and the cursor-up key were depressed by the operator, the previous page of up to 12 lines will be transmitted to this CRT and the cursor positioned at the upper leftmost position. If the cursor is not located at the uppermost line, then the command merely repositions the cursor at the leftmost position of the next higher line.

The logic begins, as shown in Figure 42, by testing if the cursor is at the uppermost line. If not, cursor routine continues at software label "CSRU4". At this point the line count is decremented, the cursor is repositioned one line up, the 2-byte stack pointer is reset to the new position, and a return from subroutine is executed.

If a new screen must be displayed (due to position of the cursor at the topmost line), the 2-byte stack pointer proceeds through memory backwards until up to a total of 12 lines have been counted. The CRT is then rewritten, the cursor is repositioned at the 2-byte stack pointers location, the line count is returned to the value of "1", and a return from subroutine is executed.

3.8.2 CSRD

This routine was written to perform the "cursor-down" command. When this command is given, the cursor is repositioned from its current
Figure 42. Cursor-Up Command Flow Chart
location to the leftmost screen position of the next lower line. If the cursor currently is at the bottom line of the screen, automatic rollup of the CRT is performed, and the new line is transmitted from the data stack to the CRT terminal.

The line count of the cursor remains the same ("LINE" equals \(0C\)). Should the character "EDT" be encountered, this signals the end of text and subsequent lack of data beyond it. Operation of this command terminates and any attempt to proceed with a cursor-down command will be ignored (Figure 43).

3.8.3 CSRL

This routine is designed to effect the command to place the cursor left one position on the CRT. It was decided that for the use of this project, if the cursor were located at the leftmost screen column, then an additional cursor-left command will reposition the cursor one line higher at the right end. If the cursor were, however, located at the topmost left position, for simplicity of software it was decided that no additional operation would be executed. Therefore, from this position, additional cursor-left commands would be ignored.

The logical flow chart for this subroutine may be found in Figure 44. Machine control continues after correct decoding of the cursor-left command at software label "CSRL". The tests for exceeding the data stack and availability of the next higher line are made. If the stack is exceeded, the command is ignored and a return from subroutine is executed. If the cursor encounters a carriage return immediately preceding the current location, the software attempts to
Figure 43. Cursor-Down Command Routine
Figure 44. Cursor-Left Command Routine
reposition the cursor on the next higher line at the far right. Otherwise the machine control continues at label "CSRL6" where a cursor-left ASCII control character transmitted to this CRT terminal, and a return from subroutine is executed.

If an attempt is made to reposition the cursor on the next higher line, a test is then made to first determine if the cursor is located at the top of the CRT screen. If this is true, the cursor command is ignored. If not, the stack printer is used to count through the next higher line, determine the length, locate the end of the line, cycle the cursor through to the correct position on the next higher line of the CRT, match the stack pointer with the new cursor location, and then return from the subroutine.

3.8.4 CSRL5

This is a short subroutine triggered by the keyboard command "control-L". When decoded, this routine effectively calls the subroutine "CSRL" a total of five times. This causes the use of the cursor-left routine to be executed at normal machine speed, thus eliminating the 30 millisecond wait per keystroke encountered by the operator if attempting multiple cursor-left commands from the keyboard. This gives the operator very high speed cursor-left capability. The entire screen may be sequenced through in approximately 3/4 of a second (Figure 45).

3.8.5 CSRR

This is a routine designed to provide the operator with a cursor-right command. When the correct command is decoded in the
Figure 45. Cursor-Left Five Times Logic Diagram
editor package, machine command continues at the label "CSRR". For purposes of this thesis, it was decided that should the cursor be found at the CRT bottom right position, no further action by this command would be executed. This simplifies the command and provides at the same time suitable cursor-right flexibility.

The logical flow chart may be found in Figure 46, and machine execution begins by testing for the end of the text. If found, the command is ignored and a return from subroutine is executed. If not, a test for a carriage return is performed and, if found, the line value is determined. If the line value is currently at 12 (/0C), no further action is performed and a return is executed. If not, the cursor is located on the next line down at the left edge by use of the line feed and carriage return characters sent to the CRT.

In all other instances, a simple "cursor-right" command is sent to the CRT terminal. The stack pointer is then relocated one position down in memory, and a return from subroutine is executed.

3.8.6 CSRR5

The following short routine is designed to execute the "cursor-right" command five times, thus considerably speeding up the cursor positioning techniques. This effectively avoids the inherent "roll-over" delay introduced by the keyboard when repeated control characters are sent from the CRT. The entire CRT may be scanned by the cursor in less than 3/4 of a second (Figure 47).
CLEAR THE Y-REG TO AVOID OFFSETS

IS THE SP POINTING AT AN 'EOT' CHAR?
  YES
  RTS
  (CSRR2)
  NO

IS THE SP POINTING AT A 'CR' CHAR?
  YES
  SEND A CURSOR-RT TO THE CRT
  (CSRR1)
  NO

IS THE LINE COUNT OF THE CRT = 120?
  YES
  SEND A 'LF' & A 'CR'
  TO THE CRT; REPOSITION
  THE SP 1 LOCATION ON
  ON THE STACK
  INCREMENT THE LINE COUNT
  RTS

Figure 46. Cursor-Right Command Routine
Figure 47. Cursor-Right Five Times Logic Diagram
3.8.7 DELETE

This routine is designed to provide the operator with the ability to delete a given character at the current cursor location. For the purposes of this project, the additional characters following the cursor location would be moved up one location in memory and repositioned on the CRT display.

The machine code begins by saving the current stack pointer location and clearing the variable "MODE". If the deleted character is a carriage return, then the entire screen must be rewritten from the current location onward. If not, then only the current line must be rewritten. Thus, if the deleted character is not a carriage return, "MODE" remains cleared and the test for the end of text is performed. Again if the end of text is found, an immediate return is executed and the delete operation is ignored. If we are not at the end of the text, then machine execution continues at label "DLET3".

The two-byte stack pointer is immediately repositioned one place down in memory and the Y-index is cleared for future deferred-Y addressing. From this initial location, until the end-of-text character is located, all characters are moved up one position in memory, including the "EOT". When completed, the stack pointer is restored to its original position, the "MODE" variable is tested for set or cleared status, and depending upon this, the corresponding line or CRT screen is rewritten.

After these operations, a return from subroutine is executed. The logical flow diagram for this subroutine may be found in Figure 48.
Figure 48. Delete-Character Operation
3.8.8 RSTSC

This next portion of source code contains a routine designed to reset the CRT screen. The logic flow diagram for this routine may be found in Figure 49.

The software begins by saving the current stack pointer location and clearing the variable "CNTR". From the current cursor position onward to the end of this line of the CRT the line is rewritten with each cursor position of the CRT filled by the next data character found on the data stack downwards. At the end of the current line an additional blank is transmitted (label "RST2") to erase a possible additional character. Once the line has been completed, the cursor is repositioned by first determining how far the original stack pointer was positioned from the start of the current line. The cursor is then cycled from the left edge of the screen to its original position, the stack pointer is restored to its original value, and a return from subroutine is executed.

The second portion of this routine contains software code related to the entire rewrite of the CRT screen. This is called independently of the first half and program execution ends at label "RST4". Again the program begins by saving the stack pointer but continues by erasing the entire screen from the current cursor location onward. Next, the current line value is saved in "TLINE", and the process for transmitting the new screen begins.

Each new character, fetched from the data stock by the ".DOWN" command is transmitted to the CRT utilizing the transmission at 9,600 Baud routine. Until either the line count, an end-of-text character,
Figure 49. Reset the CRT Screen Operation
Figure 49.—Continued  Reset the CRT Screen Operation
Figure 49.—Continued  Reset the CRT Screen Operation
or an error character is encountered, the software will continue to transmit data to the CRT for the screen rewrite.

When one of the three preceding situations is encountered, the routine is interrupted, the cursor is repositioned at the original position, stack pointer is repositioned to point at the original location, and a return from subroutine is executed.

3.8.2 INSRT

The following short routine contains the necessary software to insert a legal ASCII character transmitted from the CRT terminal under editor control. The logical flow chart for this routine may be found in Figure 50.

The general operation of this routine is to move each data byte one position down in the data stack and then insert the newly received character at the vacated position. The stack pointer is reset to its original value, now pointing to the newly received character. A ".DOWN" command is executed to move the 2-byte stack pointer down one location in memory, and a return from subroutine is executed.

3.8.10 EFILE

This short routine is designed to erase all characters in the data stack from the current cursor location (Figure 51). To perform this operation, an "end-of-text" character is inserted onto the data stack at the current stack pointer location and the CRT terminal screen is erased from the current cursor location onwards. The 2-byte stack pointer is correctly relocated and a return from subroutine is executed.
Figure 50. Insert a Character Subroutine

ENTER (INSRT)

SAVE THE 2-BYTE STACK POINTER

ECHO THE CHAR TO BE INSERTED BACK TO CRT

CLEAR Y REGISTER

LOAD 'CHAR1' WITH CURRENT STACK BYTE

PUSH 'CHAR' ONTO STACK

TRANSFER 'CHAR1' TO 'CHAR'

HAS 'EDT' BEEN FOUND ON STACK YET?

NO

YES

PUSH THE FINAL CHARACTER ONTO THE STACK & RESTORE THE 2-BYTE SP

REPOSITION THE SP ONE CHARACTER DOWN FROM ITS LAST POSITION

RTS
ENTER

MOVE THE 'END-OF-TEXT' CHARACTER INTO 'CHAR'

PUSH 'CHAR' ONTO DATA STACK

REPOSITION THE SP BACK TO ITS ORIGINAL LOCATION

'XMT AN 'ERASE' CHAR TO CRT & ERASE SCREEN FROM CURSOR ONWARDS

RTS.

Figure 51. Erase File Contents Utility
3.8.11 PRINT

This routine is designed to printout a given data file currently on the data stack upon command received through the monitor routine. This routine was included in the cursor package for programming convenience due to the lack of available memory left in the utility package "Master Utility Subroutines."

The software begins by clearing the CRT screen and repositioning the teletype carriage to the left margin. Assuming that the cursor has been correctly positioned on the stack, printing begins by using the "DOWN" command to fetch the next data byte from this stack. Tests are made for the end of stack, the end of text, and a carriage return. Upon encountering the first two, the software then continues at label "PRIN2" which completes the software responsibilities and terminates the output.

If the decoding process determines a carriage return, then both a carriage return and a line feed are sent to both the CRT and the teletype. This is required due to the fact that the action of the carriage return in both peripherals merely repositions the cursor (or the carriage) to the left margin. It is also desired that the line feed be sent to add an extra line to the printing process. If a carriage return was not detected, then a legal ASCII character was decoded from the data stack and was printed at both the CRT and the teletype.

Termination of this printing process involves the relocation of the cursor at the left edge of the CRT and relocation of the carriage at the left edge of the teletype. In addition, an initial
three lines are provided for the teletype for ease of viewing the completed text, and a return from subroutine is executed (see Figure 52).
ENTER

CLEAR 'CMD1', SEND A 'CLEAR' CODE TO THE CRT, & REPOSITION TTY CARRIAGE TO THE LEFT

USE SUBRHN 'DOWN' TO FETCH NEXT CHAR FROM STACK

DOES THE BYTE EQUAL EITHER AN 'EDT' OR AN 'ERR' CHARACTER?

NO

YES

IS IT A 'CR'?

NO

YES

XMT A 'LF' TO BOTH THE CRT & THE TTY

PLACE A 'CR' IN 'CHAR'

XMT 'CHAR' TO BOTH THE CRT & TTY

TEST THE CRT PORT FOR A SET BIT

SET

CLEARED

IS 'CMD1' CLEARED?

NO

YES

LOAD 'CMD1' WITH /FF

RESET TTY & CRT

RTS

Figure 52. TTY Print Service Routine
CHAPTER 4

MACROPROCESSING AND ASSEMBLY

4.1 Macroprocessor

To speed line programming techniques, it was decided that the use of Macro Definitions would provide the programmer with a far more flexible medium of writing source code. The use and rules of this macroprocessor are given along with the source code in McCartney (1977b). The processor is of use in the simplification of course code requirements and in providing the programmer with a slightly higher level language for programming purposes than the ordinary assembly-level code mnemonics.

Although the rules and restrictions for operation of this macroprocessor are completely outlined in the source code, a brief description of this application is presented here. The following general utilization rules apply:

1. A macro definition must precede a macro call.
2. The programmer is limited to less than 50 macro definitions.
3. Not more than 72 characters per text line are permitted.
4. A macro definition begins with the label ".MACRO".
5. The name of the macro definition must be separated from the macro label by at least 1 blank space.
6. Any dummy variables must be separated from the macro name by at least 1 blank space.
7. Dummy variables must be separated from each other by a comma.

8. In the replacement code, all dummy variables to be replaced by source code variables must be preceded by an apostrophe.

9. All unknown macro calls will be passed without error flag to the output.

10. The end of a macro definition must be flagged by the use of the command ".ENDM".

11. A macro call consists of a period followed by the macro name and, if necessary, a list or real variables separated by commas with at least 1 blank space following the macro name.

An example of a macro definition and its corresponding macro call may be found in Section 3.3.

This macroprocessor is designed as a complete software package and was written in the FORTRAN programming language for portability. The package contains all necessary comments, rules, and logical assignments for correct operation. In addition, the source code was written to conform to the standards of ANSI FORTRAN. Requirements for this program include at least 23 K words of memory core and corresponding ability to accept disc-based files.

For a successful operation of this program, two additional files are required for operation: the source list found in "INST.RHM", and a second file named "ROUT.RHM", used as destination vector. The second file contains no information at the beginning of the operation and is used as a storage medium for the processed output code.

In the resulting output code, all comments beginning with a semicolon are deleted from the text. Minimal error control was
incorporated into the processor so that nonstandard macro calls and syntax may be passed through the macroprocessor without hindrance of performance. The resulting output code in the file "ROUT.RHM" will contain no error messages, no comments, and is intended by the author to be ready for direct assembly. This computer program was written by the author at The University of Arizona's Department of Electrical Engineering in the spring of 1977. It has been successfully run on a DEC PDP-11/40 with a standard FORTRAN compiler supplied by the DEC Corporation.

4.2 Assembler

The translation of mnemonic source code into machine level code was performed by a FORTRAN-based cross assembler written by the author in the spring of 1977. This program (McCartney, 1977a) contains a total of three additional assembly-level programs that must be linked to the main program for correct operation. The assembly-level language is designed for the DEC PDP-11/40 minicomputer and was translated into relocatable object code by a DEC-supplied macro-assembler.

The cross assembler is a field-oriented assembler designed primarily for the MOS Technology model 6502 microprocessor. The source code comes in three fields:

Field 1: Label
Field 2: Machine Instruction
Field 3: Operand

Comments are inserted in the source code by being preceded with a semi-colon. Thus it is possible to write source code containing comments
without hindrance of performance. In all cases, the semicolon will be ignored. The standard mnemonic code syntax as supplied by the MOS Corporation was utilized by the author in preparing the software code for this project. In addition to the standard syntax labels, an additional two characters were added to the normal three-character symbol to indicate addressing mode. Thus, the absolute loading of the accumulator is designated by "LDAAB". The entire list of addressing mnemonics may be found in Table 15. The standard MOS Technology mnemonic instructions for the 6502 processor may be found in Software Application Manual (1976).

For correct operation of this assembler, the original source code must be located in a disc based file titled "MACHIN.RHM". In addition, the assembled machine code as well as the original source code will be printed out by the program at logical unit #9.

4.3 Operation

The macroprocessing and assembly of this thesis project source code was accomplished at the Department of Electrical Engineering at The University of Arizona on a DEC Model PDP-11/40 minicomputer in the fall of 1977. Original source code was organized into a logical ASCII file titled "INST.RHM" and stored on the system disc. Under control of the RT-11D Monitor, either the teletype and scope (logical name "TT") or the line printer (logical name "LP") was assigned the logical number "9". With this done and the macroprocessor program correctly compiled and linked, the operator then typed the instructions to run file program "LABI". At the completion of the macroprocessing, the operator is
Table 15. MOS 6502 uP Assembly Code Addressing Mnemonics

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Corresponding Addressing Mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>(blank)</td>
<td>Branch instruction or Accumulator-Mode</td>
</tr>
<tr>
<td>IM</td>
<td>Immediate</td>
</tr>
<tr>
<td>AB</td>
<td>Absolute</td>
</tr>
<tr>
<td>ZP</td>
<td>Zero Page</td>
</tr>
<tr>
<td>XD</td>
<td>(Index X)</td>
</tr>
<tr>
<td>DY</td>
<td>(Index Y)</td>
</tr>
<tr>
<td>ZX</td>
<td>Zero Page X</td>
</tr>
<tr>
<td>ZY</td>
<td>Zero Page Y</td>
</tr>
<tr>
<td>AX</td>
<td>Absolute X</td>
</tr>
<tr>
<td>AY</td>
<td>Absolute Y</td>
</tr>
<tr>
<td>IN</td>
<td>Indirect</td>
</tr>
</tbody>
</table>
prompted with a request to continue and review the processed output. This may be terminated by simply typing "control-C". At this point, the operator types the commands to run program "LAB4". The first pass of the cross assembler will print a header and any incurred errors during that run. If no errors are detected, a prompting message requesting a carriage return to continue is presented to the operator. The second pass will print at logical unit "9", including not only the assembled machine code by the original source code as well.
CHAPTER 5

DOWNLOAD OPERATION

The download process was required for this thesis project due to the split nature of the software organization. Original source code files were constructed with the use of a text editor on a PDP-11/40 minicomputer and then stored as ASCII files on discs. In addition, the macroprocessing and assembly operations were again performed on this minicomputer resulting in a disc-based output file consisting of the assembled machine code to implement the subroutines listed in Chapter 3. However, for correct execution, it was necessary to send this information from the minicomputer to the prototype test bed. This required use of the serial data communication line and routines necessary to transmit the data from the minicomputer to the KIM-1.

The downloading routines, as noted in Appendix C, were written for two major functions: the normal download process itself and the initialization process whereby a download reception program is originally placed into a KIM-1 unit.

The first program was written by the author and is titled "KIM-1 Mini-Loader." This program is written in the assembly-level mnemonic instructions for the 6502 microprocessor and is processed and assembled machine code is hand-loaded into the KIM-1, and then is run in the normal fashion. This loader program is a simplified version of a more sophisticated program to follow shortly. In essence, it merely
accepts serial data, selects the next sequential location, and inserts the data byte at that location. No error control, advanced memory selection techniques, or program termination are supplied.

The next program is the main down loader program which must eventually be linked with an assembly-level subroutine for correct operation by the DPll minicomputer. This program transmits data in a serial format to the prototype test bed complete with control bytes for termination of program, changes of absolute address, and continuation of current address. The final data byte sent to the KIM-1 unit is preceded by the control byte "STOP". The data byte contains a sumcheck performed on all previous data bytes sent from the minicomputer. This sumcheck at the end of transmission is displayed, a complete message is given to the operator, and the subsequent data files are closed.

The necessary assembly-level subroutine required for successful operation of the main downmoding program was written by the author with the assistance of Fil J. Yeskel at The University of Arizona's Department of Electrical Engineering. This subroutine permits the transmission of serial data converted from ASCII code into binary code. These data are clocked out through a given data port of the minicomputer along with a parallel signal line that indicates to the KIM-1 when a data bit is available.

The assembly-level language routine used for complete downloading and reception of serial data from the minicomputer at the KIM-1 unit is titled "KIM-1 Assembly Language Loading Routine." This routine is written in the standard mnemonic language developed for the main source code of the utility routines presented previously in Chapter 3.
and performs far more functions than the simplified, hand-loaded routine mentioned previously. This KIM-1 loading routine accepts serial data from the PDP-11 minicomputer and translates them into two bytes: a single control and a single data byte. The three general commands for the control bytes are: increment previous address for new address, accept data byte as a new address (high or low order), and terminate download process.

Upon reception of the termination control byte from the minicomputer, this routine causes the KIM-1 to perform a test between the received byte indicating the sumcheck developed by the minicomputer and the sumcheck developed internally by this routine in the KIM-1. If the result is identical, the LED memory locations are filled with \(CC\)'s and a jump is made to the KIM-1 memory routines. This provides the operator with the immediate message that all data were correctly downloaded. However, should an error occur during transmission, the sumchecks would not match, and the resulting error flag is displayed by the LED's. The error flag consists of \(EE\)'s placed at the LED memory locations.

Successful operation of this download process is dependent upon the correct operation of the KIM-1 loading routine and the operation of the main downloading program located in the PDP-11. The serial format used in data transmission requires three signal lines and a respective ground between the minicomputer and the prototype. Line 1 constitutes a data line on which the respective data bit is placed. The minicomputer program then raises line number 2 which is designated "DS", and waits a predetermined amount of time (approximately 100 microseconds) for the KIM-1 to process the bit. Then, without indication
from the KIM-1, the minicomputer lowers the data strobe line and prepares the next data bit on the data line. When the sixteenth data bit is transmitted from the minicomputer to the KIM-1, in addition to the data strobe line going high, the "end-of-word" line is raised by the minicomputer. This is a message to the KIM-1 software that a sixteen-bit word has been completely transmitted and that the KIM-1 should prepare for a new data word. The end-of-word line is then lowered, approximately 100 microseconds later and the first data bit of the next computer word is subsequently placed on the data line. Figure 53 denotes the resultant logic flow and approximate timing found on these three lines. As a general note, the author would like to state that during his element operation of the software, the downloader program had been run over 180 times without a single error in data transmission.
26-Pin KIM-1/PDP-11 Connector

```
1 3 5 7 9 11 13 15 17 19 21 23 25
2 4 6 8 10 12 14 16 18 20 22 24 26
```

<table>
<thead>
<tr>
<th>Pin</th>
<th>Function</th>
<th>KIM-1 Addr.</th>
<th>PDP-11 Addr.</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DATA</td>
<td>Bit 1 or /702</td>
<td>LSB or #167772</td>
</tr>
<tr>
<td>2</td>
<td>DATA GROUND</td>
<td>Bit 2 or /702</td>
<td>Bit 1 or #167772</td>
</tr>
<tr>
<td>5</td>
<td>STROBE</td>
<td>Bit 3 or /703</td>
<td>LSB or #167772</td>
</tr>
<tr>
<td>6</td>
<td>STROBE GROUND</td>
<td></td>
<td></td>
</tr>
<tr>
<td>25</td>
<td>END-OF-WORD</td>
<td></td>
<td></td>
</tr>
<tr>
<td>26</td>
<td>END-OF-WORD GROUND</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 53. Download Timing and Pinout Connections
A proposed design for a viable word processor is shown schemat­ically in Figure 54. Virtually all major logic functions are delegated to the CPU, with only minor household duties relegated to peripheral devices. This configuration is designed to relieve the microprocessor from major transmission duties, and results in far more simplified software.

This first major departure from this thesis project in final design criteria was the reorganization of the CRT monitor logic. It was decided that a more streamlined design could be obtained resulting in less overhead imposed upon the CPU by installing independent monitor driving circuitry and the corresponding screen memory (approximately 4K). Access to this memory sector would be performed by use of a peripheral interface adapter (PIA). Thus, by use of a PIA, only momentary halts of the monitor driving circuitry are imposed by the CPU during screen memory access. The CPU is permitted to operate directly on the screen memory, avoiding the serial data transmission techniques imposed by the prototype. The imposed delay on the screen output timing would be invisible to an observer.

General memory requirements would be provided by two primary areas: approximately 8K region of random access memory and approximately 6K of read-only-memory. The read-only-memory (ROM) will house
Figure 54. Proposed Design Logic Schematic
the basic monitor routines and text editor, thus eliminating the necessity of a bootstrap operation prior to its use. This simplifies operator controls and reduces operator training and functional errors. The size of random access memory was chosen merely as a convenience to permit a relatively large data file to be stored at any given instant.

It was decided that both high speed and high quality ranked equally among the major points of a system printer. Since most of the major high quality low-to-moderate speed printing mechanisms provide interfacing with either serial or parallel modes, a parallel-input, 60 character-per-second daisy wheel printer was considered suitable for this operation. Several models are available currently on the commercial market. Interfacing to this peripheral would require exactly one-half of a peripheral interface adapter.

Keyboard entry of data from an operator would be performed with a standard, light tough, high speed, 12-degree keyboard with tactile response. The keyboard would provide data in parallel format to the microprocessor, and would interface through one-half of a PIA as shown in the diagram. The control of at least two flexible disc drives would be performed through a flexible disc controller similar to the SMS model FD 0305-2. This board would interface to the processor by use of one-half of a PIA, plus the two independent control lines supplied by the standard PIA format. The flexible disc controller would then interface directly through a 4-pin parallel ribbon connector with suitable open-collector terminations. This interfacing would be identical to the format discussed in the prototyping model.
The resulting system model as described would be able to perform in the following functions as a word processor:

1. Very high speed, simplified text-editing through use of a CRT terminal, specialized keyboard functions, and high speed software utilities.

2. Bulk storage of data file on flexible disc drive. At least 200 pages of doubly-spaced text may be stored per side of a flexible discette.

3. Sixty-character per second, very high quality printing output.

4. Simplified operator familiarization.

5. Simplified system expansion. Multi-work station potential is an important design aspect for expanding office loads.

A simplified cost estimate revealing only the required components purchased by an original equipment manufacturer is now presented. The resulting parts list and component cost are presented in Table 16. The author wishes to note that this preliminary estimate does not include labor, manufacturing overhead, warehousing and transportation, sales commissions, distributor fees, or corporate profits. In addition, the parts indicated do not represent a thorough corporate search of existing equipment available to the designer, and are intended only as a general guide to the expected prices to be found in the industry at this time.
# Table 16. Parts List and Projected Material Costs

<table>
<thead>
<tr>
<th>Component</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Keyboard</td>
<td>$200</td>
</tr>
<tr>
<td>32 MHz CRT Monitor Screen</td>
<td>$650</td>
</tr>
<tr>
<td>CPU, CLOCK, and Support Circuitry</td>
<td>$100</td>
</tr>
<tr>
<td>Monitor Screen Memory (4096 bytes)</td>
<td>$250</td>
</tr>
<tr>
<td>Scratchpad Memory (8192 bytes)</td>
<td>$450</td>
</tr>
<tr>
<td>Monitor Driving Circuitry</td>
<td>$300</td>
</tr>
<tr>
<td>Read-Only-Memory (6 K bytes for Monitor and Editor)</td>
<td>$350</td>
</tr>
<tr>
<td>Two Flexible Disc Drives</td>
<td>$1300</td>
</tr>
<tr>
<td>One Disc Drive Controller</td>
<td>$680</td>
</tr>
<tr>
<td>Interfacing Circuitry and Bus Structure</td>
<td>$150</td>
</tr>
<tr>
<td>High-quality Daisy Wheel Single Head Printer</td>
<td>$2600</td>
</tr>
<tr>
<td>Power Supplies</td>
<td>$550</td>
</tr>
<tr>
<td>Cabinetry</td>
<td>$400</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>$500</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>$8480</strong></td>
</tr>
</tbody>
</table>
CHAPTER 7

FUTURE APPLICATIONS AND EXPANSION

The final section of this report will be oriented not toward the possible expansion of the design for a word processing unit, but rather toward one potential evolution design proposed to the author at the conception of this project. This possibility entails the diversion of hardware and software from a word processing context to that of a generalized instruction laboratory pertaining to microprocessors for students enrolled at The University of Arizona. The laboratory design has many characteristics similar to the design goals of an intelligent word processor. It is expected that application of software listed in this report will find considerable use as an instructional tool.

The first possibility in expanding the software capability of this prototype is to include the use of a string search. A string search is normally defined as the ability of a processor to take a given set of data characters, search the current data file for a sequence matching the given characters, position the stack pointer and resulting screen cursor at the beginning of the desired character strong, and then present the data to the operator at the CRT. One possibility is presented in Figure 55 describing the required software instructions to execute a suitable string search. Although this subroutine has not been tested due to its omission at the project conception, the preliminary logical flow chart and proposed software instructions are presented in
THE FOLLOWING ASSEMBLY-LEVEL ROUTINE IS WRITTEN FOR THE KIM-1 MICROPROCESSOR EVALUATION BOARD AND IS INTENDED TO ILLUSTRATE A BASIC FORM OF STRING SEARCH. THE SOURCE CODE IS TO BE PROCESSED AND ASSEMBLED USING THE AUTHOR'S MACROPROCESSOR AND ASSEMBLER (SEE REFERENCES #27 AND 28). ALL MACRO CALLS INCLUDED BELOW ARE TO BE INTERPRETED BY THE DEFINITIONS GIVEN WITH THE MAIN THESIS SOURCE CODE (SEE APPENDIX A).

NOTE: THIS PROGRAM HAS NOT BEEN TESTED. OPERATION REQUIRES EXTENSIVE DEBUGGING.

WE BEGIN BY PROMPTING THE OPERATOR WITH THE REQUEST FOR A CHAR STRING: "STRING?".

```
STR: LDXIM /00
STR1: MOVE AX,STRN,ZP,CHAR
  XMT96
  INX
  CMXIM /08
  ONE STR1
  .GET-NAME
    USE *.GET-NAME* TO FETCH CHAR STRING.
    LDXIM /0A
    COUNT BACKWARDS IN 'FILNM' TO LOCATE
STR2: DEX
    1ST NON-BLANK CHARACTER.
    .CMPR IM,BLANK,AX,FILNM
      IF 'FILNM' CHAR IS A BLANK, LOOP.
    BEQ STR2
    CPXIM /00
      TEST IF X-REG POINTS TO START OF STRING.
    BPL STR3
      IF NOT, DON'T RETURN & BRANCH AROUND
    RTS
      THE 'RTS'.

STR3: INX
    POINT X-REG AT 1ST BLANK OF TRAILING BLANKS.
    STXAB RNMX
    STORE IT IN 'RNMX'.
    LDXIM /00
    .SP-SET TOS-H,TOS-L
      SAVE THE STACK POINTER FOR LATER.
STR4: DOWN
    FETCH NEXT CHARACTER ON STACK.
    .CMPR ZP,CHAR,IM,EOT
      IF EOT, RETURN WITH ERROR CODE '4,2'.
    BEQ STR7
    .CMPR ZP,CHAR,AX,FILNM
      IF NO MATCH BETWEEN 'FILNM' & STACK CHAR,
```

Figure 55, Proposed String Search Subroutine
BNE STR4 ; CONTINUE TO LOOP DOWN THE STACK.
.SP-SAVE ; HERE, THE 1ST CHAR OF 'FILNM' MATCHES.
STR5:
INX ; SAVE THE SP & MOVE OFFSET POINTER DOWN ONE.
CMPXAB RNAMX ; LOOP TIL OFFSET PTR MATCHES STORED OFFSET.
BEQ STR6 ; IF A MATCH, STRING WAS FOUND; THUS EXIT.
.DOW ; FETCH NEXT CHAR ON STACK.
.CMPR ZP,CHAR,IM,EOT ; IF END-OF-TEXT, LOOP AROUND NEXT TEST.
BEQ STR8 ; IF STACK CHAR MATCHES 'FILNM' CHAR, LOOP
.CMPR ZP,CHAR,AX,FILNM ; IF STACK CHAR MATCHES 'FILNM' CHAR, LOOP
BEQ STR5 ; BACK FOR ANOTHER TEST.
CMPR ; MATCHING IS DONE & DOES OFFSET EQUAL THE
BEQ STR6 ; STORED VALUE OF THE OFFSET IN 'RNAMX'?
.SP-RESTORE ; IF NOT, RESTORE THE SP TO ORIG VALUE &
JR STR4 ; CONTINUE SEARCH THROUGH REST OF STACK.
.SP-RESTORE ; A VALID STRING MATCH WAS FOUND. THE RETURN
MOVE IM,/01,ZP,LINE ; WILL POSITION THE DESIRED STRING AT THE
.JMP ; TOP OF THE CRT SCREEN WITH THE CRT RESET.
JMP EDIT ; THE PROGRAM RESUMES IN 'EDIT' MODE.
.SP-RESTORE ; NO STRING MATCH WAS FOUND, & THE ERROR CODE
MOVE IM,/34,AB,NUM1 ; '4.2' IS SENT TO THE OPERATOR VIA ROUTINE
.MOVE IM,/32,AB,NUM2 ; 'TST6'.
.JMP TST6 ; THE PROMPTING MESSAGE IS NOW INSERTED INTO THE PROGRAM CODE.

STRNG:
BYTE /53 ;
BYTE /54 ;
BYTE /52 ;
BYTE /49 ;
BYTE /4E ;
BYTE /47 ;
BYTE /3F ;
BYTE /20 ;

END

Figure 55.---Continued
this section for future application. Another possibility would be to redefine the "cursor-up" function to provide the effective "scroll-down" result that is more comfortable to witness as an operator than the existing screen rewrite function. This would entail a peripheral that would accommodate the data shift required for a "scroll-down" operation or the ability to execute an extremely fast screen rewrite. The second alternative mandates a very high speed serial data interface or a parallel data interface to the CRT peripheral. The current data rate of 9600 bits per second results in a noticeable flicker which may become uncomfortable to an operator after repeated use. It is also somewhat cumbersome to use due to the abrupt jump of the cursor to a new line position at the top of a previous page.

A provision for a "line-delete" function involving the complete erasure of a chosen line would be desirable for future use as a word processor. This would enable an operator to remove entire, selected lines with a simple command. The change would allow extremely flexible deletion on a larger scale than the current "delete" command of a single character at a time and the awkward "erase" command of eliminating a data file from the current cursor location onward.

The disc storage routines currently provide a fixed length file for data storage. A more flexible operating mode would be to provide a variable length file under software control which would be completely invisible to the operator. This would eliminate any necessary operator commands at storage time and provide a much more efficient storage mode for files on the flexible disc. To accommodate future expansion in this direction, the current design has included an additional six bytes per
data file that may include such items as absolute track and sector address and data file lengths.

The ability to transfer data files from one disc to the next would provide an operator with the capability to produce backup files of important data. This permits redundant information and the ability to retrieve data should a file be lost. The author notes that this function may be obtained without the use of an internal disc buffer by transfer of data solely through the flexible disc controller board buffer.

Control of disc choice, disc operations, and future software combinations could be supplied through the current industry time sharing standard of "switches." Slash marks added after a given command or character string would then modify the action performed for the user, but must require additional, more sophisticated software for adequate decoding. Thus the choice of disc to disc operation, storage of non-standard characters, concatenation of files, and special deletion-insertion-manipulation of data files may be performed through use of the correct switch, decoding, and software operation.

Another major point for future applications is the concept of block manipulation for more sophisticated, advanced data manipulation. For use in a word processing environment, block manipulation was not deemed necessary to fulfill requirements of this project. However, for a laboratory environment or instructional facility, block manipulation of software code in a chosen format would be an extremely powerful tool for programmers. The ability to switch block-oriented routines, data, and results would permit extremely flexible restructuring of a software
package under development by a student or researcher. This block manipulation would provide the operator with the ability to insert blocks at a given file location, delete designated lines of a given data file, preserve particular regions of a data file, and rearrange a sequence of segments of a desired data file.

The final application proposed by the author is the ability of the monitor routine to branch or jump from its current operation under correct decoding of a user-given command to a machine-code program and continue with its operation. This can be very simply done by assuming that during the decoding sequence in the monitor routine, should a command for running a given data file be encountered, machine execution is transferred to the site of the desired object code data file. Return from the new program to the original monitor routine could be performed by two methods: either an absolute jump to the monitor initialization segment at the end of the program, or by a nonmaskable interrupt initiated by the operator (by pressing the "ST" KIM-1 key). The addressing and designated jump vectors must be correctly assigned to the new program's object code, and the resulting return should be correctly provided by the routine. These responsibilities should be closely coordinated between the structuring of the monitor routine and the assembler (or linking-loader) to be employed by this system in the future.
MOS 6502 MICROPROCESSOR
SOURCE CODE

This program is assembly-level code designed for the MOS-6502 micro-
processor and is to follow format rules established for the author's
cross-assembler and macro-processor.

The purpose of this program is to control the required functions that
are sought in an intelligent typewriter based upon a micro-processor.
This software is written to fulfill the requirements of EE410: Master's
thesis.

Author: R. H. McCartney
Date: Summer/Fall, 1977
Location: Computer Science Laboratory, Dept. of Elect. Engr.,
University of Arizona, Tucson, AZ.
Machine: PDP-11/40 W/RT-11B Monitor
File: T.RHM
Project: EE 410 Master's Thesis

MASTER SYSTEM
UTILITIES

The initial address is chosen at the bottom of the zero page.
.org /0000

The first section of the software package lists single- and double-
byte variables. Using the '.BYTE' macro command, each variable is
assigned a memory address on the zero page and initially given the
value of the operand field.

CNTK: .BYTE /00
CNTKL: .BYTE /00
CNTKR: .BYTE /00
CHAR: .BYTE /00
CHAR1: .BYTE /00
CM1: .BYTE /00
CMD1: .BYTE /00
CMD2: .BYTE /00
DISK: .BYTE /00
SF-L: .BYTE /00
SF-H: .BYTE /00
TSP-L: .BYTE /00
TSP-H: .BYTE /00
SECTR: .BYTE /00
TRACK: .BYTE /00
BITS: .BYTE /00
LINE: .BYTE /00
TLINE: .BYTE /00
MODE: .BYTE /00
STATS: .equ /10
THE ABSOLUTE ADDRESSES OF PERIPHERAL EQUIPMENT IS NOW DENOTED USING THE 'EQU' MACRO COMMAND. THE ABSOLUTE ADDRESS OF EACH CODED LOCATION IS LISTED IN A TABLE FOR FUTURE USE BY THE ASSEMBLER.

TTY: .EQU /1742 ITTY DATA PORT.
DDR2B: .EQU /1743 ITTY DATA DIRECTION REG.
CRT: .EQU /1702 HI-DIR PORT ASSIGNED TO CRT.
DDR3B: .EQU /1703 ICTX DATA DIRECTION REG.
DSK1: .EQU /1008 IDISK CONTROL PORT
DSK2: .EQU /1700 IDISK DATA/CMMD PORT
DSK3: .EQU /1010 IDISK STATUS PORT
DDR3A: .EQU /1701 IDATA DIRECTION REG (DSK DATA).

TO EXPEDITE THE ARRANGEMENT OF STRING STORAGE, THE STACK ADDRESSES ARE NOW GIVEN LABELS. THUS, FUTURE REARRANGEMENTS NEED ONLY CHANGE THE VALUES IMMEDIATELY BELOW TO REPOSITION THE TEXT STACK AND THE HOST/DISK BUFFER AREA.

TOS-H: .EQU /04 'TOP-OF-STACK' HIGH ORDER BYTE.
TOS-L: .EQU /02 'TOP-OF-STACK' LOW ORDER BYTE.
BOS-H: .EQU /02 'BOTTOM-OF-STACK' HIGH ORDER BYTE.
BOS-L: .EQU /00 'BOTTOM-OF-STACK' LOW ORDER BYTE.
BUF-H: .EQU /04 TOP OF DISK BUFFER AREA (HIGH).
BUF-L: .EQU /00 TOP OF DISK BUFFER AREA (LOW).
BUFFR: .EQU /0401 'BOTTOM OF THE DISK BUFFER AREA.'
NMIVH: .EQU /17FB (NON-MASKABLE INTERRUPT VECTOR (HI).
NMIVL: .EQU /17FA (NMIV (LOW).

NEW PROGRAM POSITIONS FOR ADDITIONAL BLOCKS OF OBJECT CODE ARE NOW DEFINED. FOR FUTURE RELOCATION, ONLY THE VALUES ITEMIZED BELOW NEED BE CHANGED.

INTIL: .EQU /0481 ISTART OF INITIALIZATION RTN.
MONTR: .EQU /0487 ISTART OF MONITOR ROUTINE.

MONFH: .EQU /1C HIGH BYTE OF NMI VECTOR.
MONFL: .EQU /00 LOW BYTE OF NMI VECTOR.
CURSRH: .EQU /0100 ISTART OF CURSOR UTILITY PACKAGE.

SINGLE-BYTE CONSTANTS USED THROUGH THE PROGRAMMING ARE NOW EQUATED TO THE DESIRED VALUES THEY REPRESENT. THESE CONSTANTS REMAIN INVARIANT THROUGHOUT ALL ROUTINES AND ARE ESTABLISHED AS AN AID TO THE OPERATOR FOR BOTH PROGRAMMING AND FAULT-SEARCH WORK.

LF: .EQU /0A LINE FEED CHARACTER CODE.
CR: .EQU /0D CARRIAGE RETURN CHARACTER CODE.
QUEST: .EQU /3F ASCII QUESTION MARK.
ASTKS: .EQU /2A ASCII ASTERISK MARK.
BLANK: .EQU /20 ASCII SPACE CHARACTER.
DASH: .EQU /2D ASCII HYPHEN MARK.
DOT: .EQU /2E ASCII PERIOD.
EDIT: .EQU /0E IEND OF TEXT CHARACTER.
ERR: .EQU /7F IERROR FLAG.
CLEAR: .EQU /1C ICLEAR CHAR---> CLEARS CRT.
ERASE: .EQU /0C IERASE CRT FROM CURSOR ON.
C-HM: .EQU /1E IASCII CHARACTER FOR 'CURSOR HOME'.
NUM: .EQU /00 IASCII 'NO-OPERATION' CODE.
C-UP: .EQU /0B ICURSOR UP.
C-DN: .EQU /0A ICURSOR DOWN (SAME AS 'LF').
C-RIGHT: .EQU /1F ICURSOR RIGHT.
"R: .EQU /12 IT FOR CURSOR RIGHT 5 TIMES.
C-LFT: .EQU /0B ICURSOR LEFT.
"L: .EQU /0C IXTM ONLY 'L FOR CSR LFT 5 TIMES.
"D: .EQU /7F IDELETE A SINGLE CHARACTER.
"E: .EQU /05 ENTER EDIT MODE.
`W: .EQU /17 ; WIPE A FILE TO END OF STACK.
°C: .EQU /03 ; RETURN TO COMMAND MODE.
 serialVersionUID = 22 ; ABSOLUTE DISK READ OPERATION.
%A: .EQU /42 ; DISPLAY DISK CONTENTS.
D: .EQU /44 ; DELETE A DISK FILE.
E: .EQU /46 ; ASCII CHAR 'E'.
F: .EQU /46 ; FORMAT A CHOSEN DISKETTE.
K: .EQU /4B ; KILL A DISK FILE.
L: .EQU /4C ; LOAD A FILE FROM DISK.
N: .EQU /4E ; ASCII CHAR 'N'.
O: .EQU /4F ; ASCII CHAR '0'.
T: .EQU /54 ; GO TO TYPEWRITER MODE.
W: .EQU /57 ; ASCII CHAR 'W'.

MACRO-DEFINITIONS FOR FUTURE USE ARE NOW ESTABLISHED.

**MACRO RESET**

LDXIM /00
SIXAB CMD1
SIXAB CMD2
JSR CMD0
ENDM

**MACRO SEEK**

JSR SEEK

**MACRO PUSH**

JSR PUSH

**MACRO PULL**

JSR PULL

**MACRO DISK-CMMD**

JSR CMD0

**MACRO READ**

JSR READ

**MACRO WRITE**

JSR WRITE

**MACRO XMT96**

JSR XMT96

ENDM
MACRO RCV96
JSR RCV96
ENDM

MACRO XMT11
JSR XMT11
ENDM

MACRO CMFR A,B,C,D
LDA'C D
CMP'A B
ENDM

MACRO MOVE E,F,G,H
LDA'E F
STA'G H
ENDM

MACRO XMT I
LDYIM I
JSR XMT95
ENDM

MACRO SEARCH/TBL
JSR SRCHT
ENDM

MACRO CLEAR J
LDAIM /00
STA'ZP J
ENDM

MACRO TSTDSK
JSR TSTDK
ENDM

MACRO GETNAME
JSR GTNAM
ENDM

MACRO CRT-RST
JSR RST4
ENDM

MACRO SP-SET HIGH,LOW
LDAIM 'HIGH
LDYIM 'LOW
JSR SETSP
ENDM

MACRO SP-SAVE
JSR SAVSP
ENDM
MACRO SP-RESTORE
    JSR RS TSP
ENDM

************************************************************
MACRO XMTLF
    JSR XM TLF
ENDM

************************************************************
MACRO INC-SECTOR
    JSR SCTR
ENDM

MAJOR
UTILITY SUBROUTINES

THIS FIRST SUBROUTINE IS THE EQUIVALENT OF A ‘PSH’ MNEMONIC OPERATION 
EXCEPT THAT A TWO-BYTE STACK POINTER IS USED AND MAY BE INITIALIZED 
ANYWHERE. THE TWO BYTES USED AS THE EFFECTIVE ADDRESS ARE ‘SP-L’ 
AND ‘SP-H’, BOTH OF WHICH ARE LOCATED ON THE ZERO PAGE OF MEMORY.
NOTE:
The accumulator and Y-reg are used. Previous contents of 
these registers are lost.

(SP) < — CHAR
SP < — SP-1

PUSH: .CHFR ZP,SP-L,IM,POS-L;TEST IF LOWER BYTE HAS ATTAINED '/FF'.
PUSH1: BNE PUSH1 ;IF NOT, DON'T TEST UPPER BYTE.
BEQ ERROR ;SEND 'ERR' CODE BACK TO MAIN PROGRAM.
PUSH1: LDYIM /00 ;CLEAR THE Y-INDEX.
.CHFR IM,CHAR,DP,SP-L ;PUT THE CHAR TO BE STORED ON STACK.
DECZP SP-L ;LOW-BYTE OF ADDR IS DECREMENTED.
.CHFR IM,SP-H,SP-L ;TEST IF UPPER BYTE MUST BE DECREMENTED.
DECZP SP-H ;DECREMENT UPPER BYTE.
RTS ;RETURN.

THE FOLLOWING SUBROUTINE CONDUCTS A DATA-PULL FUNCTION USING AN 
ABSOLUTE Address OF TWO BYTES. THIS IS THE COMPLEMENT OPERATION TO 
the ‘PUSH’ SUBROUTINE LISTED ABOVE. THE SAME TWO BYTES FOR THE 
eFFECTIVE ADDRESS OF THE STACK POINTER ARE USED.

SP<---SP+1
CHAR<--(SP)

FULL: .CHFR ZP,SP-L,IM,TOS-L ;TEST IF THE SP IS AT THE TOP OF THE
BNE FULL1 ;DATA STACK. IF SO, RETURN VIA
BEQ ERROR ;‘ERROR’.
FULL1: INCZP SP-L ;PERFORM A TWO-BYTE INCREMENT ON THE SP.
BNE FULL2
INCZP SP-H
FULL2: LDYIM /00 ;CLEAR THE Y-REG SO NO OFFSET OCCURS.
.CHFR DP,SP-L,CHAR ;PLACE 'CHAR' INTO NEW LOCATION CHOSEN
BNE ERROR; BY SP, % RETURN.
FULL3: RTS
EPFDY: .MOVE IM,EP,DP,CHAR ;ERROR CONTINUE.
THE FOLLOWING SUBROUTINE PERFORMS A 'DOWN' OPERATION. IT IS SIMILAR TO THE 'PUSH' MACRO EXCEPT THAT DATA IS READ INTO A CHARACTER BUFFER FROM THE STACK INSTEAD OF DATA BEING WRITTEN ONTO THE STACK FROM A CHARACTER BUFFER.

CHAR <- (SP)
SP <- SP-1


BNE DOWN1: .MOVE DP,SP-L,ZP,CHAR PLACE 'CHAR' INTO LOCATION CHOSEN BY .CMRR SP-L; SP.

DOWN1: LDYIM /00 CLEAR Y-REG SO NO OFFSET OCCURS.
.MOVE BY,SP-L,ZP,CHAR PLACE 'CHAR' INTO LOCATION CHOSEN BY .CMRR IM, /FF,SP-L,SP-D DECREMENT THE SP.
BNE DOWN2: .DECZP SP-H

DOWN2: RTS RETURN.

THE FOLLOWING SUBROUTINE RECEIVES A 7-BIT CHAR AT 9600 BAUD UNDER THE GIVEN CONDITIONS: 1 START BIT, 7 DATA BITS, 1 PARITY BIT (WHICH IS IGNORED), AND 1 STOP BIT. THE RESULTING 7-BIT CHAR IS PLACED IN LOCATION 'CHAR' WHEN A RETURN TO MAIN PROGRAM IS EXECUTED. THE MSB WILL ALWAYS BE ZERO UNLESS A FALSE START IS DETECTED. FOR THIS ERROR, THE NUMBER 'BO' IN HEX IS RETURNED IN 'CHAR'. THE MSB OF DATA PORT 'CRT' IS WIRE AS THE INPUT DATA LINE (ASSUME INVERTED TTL DATA LEVELS).

DUE TO THE DESIRE FOR EASE OF ACCURATE TIMING, NO MACRO DEFINITIONS WERE USED IN THIS ROUTINE. ONLY THE ACCUMULATOR AND Y-REG ARE USED.

NUMBERS ENTERED AT THE START OF EACH LINE COMMENT DENOTE THE "NUMBER OF MACHINE CYCLES REQ U IRRED TO EXECUTE THAT PARTICULAR INSTRUCTION. " BRANCH INSTRUCTIONS VARY IN DURATION.

RCVP6: LDA IM /01 ESTABLISH READ/WRITE DIRECTIONS.
STAA DDR3B
LDAZP /00 CLEAR THE CHARACTER BUFFER.
STAZP CHAR

RCVO: LDA AB CRT FETCH THE SERIAL DATA LINE LEVEL.
ANDIM /00 BLANK ALL BITS EXCEPT THE MSB.
BNE RVCO LOOP IF NO START BIT IS SEEN.
LDYIM /04 SET A 1/2-BIT DELAY COUNTER
STYZP CNTRL IN LOCATION 'CNTRL'.
LDAZP 07 USE Y-REG AS BIT COUNTER.

RCV1: .DECZP CNTRL USE OF CYCLES COUNT IS TO COUNT 7 START BITS.
BNE RV1 12 LOOP IF ADDITIONAL BIT REMAIN.
LDA AB CRT LOAD DATA FROM CRT.
ANDIM /00 BLANK UNNECESSARY BITS.
BNE ERR2 TEST IF START BIT REMAINED '1'.

RCV2: LDAZP /0A SET COUNTER FOR 1-BIT DELAY.
STAZP CNTRL

RCV3: .DECZP CNTRL USE OF CYCLES COUNT IS TO COUNT 7 START BITS.
BNE RV3 12 LOOP IF ADDITIONAL BIT REMAIN.
LDAAB CRT LOAD DATA FROM CRT.
ANDIM /80 BLANK UNNECESSARY BITS.
ORAZP CHAR PLACE MSB OF ACC IN 'CHAR'.
LSR SHIFT ACC RIGHT ONE BIT.
STAZP CHAR PUT RESULT IN 'CHAR'.
DEY DEC BIT COUNTER (Y-REG).
BNE RV2 LOOP IF ADDITIONAL BITS REMAIN.
LDYIM /16 LOOP MUST WAIT FOR ONE PARITY AND ONE STOP BIT.
STYZP CNTRL

RCV4: .DECZP CNTRL USE OF CYCLES COUNT IS TO COUNT 7 START BITS.
BNE RV4 12 LOOP THROUGH PARITY & STOP DELAY.
LDAIM /FF
ERAZP CHAR COMPLEMENT THE CONTENTS OF 'CHAR'.
ANDIM /FF BLANK THE MSB.
STAZP CHAR STORE COMPLEMENTED RESULT IN 'CHAR'.
NOP WORD NOW OCCUPIES BITS 40-47.
NOP 'NOP' OPERATIONS COMPLETE TIMING.

RCV5: RTS RETURN.

ERR2: LDAZP /BO PLACE ERROR CODE 80-HEX IN ACC
STAZP CHAR AND IN 'CHAR'.
JMP RCVS JUMP TO RETURN CODE.
THE FOLLOWING SUBROUTINE TRANSMITS A 7-BIT CHAR (PREVIOUSLY LOADED INTO MEMORY LOCATION 'CHAR') AT A BAUD RATE OF 110 TO THE TTY FOR HARD-COPY RESULTS OF SYSTEM COMMUNICATIONS. IT IS ASSUMED THAT THE X- AND Y-REGISTERS ARE AVAILABLE FOR USE. NO MACRO DEFINITIONS WERE USED IN THIS ROUTINE TO ASSIST THE PROGRAMMER WITH TIMING CALCULATIONS.

XMT11:
LDAM /3F iINITIALIZE THE DDR FOR TTY OUTPUT.
STAAB DDR2B
LDAM /07 iINITIALIZE TRANSMITTED BIT TO A LOGICAL 0 (I.E. A HIGH VOLTAGE LEVEL).
STAA AB TTY iSHIFT CHAR LEFT & PLACE '0' IN LSB.
LDAM /06 iSET THE COUNTER 'BITS' TO XMT 8 BITS.
STAZP BITS

XMT12:
LDAA AB TTY iLOAD ACC WITH PREVIOUS XMT'D BITS.
ADCAIM /FE iBLANK THE LSB.
LSRZP CHAR iPLACE LOWEST BIT OF 'CHAR' IN CARRY.
ADCAIM /00 iMAKE NEW SUM: LSB IS SAME AS LOWEST 'CHAR' BIT.
STAA AB TTY iWRITE BYTE TO TTY OUTPUT LATCH.
JSR DELAY iWAIT FOR ONE BIT AT 110 BAUD.
DECZP BITS iDECREMENT THE BIT COUNTER.
BNE XMT12 iIF ANY BITS REMAIN, LOOP BACK.
LDAM /07 iSEND 1 PARITY & 2 STOP BITS, REQUIRING 2 JSR'S TO 'DELAY'.
STAA AB TTY
JSR DELAY
JSR DELAY
JSR DELAY
RTS iRETURN.

THE MEMORY AREA FOR THE 16 BYTES USED BY THE KIM-1 ROM ROUTINES IS NOW RESERVED.

KIM!
.BLKB /14

THE STACK POINTER 'SET' OPERATION IS NOW INCLUDED.

SETSP:
STYZP SP-L
STAZP SP-H
RTS

THE NEXT SECTION CONTAINS A 9.09 MILLISEC DELAY (ONE BIT DURATION @110 BAUD).

DELAY:
LDAM /08
STAA AB CNTRH

DELY1:
LDAM /0C
STAA AB CNTRL

DELY2:
DECZP CNTRL
BNE DELY2
DECZP CNTRH
BNE DELY1
RTS

164
THIS SUBROUTINE TRANSMITS A CHARACTER AT A RATE OF 9600 BAUD. IT IS ASSUMED THAT THE ACC & Y-REGISTERS ARE AVAILABLE. THE CHAR IS CALLED FROM LOCATION 'CHAR'. THE ACCUMULATOR IS USED DURING THE ROUTINE'S USE. NO MACRO DEFINITIONS WERE USED FOR EASE OF TIMING. THE NUMBERS ENTERED AT THE START OF EACH COMMENT REFER TO THE NUMBER OF MACHINE CYCLES REQUIRED TO EXECUTE THAT PARTICULAR MACHINE OPERATION. BRANCH INSTRUCTIONS VARY IN DURATION.

XMT95: STYZP CHAR 13 PUT DESIRED CHAR IN 'CHAR'.
XMT96: LDYIM /01 12 SET DATA DIRECTION REG TO DESIRED READ/WRITE MODE.
LDAZP CHAR 13 PUT ASCII CHAR INTO ACC.
EDRIM /FF 12 LOGICALLY INVERT THE CHAR.
LDYIM /08 12 SET 'BITS' TO COUNT ONE START BIT.
STYFP BITS 13 BIT LENGTH.
LDYIM /01 12 SET LSB OF PORT TO SIGNAL A START BIT.
XMT91: STYAB CRT 4 START BIT.
DECZP BITS 15 DECREMENT THE BIT COUNTER.
BNE XMT91 2 LOOP IF TIME REMAINS.
NOP 2 NOP'S COMPLETE THE TIMING.
LDYIM /08 12 USE Bits TO COUNT DATA BITS.
STYFP BITS 13
XMT92: LDYIM /OA 12 USE Y-REG AS TIMING COUNTER.
XMT93: STAAB CRT 4 STORE ASCII CHAR ON CRT PORT.
DEY 12 DEC THE TIMING COUNTER.
BNE XMT93 2 LOOP IF TIME REMAINS.
LSR 2 SHIFT HIGHER BIT INTO LSB.
DECAB BITS 6 DEC THE BIT COUNTER.
BNE XMT92 2 LOOP IF BITS REMAIN.
LDYIM /17 2 USE Y-REG AS TIMING COUNTER.
LDAIM /00 12 LOWER DATA LINE FOR PARITY & STOP BITS.
DEY 12 DEC THE TIMING COUNTER.
BNE XMT94 2/3 LOOP IF TIME REMAINS.
RTS 6 RETURN.

THE SCRATCHPAD AREA OF TEN BYTES IS NOW SAVED FOR USE BY THE PROGRAM WHEN A FILENAME MUST BE RECORDED.

FILNH: .BLKB /OA 1 SAVE ROOM FOR A FILENAME OF TEN BYTES.

THIS SUBROUTINE SAVES THE EXISTING STACK POINTER BY PLACING IT IN THE TEMPORARY STACK POINTER BUFFER. THIS BUFFER CONSISTS OF TWO BYTES ON THE ZERO PAGE: TSP-L AND TSP-H.

SAUSP: LDAZP SP-L
STA2P TSP-L
LDAZP SP-H
STA2P TSP-H
RTS

THE FOLLOWING SUBROUTINE RETURNS THE STORED STACK POINTER VALUES TO THE POINTER FROM THE TEMPORARY STORAGE BUFFERS TSP-L AND TSP-H.

RSTSP: LDAZP TSP-L
STA2P SP-L
LDAZP TSP-H
STA2P SP-H
THE FOLLOWING SUBROUTINE TRANSMITS A LINE-FEED CHARACTER TO THE CRT DISPLAY AND, SENSING WHETHER OR NOT THE CURSOR IS SUFFICIENTLY FAR DOWN THE SCREEN, A DELAY OF ABOUT 30 MILLISECONDS IS INCLUDED. THIS PERMITS THE SLOWER CRT LOGIC TO REFRESH THE SCREEN AND ROLL EXISTING CHARACTERS UPWARDS IF NECESSARY.

```
XMTLF: .XMT LF
JSR DELAY
JSR DELAY
JSR DELAY
RTS
```

FILN2: .BLKB /OA TEMPORARY FILENAME STORAGE FOR SEARCH USES.

```
MONITOR ROUTINE

THE FOLLOWING MAIN ROUTINE LOOPS THROUGH A POLLING SCHEME TO CHECK FOR ANY ATTEMPT AT DATA TRANSMISSION FROM THE CRT. IF DETECTED, THE LOOP JUMPS TO THE RECEIVE 9600 BAUD ROUTINE (RCV96) AND DECODES THE INCOMING BYTE.

THE INITIALIZATION ROUTINE IS SITUATED IN MEMORY BEGINNING AT ADDRESS 'INTIL' AS DEFINED UNDER 'MASTER SYSTEM UTILITIES'.

```
.ORG INTIL

THE NEXT ADDRESS IS SET TO THE DESIRED POSITION FOR THE MONITOR Routines.

```
```
THIS NEXT SECTION CONTAINS A POLLING LOOP THAT WATCHES THE CRT KEYBOARD FOR AN ATTEMPT TO TRANSMIT A CHARACTER. IF AN ATTEMPT IS MADE, THE DATA LINE CONNECTED TO THE MSB OF LOCATION 'CRT' GOES HIGH. THE PROGRAM THEN JUMPS TO A ROUTINE THAT DECIDES THE INCOMING CHARACTER (PROVIDED THAT THE START BIT IS OF CORRECT LENGTH).

PROMPT THE OPERATOR WITH A CLEAR SCREEN AND A PERIOD.

JUMP TO THE RECEIVE ROUTINE.

TEST FOR AN ERROR IN RECEPTION.

LOOK FOR A 'CR' FROM KEYPAD.

PUT LAST CHAR IN 'CHAR' 1 LOOK FOR A 'CR'.

RESET CURSOR TO NEXT LINE.

NOW BEGIN THE COMMAND CHARACTER DECODING ROUTINES.

IF A FILE-DELETE IS REQUESTED.

BRANCH AROUND IF NOT REQUESTED.

JUMP TO ROUTINE 'FILE'.

TEST FOR A DISK FORMAT OPERATION.

BRANCH AROUND IF NOT REQUESTED.

USE 'FORMT' TO FORMAT A DISKETTE.

TEST FOR A FILE-LOAD OPERATION.

BRANCH AROUND IF NO MATCH.

USE 'LOAD' TO GET A FILE FROM DISK.

TEST FOR A FILE PRINT CMMD.

BRANCH AROUND IF NOT REQUESTED.

USE 'PRINT' TO TYPE FILE ON TTY.

TEST FOR A DISK FILE RENAME REQUEST.

BRANCH AROUND IF NOT APPLICABLE.

USE 'RNAME' TO RENAME THE DISK FILE.

TEST FOR A SAVE-FILE OPERATION.

BRANCH AROUND IF NOT REQUESTED.

'ISTORE' PLACES THE STACK ON DISK.

TEST FOR A CMD TO LIST THE DISK TABLE.

BRANCH AROUND IF NOT REQUESTED.

USE THE DISPLAY RTN TO SHOW THE TABLE.

TEST FOR 'ENTER EDIT MODE' CMMD.

BRANCH AROUND IF NOT EQUAL.

PLACE SP POINTING AT TOP OF DATA STACK.

ESTABLISH THE LINE POSITION OF THE CURSOR.

CLEAR THE CRT SCREEN.

DELAY 30 MILLISECONDS FOR CRT TO CATCH UP.

PRINT 1ST PAGE OF DATA ON SCREEN.

JUMP TO THE EDITOR DECODING ROUTINE.

TEST FOR TYPEWRITER-MODE CMMD.

BRANCH TO A ROUTINE IF NOT EQUAL.

CLEAR THE CRT.

PLACE A CARRIAGE-RETURN IN 'CHAR'.

REPOSITION THE TTY PRINTING HEAD.

TEST FOR A FALSE DATA START.

LOOP BACK FOR A FALSE START.

IF PROGRAM RETURNS.
THE FOLLOWING ROUTINES CONTAIN THE REQUIRED SERVICE AND UTILITY FUNCTIONS FOR ALL DISK OPERATIONS SPECIFIED IN THE THESIS DEFINITION. THE CHOSEN DISK FORMAT IS AS FOLLOWS:

**TRACK 00**
- SECTOR 1: DISK IDENTIFICATION/SPECIAL MESSAGES
- SECTORS 2-26: BOOTSTRAP/MONITOR/EDITOR/ETC.

**TRACK 01**
- SECTOR 1: TRACK IDENTIFICATION
- SECTORS 2-25: DIRECTORY (192 FILEDATA SLOTS)
- SECTOR 26: FUTURE EXPANSION

**TRACK 02**
- SECTOR 1: TRACK IDENTIFICATION
- SECTORS 2,8,14,20: DIRECTORY (30 FILEDATA SLOTS)
- SECTOR 3: END OF FILEDATA MARKER
- SECTORS 13-26: FUTURE EXPANSION

**TRACKS 03-76**
- FILE STORAGE USING THE FOLLOWING ORGANIZATION:
  - 1024 BYTES PER FILE
  - THREE FILES PER TRACK
  - TWO EMPTY SECTORS PER TRACK
  - SECTOR 1: TRACK IDENTIFICATION
  - SECTOR 26: FUTURE EXPANSION

SECTOR SEQUENCING WAS CHOSEN TO AVOID LONG WAITS BETWEEN SECTOR POSITIONINGS UNDER THE DISK HEAD. THUS, FILE SECTORS WERE ALTERNATED AT LEAST SIX SECTORS APART. THE FILES OF ANY TRACK WILL BE FOUND IN THE FOLLOWING SECTOR SEQUENCE:

<table>
<thead>
<tr>
<th>FILE</th>
<th>SECTOR SEQUENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 8, 14, 20, 3, 9, 15, 21</td>
</tr>
<tr>
<td>2</td>
<td>4, 10, 16, 22, 5, 11, 17, 23</td>
</tr>
<tr>
<td>3</td>
<td>6, 12, 18, 24, 7, 13, 19, 25</td>
</tr>
</tbody>
</table>
'FILEDATA' IS A TERM THAT DENOTES THE ORGANIZATION OF INFORMATION ABOUT FILES IN THE DIRECTORY. THE CURRENT METHOD IS TO PLACE A FILENAME WITH TEN ASCII CHARACTERS IN THE FIRST TEN BYTES AND THEN RESERVE THE NEXT SIX BYTES FOR FUTURE EXPANSION. THE LOCATION OF A FILEDATA SLOT IN THE DIRECTORY DENOTES THE ABSOLUTE DISK ADDRESS OF THE FILE.

THE FOLLOWING ROUTINE PERMITS THE OPERATOR TO SELECT THE DESIRED DISK BEFORE ANY OPERATIONS ARE CONDUCTED.

```
DISKC:  LDXIM /00  FROM THE OPERATOR WITH 'DISK 4?'.
DISKO:  .MOVE AX,DX,ZP,CHAR  MESSAGE LOCATED AT 'OK'.
  .XMT96
  INX
  CPXIM /0B
  BNE DISKO

DISK1:  .RCV96  RECEIVE ASCII CHAR FROM OPERATOR.
    .CMPR IM,ERR,ZP,CHAR  TEST IF ASCII CHAR IS < ZERO.
    BNE DISK1  IGNORE AN INCORRECT CHOICE.
    .CMPR IM,32,CHAR  TEST IF ASCII CHAR IS > ONE.
    BNE DISK1  IGNORE AN INCORRECT CHOICE.
    LDAZP CHAR
    SEC
    LDAZP SECTR
    SBCIM 706
    CMPIM 71A  JIS THE RESULT GREATER THAN 26?
    BMI SCTR1  IF NOT, TO TO SCTR1.
    SEC  IF SO, SECTR = SECTR - 23
    SBCIM 717
    STAZP SECTR  STORE RESULT IN 'SECTR'  TO BE USED AS THE NEW SECTOR CHOICE.
    RTS
```

THIS UTILITY IS WRITTEN TO CONVERT THE ABSOLUTE FILE NUMBER OF A GIVEN FILE (FROM 1 TO 222) INTO A TRACK ADDRESS AND THE ABSOLUTE FILE NUMBER ON THE SPECIFIC TRACK (FROM 1 TO 3). INITIALLY, 'CNTR' MUST CONTAIN THE ABSOLUTE FILE NUMBER. THE LOCATIONS 'TRACK' AND 'SECTR' MUST BE AVAILABLE FOR USE. 'TRACK' WILL CONTAIN THE ABSOLUTE TRACK ADDRESS UPON RETURN, AND 'SECTR' WILL CONTAIN THE STARTING SECTOR NUMBER ON THAT TRACK.

```
TRK:    .CLEAR TRACK  SET 'TRACK' EQUAL TO ZERO.
TRK1:   CMPIM /04  TEST FOR A VALUE OF 3 OR LESS.
        BNE TRK2
    .CLRCH SEC
    SBCIM /03
    INCPZ TRACK
    JMP TRK1  LOOP UNTIL A REMAINDER OF <4 OCCURS.

TRK2:   .STAZP SECTR  REMAINDER (1,2, OR 3) IS PLACED IN 'SECTR'
    ASLPZ SECTR  % MULTIPLIED BY 2.
    INCPZ TRACK
    INCPZ TRACK
    RTS  RETURN.
```

THE FOLLOWING SECTION OF PROGRAM CODE PERFORMS AN ALGORITHMIC CHOICE FOR THE NEXT SECTOR TO BE OPERATED UPON IN THE SEQUENCING PATTERN USED FOR THIS THESIS. AN ALTERNATING SEQUENCE WAS EMPLOYED TO SAVE DISK ACCESS TIME.

```
SCTR:   LDAZP SECTR  SECTR = SECTR + 6
        CLC
    ADB9IM /06
    CMPIM /1A  IS THE RESULT GREATER THAN 26?
    BMI SCTR1  IF NOT, TO TO SCTR1.
    SEC  IF SO, SECTR = SECTR - 23
    SBCIM /17

SCTR1:  .STAZP SECTR  STORE RESULT IN 'SECTR' TO BE USED AS
        RTS  THE NEW SECTOR CHOICE & RETURN.
```
A TRACK SEEK OPERATION FOR EITHER DISK 40 OR 41 IS PERFORMED HERE.
THE PROGRAMMER MUST ASSUME THAT THE DESIRED TRACK IS NOTED IN
LOCATION 'TRACK', (I.E. THE NUMBER IS <=76), AND THAT THE CHosen
DISK IS ENTERED IN THE BUFFER LOCATION 'DISK'.

SEEK:
MOVE IM_/O1, ZP, CMD1 'PLACE THE 'SEEK' NUMERICAL COMMAND #01
IN THE LOCATION 'CMD1'.
MOVE ZP, TRACK, ZP, CMD2 'PLACE DESIRED TRACK IN #2 CMD BYTE.
DISK-CMD 'USE THE SUBROUTINE 'CMD' TO EXECUTE ALL
DESIREO DISK TASKS.
MOVE IM_/O7, ZP, CMD1 'REQUEST A 'STATUS' CMD.
CLEAR CMD2

SEEK1:
JSR DELAY 'WAIT 9 MILLISEC & AVOID LOADING FDC.
.DISK-CMD
.CMPR IM_/20, ZP, STATS 'TEST IF SECONDARY STATUS DENOTES A
BEQ SEEK1 'SEEK IN PROGRESS.
RTS 'RETURN.

THE FOLLOWING SUBROUTINE PREPARES TWO CONTROL BYTES AND WRITES
THEM TO THE FDC FOR IMPLEMENTATION OF OPERATIONS BY THE DISKS.
THE CHOICE OF DISK IS HELD IN 'DISK', THE COMMAND WORD WITH THE
2-BIT SUBCOMMANDS IS HELD IN THE MEMORY LOCATION 'CMD'. THE
SECOND COMMAND BYTE ALWAYS CONTAINS A SECTOR ADDRESS AND IS
KEPT IN THE MEMORY LOCATION 'SECTR'.

CMMD: JSR TSTST 'TEST FLOPPY CONTROLLER STATUS.
.CMPR IM_/04, ZP, MODE 'TEST IF FDC IS READY.
BEQ CMMD1 'IF BUSY, LOOP UNTIL DONE.
MOVE IM_/31, A#, NUM1 'SET ERROR CODE TO '1'.
MOVE IM_/36, A#, NUM2 'USE 'TSTST' TO PRINT ON
JMP TSTST6

CMMD1: MOVE IM_/FF, A#, DSK3A 'USE 'DSK2' TO WRITE FROM KIM.
LDAZP DISK 'LOAD ACC WITH DISK CHOICE AND
ASL ASL ASL 'SHIFT BITS 3 PLACES LEFT.
ASL ASL ASL
ORA2P CMD1 'JOIN DISK CHOICE & CMD BYTES.
STAB DSK2 'PLACE CMD BYTE ON DATA PORT.
.CMPR IM_/02, A#, DSK1 'RAISE 'CMD' LINE & XMT 1ST CMD BYTE.
CMMD2: LDAAB DSK3 'WAIT UNTIL THE 'XFR' LINE GOES HIGH
ANDIM /01 'FROM FDC.
CMPIM /01
BNE CMMD2
MOVE IM_/00, A#, DSK1 'LOWER 'CMD' LINE.
NOP 'WAIT 2 us FOR FDC SOFTWARE.
MOVE ZP, CMMD2, A#, DSK2 'PUT #2 CMD BYTE ON DATA PORT.
NOP 'WAIT 2 us FOR FDC SOFTWARE.
MOVE IM_/02, A#, DSK1 'RAISE 'CMD' LINE.
.CMPR IM_/NULL, ZP, CMD1 'TEST IF 'RESET' CMD WAS SENT.
CMMD6 'IF NOT SENT, BRANCH AROUND ROUTINE.

CMMD5: LDAAB DSK3 'WAIT UNTIL CONTROLLER IS RESET BY
CMPIM /04 'BY LOOPING UNTIL STATUS PORT FROM
BNE CMMD5 'FROM FDC/=04.
RTS 'RETURN.
CMMD6: LDAAB DSK3 'WAIT UNTIL 'XFR', 'BUSY', & 'DONE'
ANDIM /03 'ARE HIGH.
CMPIM /03
BNE CMMD6 'AGAIN TEST THE FDC FOR CMMD STATUS.
JSR TSTST 'AGAIN TEST THE FDC FOR CMMD STATUS.
RTS 'RETURN FROM SUBROUTINE.
THE FOLLOWING DISK UTILITY SUBROUTINE TESTS THE STATUS BYTE FOR ANY INCURRED ERRORS. IF DETECTED, THE APPROPRIATE ERROR CODE IS AUTOMATICALLY XMTTD TO THE CRT.

TSTST1: LDAAB DSK3
ANDIM /06
STAZP MODE
CMPIM /06
BED TSTST2
JSR CLDSK
RTS

TSTST2: MOVE IM/00+AB,DDR3A
MOVE AB,DSK2,ZP,STATS
LDAAB DSK3
ANDIM /20
CMPIM /20
BEQ TSTST3
JSR CLDSK
RTS

TSTST3: LDAAB DSK3
ANDIM /18
STAZP NUM1
MODE
CMPIM /IS
LSR
LSR
LSR
CLC
ADCIM /30
STAAB NUMI
MOVE AB,DSK2,ZP,CHARI
CLEAR CNTRL
CMPR IM,/FF,ZP,CHARI
BNE TSTST4
MOVE IM,/30+AB,NUM2
JMP TSTST6

TSTST4: CMPR AB,NUMI,IM,/33
IF THE TYPE OF ERROR IS '3', BLANK THE BNE TSTST6: LSR OF THE DATA PORT, SHIFT IT
LDAIM /FE
RIGHT ONCE, RETURN IF TO 'CHARI'.
ANDZF CHAR1
SEE FDC MANUAL FOR EXPLANATION OF THE LSR CHAR1
LSB IF THE STATUS TYPE = '3'

TSTST5: INCZF CNTRL
CMPR IM,/09,ZF,CNTRL
IF 'CNTRL'=/09, THE BYTE WAS CLEARED.
BEC TSTST5
WHEN THE 1ST SET BIT IS FOUND,
BCC TSTST4
LEAVE THIS LOOP.

TSTST6: LDAZF CNTRL
CLC
ADCIM /30
STAAB NUM2
JSR CLDSK

THE FOLLOWING IS A SHORT ERROR-PRINTING ROUTINE.
IT IS ASSUMED THAT THE FIRST ASCII NUMBER IS IN THE LOCATION 'CHAR' AND THE SECOND NUMBER IN 'CHAR1'.

TSTST7: LDXIM /FF
RESET THE 6502 STACK POINTER.
TXS
LDXIM /00
RESET THE Y-REG.

TSTST8: MOVE AX*EFLAG,ZF,CHAR
PRINT AN ERROR MESSAGE.
.XNTP6
INK
CPXIM /11
BNE TSTST7
JMP MON

THE FOLLOWING SEQUENCE OF DATA BYTE LOCATIONS CONTAIN PROMPTING MESSAGES WHICH ARE PRINTED AT THE CRT BY VARIOUS PORTIONS OF MACHINE CODE LOCATED THROUGHOUT THE LISTINGS.
EFLAG: .BYTE /2A 1\E'
      .BYTE /2A 1\E'
      .BYTE /2A 1\E'
      .BYTE /20 1\E'

EFLAG: .BYTE /45 1\E'
      .BYTE /52 1\R'
      .BYTE /52 1\R'
      .BYTE /4F 1\D'
      .BYTE /52 1\R'
      .BYTE /20 1\E'

NUM1: .BYTE /00 1\FIRST ERROR CODE NUMBER
      .BYTE /2E 1\E'

NUM2: .BYTE /00 1\SECOND ERROR CODE NUMBER
      .BYTE /20 1\E'
      .BYTE /2A 1\E'
      .BYTE /2A 1\E'

FN:  .BYTE /44 1\E'
     BYTE /44 1\E'
      .BYTE /45 1\E'
      .BYTE /4E 1\N'
      .BYTE /41 1\A'
      .BYTE /40 1\M'
      .BYTE /45 1\E'
      .BYTE /3F 1\E'
      .BYTE /20 1\E'

OLD: .BYTE /4F 1\D'
      .BYTE /4C 1\E'
      .BYTE /44 1\D'
      .BYTE /20 1\E'
      .BYTE /46 1\E'
      .BYTE /4E 1\N'
      .BYTE /3F 1\E'
      .BYTE /20 1\E'

NEW: .BYTE /4E 1\N'
      .BYTE /45 1\E'
      .BYTE /57 1\W'
      .BYTE /20 1\E'
      .BYTE /46 1\E'
      .BYTE /4E 1\N'
      .BYTE /3F 1\E'
      .BYTE /20 1\E'

DR:  .BYTE /44 1\D'
      .BYTE /49 1\I'
      .BYTE /53 1\E'
      .BYTE /4B 1\K'
      .BYTE /20 1\E'
      .BYTE /23 1\E'
      .BYTE /3F 1\E'
      .BYTE /20 1\E'

TRCK: .BYTE /5A 1\T'
       .BYTE /52 1\R'
       .BYTE /41 1\A'
       .BYTE /43 1\C'
       .BYTE /4B 1\K'
       .BYTE /3F 1\E'
       .BYTE /20 1\E'

SCR:  .BYTE /53 1\S'
      .BYTE /45 1\E'
      .BYTE /43 1\C'
      .BYTE /54 1\E'
      .BYTE /4F 1\D'
      .BYTE /52 1\R'
      .BYTE /3F 1\E'
      .BYTE /20 1\E'

RDWT: .BYTE /52 1\R'
       .BYTE /44 1\D'
       .BYTE /20 1\E'
       .BYTE /4F 1\D'
       .BYTE /52 1\R'
       .BYTE /20 1\E'
       .BYTE /57 1\W'
       .BYTE /54 1\E'
       .BYTE /3F 1\E'
       .BYTE /20 1\E'
THIS IS A SHORT ROUTINE THAT IS USED TO RESET THE FDC AFTER A SECONDARY STATUS BYTE WAS READ FROM 'DSK2'. THE COMMAND LINE 'ACK' MUST BE RAISED UNTIL THE THE 'XFR' LINE OF THE STATUS BYTE RISES. 'ACK' IS THEN LOWERED AND CONTROL PASSES TO THE ORIGINAL PROGRAM.

THE NEXT UTILITY ROUTINE SEARCHES THE TWO TRACKS #0 & #1 OF A PREVIOUSLY SELECTED DISK FOR A DESIRED FILENAME. WHEN DETECTED, THE X-REG WILL POINT AS AN OFFSET IN 'BUFFR' TO THE STARTING CHARACTER OF THE FILENAME, AND THE Y-REG WILL CONTAIN /OA. Thus the following characteristics apply to the return from subroutine:

X-REG POINTS AS OFFSET TO START OF FILENAME IN 'BUFFR'.
Y-REG CONTAINS /00.
TRACK CONTAINS DIRECTORY TRACK NO.
SECTR CONTAINS DIRECTORY SECTR NO.

SRCH1: .MOVE IM, /00, AB, FCNTR FUSE 'FCNTR' TO COUNT THE SEG. FILE NO.
     .MOVE IM, /01, ZP, TRACK FUSE 'TRACK' TO POSITION THE R/W DISK HEAD.
     .MOVE IM, /02, ZP, SECTR INITIALIZE SECTOR COUNTER 'SECTR'.
     .MOVE IM, /10, AB, SCCTC INITIALIZE SECTOR COUNTER 'SCCTC'.

SRCH2: INCAB FCNTR
       LDYM /09
       .CMPR AX, BUFR, AY, FILM COMPARE TOP OF BUFFER WITH TOP OF FILENAME.
       JBE SRCH6 IF EQUAL, BEGIN TEST RTN AT 'SRCH6'.
       LDIM /7F
       .READ READ A SECTOR FROM DISK.

SRCH3: LDXIM /7F
       INC-SECTOR
       INCZP TRACK CHOOSE THE NEXT TRACK AND LOOP.
       JMP SRCHO

BEGIN A SEQUENTIAL MATCHING OF TEN FILENAME CHARACTERS ON DISK TO A DESIRED FILENAME IN 'FILM'.

SRCH6: STZF BITS TEMPORARILY STORE X-REG IN 'BITS'.

END:
THIS SUBROUTINE PERFORMS TWO FUNCTIONS: THE FIRST IS A DISK READ INTO A 128-BYTE BUFFER ON THE FDC. THE SECTOR IS DENOTED IN THE MEMORY LOCATION 'SECTR' PRIOR TO ENTRY OF THE ROUTINE. A DISK SEEK IS ASSUMED TO HAVE BEEN EXECUTED. THE SECOND FUNCTION READS THE FDC BUFFER CONTENTS, ONE BYTE AT A TIME, THROUGH THE PORT 'DSK2' A READY-READY HANDSHAKE BASIS. SINCE THIS SUBROUTINE USES THE 'PUSH' MACRO, IT IS ASSUMED THAT THE STACK POINTER IS CORRECTLY POSITIONED.

READ: .MOVE IM,/FF,AB,DDR3A
       .MOVE IM,/42,ZP,CMD1  'DISK-TO-BUFFER' CMD TO FDC.
       .MOVE ZP,SECTR,ZP,CMD2 ESTABLISH DESIRED SECTOR.
       .DISK-CMMD
       .MOVE IM,/B2,ZP,CMD1 'BUFFER-TO-HOST' READ CMMD.
       .CLEAR CMD2
       .DISK-CMMD
       .MOVE IM,/B0,ZP,LINE USE 'LINE' TO COUNT 128 BYTES.
       .MOVE IM,/00,AB,DDR3A USE THE DATA PORT TO READ FROM FDC.

READ1: .MOVE AB,DSK2,ZP,CHAR 'PUT DATA BYTE INTO 'CHAR'.
       .PUSH 'CHAR' ONTO STACK.
       .MOVE IM,/01,AB,DSK1 RAISE 'ACK' LINE TO FDC.
       .MOVE IM,/00,AB,DSK1
       .DECZP LINE
       BNE READ1
       JSR TSTST

READ3: RTS RETURN FROM SUBROUTINE.

WRITE: .MOVE IM,/84,ZP,CMD1 CHOOSE A 'HOST-TO-BUFFER' WRITE CMMD.
       .CLEAR CMD2 'CMD2' NOT USED, BUT CLEAR IT ANYWAY.
       .DISK-CMMD GIVE COMMAND TO FDC.
       .MOVE IM,/B0,ZP,LINE USE 'LINE' TO COUNT 128 BYTES.
       .MOVE IM,/FF,AB,DDR3A USE 'DDR3A' TO WRITE TO FDC.

WRITE1: .DOWN FETCH NEXT BYTE FROM STACK POINTER.
       .MOVE ZP,CHAR,AB,DSK2 WRITE BYTE TO FDC.
       .MOVE IM,/01,AB,DSK1 'TELL FDC DATA IS READY.'
       .MOVE IM,/00,AB,DSK1 RETURN 'ACK' TO LOW STATE.
       .DECZP LINE DECREMENT BYTE COUNTER.
       BNE WRITE1 'LOOP IF 128 BYTES HAVE NOT BEEN SENT.
       JSR TSTST 'CHECK FDC STATUS AT END OF EXCHANGE.'

WRITE2: .MOVE IM,/41,ZP,CMD1 'BUFFER-TO-DISK WRITE COMMAND.'
       .MOVE ZP,SECTR,ZP,CMD2 'PLACE CHOICE OF SECTOR IN 2ND CMMD BYTE.'
       .DISK-CMMD
       RTS RETURN.
THE FOLLOWING SOURCE CODE AUTOMATICALLY FORMATS A CHOSEN DISK
FOR SOFT-SECTORED DATA STORAGE. TRACKS 00 THRU 02 ARE WRITTEN
OVER WITH NULL CHARACTERS. THE SECTOR AT TRACK 02, SECTOR 12
IS USED AS AN END OF DIRECTORY MARKER. ALL BITS OF THIS SECTOR
ARE SET.

FORM1: .CLEAR TRACK ;INITIALIZE TRACK COUNTER TO ZERO.
FORM2: .SEEK ;POSITION DISK HEAD.
FORM3: .MOVE IM/06,ZP,CM01 ;WRITE A 'FORMAT' COMMAND TO FDC.
.MOVE ZP,TRACK,ZP,CM02 ;TELL THE FDC WHAT THE TRACK IS.
.DISK-CMDH ;CHANGE THE TRACK CHOICE.
.INCZP TRACK ;CHANGE THE TRACK CHOICE.
.CMPR IM/4D,ZP,TRACK ;TEST IF 77TH TRACK IS REACHED.
.BNE FORM1 ;LOOP IF ALL 77 TRACKS HAVE NOT BEEN FORMATTED.
; .RESET ;REPOSITION DISK HEAD ON TRACK 00.
.FORM4: .MOVE IM,06,AX,BUFFR ;ILOGIC LOOP TO CLEAR 'BUFFR' WITH
.INX ; NULLS.
.CPXIM /00 ;TEST IF TOP OF 'BUFFR' HAS BEEN REACHED.
.BNE FORM4 ;START IF NOT.
; .CLEAR TRACK ;START WRITING ON TRACK 00.
.FORM5: .MOVE IM/02,ZP,SECTR ;START AT THE FIRST SECTOR.
.SEEK ;POSITION THE R/W DISK HEAD OVER THE
.DECAB SCTC ;DESIRED TRACK. SET THE SP.
.WRITE ;WRITE NULLS ONTO THE DISK.
.INC-SECTOR
.DECAB SCTC ;LOOP IF NOT.
; INCZP TRACK ;CHOOSE THE NEXT TRACK.
.CMPR IM/03,ZP,TRACK ;STOP IF 3 TRACKS HAVE BEEN WRITTEN.
.BNE FORM5
.DECZP TRACK ;REPOSITION THE STACK POINTER.
.WRITE ;WRITE ONTO THE DISK.
.FORM6: .MOVE IM/FF,Ax,BUFFR ;SET UP 'BUFFR' WITH ALL BITS SET.
.INX
.CPXIM /00
.BNE FORM7 ;TEST IF TOP OF 'BUFFR' HAS BEEN REACHED.
; .CLEAR TRACK ;BEGIN WRITE ON TRACK 00.
.FORM7: .MOVE IM/01,ZP,TRACK ;BEGIN DIRECTORY READING ON TRACK 01.
.SEEK ;POSITION DISK HEAD.
.MOVE IM/18,AB,SCTC ;BEGIN DATA READING FROM SECTOR 02.
.FORM8: .RTS ;RETURN.
; THIS ROUTINE FETCHES THE DESIRED DISK FROM I/O AND DISPLAYS
; THE CONTENTS OF TRACKS 02 1 IN COLUMN FORM. THESE TRACKS:
; CONTAIN THE DIRECTORY FOR THAT PARTICULAR DISK.
.DC1: .LDXIM /00 ;CLEAR THE X-REG AND TRANSMIT THE
.DC2: .MOVE AX,ENABLE,ZP,CHAR ; PROMPTING MESSAGE "ENABLE PTR?"
.XMT96 ; TO THE CRT.
.INX
.CPXIM /0C
.BNE DC2 ;receive a 'P' TO ENABLE TTY PRINTING.
.DC0: .Rcv96 ;FUSE 'PTREN' TO GOVERN TTY PRINTER OUTPUT.
.CMPR IM/ERR,ZP,CHAR ;AX,DISABLE,ZP,CHAR ;BEGIN DIRECTORY READING ON TRACK 01.
.BEQ DCO
.STAAB PTREN ;BEGIN DATA READING FROM SECTOR 02.
.XMT96
.XMTLF
.XMT CR
.DC1: .SEEK ;POSITION DISK HEAD.
.MOVE IM/18,AB,SCTC ;BEGIN DATA READING FROM SECTOR 02.
.DC2: .SP-SET BU-FH,BU-FL ;POSITION SP TO TOP OF 128-BYTE BUFFER.
.DC3: .READ ;FUSE 'READ' SUBRTN TO FETCH 128 BYTES.
.LDXIM /7F ;FUSE 'XREG AS BUFFER OFFSET POINTER.
.DC4: .MOVE IM/0A,ZP,TLINE ;FUSER 'TLINE' TO COUNT CHAR PER FNAME.
.DCS: .LDAAX BUFFR ;FETCH NEXT CHAR IN 'BUFFR'.
.CMPIM NULL
.BNE DC6
; DEC X-REG BY SIXTEEN TO SKIP CURRENT FILEDATA SLOT.
DC6: LOADAX BUFFR  ; FETCH NEXT SEQUENTIAL CHAR FROM 'BUFFR'.
CMPIM /FF  ; TEST IF DISPLAY ROUTINE IS AT END
BEC DCA  ; IF OF DIRECTORY.
STAZF CHAR  ; MOVE BYTE INTO 'CHAR'.
.XMT96  ; XMT CHAR TO CRT.
.CMFR IM+P,AB,PTREN  ; TEST IF TTY IS ENABLED.
BNE DC7
.XMT11
DC7: DEX  ; DECREMENT THE OFFSET POINTER.
DECZF TLINE  ; DECREMENT THE FNAME COUNTER.
.Speed DC6  ; ILOOP UNTIL FNAME IS COMPLETE.
;
DC8: TXA
CLD
SEC
SBCIM /06  ; SUB 6 FROM OFFSET POINTER- NOW
.TAX  ; POINTS AT 1ST CHAR OF NEXT FNAME.
.XMTLF  ; START A NEW LINE ON CRT.
.XR
.CMFR IM+P,AB,PTREN  ; SKIP AROUND TTY PRINT ROUTINE IF THE
BNE DC9  ; VARIABLE 'PTREN' ISN'T A 'P'.
.MOVE IM+CR,2F,CHAR  ; PRINT THE CHARACTER ON THE TTY.
.XMT11
.MOVE IM+LF,2F,CHAR
.XMT11
DC9: CPIXM /FF  ; TEST IF X-REG POINTS TO END OF 'BUFFR'.
BNE DC4  ; BRANCH IF NOT.
.INC-SECTOR  ; DETERMINE BY ALGORITHM THE NEXT SEQUENTIAL
DECAB SCTCT  ; SECTOR.
BNE DC3
.INCP TRACK
JMP DC1  ; ILOOP UNTIL THE 'END-OF-DIRECTORY' SECTOR

DCA: RTS  ; IS FOUND. RETURN.
;
PTREN: .BYTE '/00  ; TTY PRINTER ENABLE STATUS BYTE.
;
;
;
;
;
THE NEXT UTILITY ROUTINE LOADS A SELECTED FILE OF 1024 BYTES
FROM A PREVIOUSLY CHOSEN DISK INTO THE KIM STACK. UPON
COMPLETION, THE ACKNOWLEDGEMENT IS AN ASTERISK SENT TO THE CRT.

LOAD: LCDXM /00  ; CLEAR THE X-REG.
LOAD1: .MOVE AX,PN,2F,CHAR  ; TRANSMIT A PROMPTING MESSAGE TO
.XMT96  ; OPERATOR.
.INX
.CPIXM /0A
BNE LOAD1

.GETNAME  ; USE 'GTNAM' TO FETCH FILENAME FROM CRT.
.SEARCH/TBL  ; 'SEARCH/TBL' MUST PUT /DF INTO 'CNTR'.
.CMFR IM+5A1,AB,FCNTR  ; AS AN ERROR FLAG IF FILE NOT FOUND.
BNE LOAD2
.MOVE IM+3A,AB,NUM1  ; PLACE ERROR CODE '4.0' IN 'NUM1'.
.MOVE IM+30,AB,NUM2  ; 'NUM2'.
.JMP TST56  ; GOTO TO AN ERROR PRINTING RTN.

LOAD2: JSR TRK  ; TRANSLATE 'CNTR' WITH ABSOLUTE FILE NO
LOAD3: .SP-SET TOS-H,TOS-L  ; INTO TRACK ADDR AND 2ND,4TH, OR 6TH SECTOR
.SEEK  ; POSITION OF THE GIVEN TRACK.

LOAD4: .READ  ; READ THE DISK AT DESERVED SECTOR & TRACK.
.INC-SECTOR

DECR AB,SCTCT
BNE LOAD4

LOAD5: .SP-SET TOS-H,TOS-L  ; RESET THE SP TO THE TOP OF THE STACK.
.RESET  ; RESET THE FDC & DRIVE.
RTS  ; RETURN.
THIS ROUTINE TAKES THE STACK AND STORES IT ON A PREVIOUSLY
SELECTED DISK WITH A FILENAME CHOOSEN BELOW BY THE OPERATOR.

STORE: LDXIM /00  ICLEAR THE X-REG.
STOR1: .MOVE AX,FN+ZF,CHAR  IPROMPT THE OPERATOR WITH "FN?".
        LXT96
        CPXIM /0A
        BNE STOR1
        GETNAME  IUSE 'GTNAM' TO FETCH THE FILENAME
        SEARCH/TBL  ISEARCH FROM THE CRT, THEN SEARCH THE
        CMPR IM+/ZP,FCNTR  IDISC DIRECTORY FOR A MATCH.
        BNE STOR5  IF A MATCH IS FOUND, GO TO 'STORS'.
        LDXIM /09  IUSE THE X-REG TO COUNT TEN CHARACTERS.
STOR2: .MOVE AX,FILEM+AX,FILEM  ISTORE DESIRED FILENAME TEMPORARILY
        .MOVE IM+NULL+AX,FILEM  IN 'FILEM'. WRITE NULLS INTO OLD
        DEX  IFILENAME LOCATION.
        BPL STOR2  IF NINE CHARACTERS WERE TRANSFERRED.
        SEARCH/TBL  ISEARCH DIRECTORY FOR AN EMPTY FILENAME
        CMPR IM+ZP,FCNTR  ILOCATION. IF NOT AVAILABLE, RETURN
        BNE STOR3  WITH AN ERROR '4.1' MESSAGE.
        MOVE IM+ZP+1,AX,NUL  IWRITE NULLS OVER DESIRED FILENAME
        BNE STOR2  IF NOT AVAILABLE, RETURN.
        JMP TSTS6  IJUMP TO THE ERROR PRINTING ROUTINE.
STOR3: LDXIM /09  ITRANSFER THE DESIRED FILENAME FROM
        STOR4: LDAAY FILM2  'FILM2' TO THE APPROPRIATE LOCATION
        STAAX BUFFR  IN 'BUFFR'.
        DEX  
        DEY  
        BPL STOR4  IF NINE CHARACTERS WERE TRANSFERRED, WRITE 'BUFFR' BACK
        WRITE  IONTO DISC.
STOR5: .SP-SET TOS-H,TOS-L  IREPOSITION THE SP TO THE TOP OF THE FILE.
        JSR TRK  ICONVERT 'FCNTR' TO 'TRACK' AND 'SECTR'.
        .SEEK  IPOSITION THE DISC READ/WRITE HEAD.
        MOVE IM+ZP,FCNTR  USE Y-REG TO COUNT 8 SECTORS.
STOR6: .WRITE  IWRITE A SECTOR OF 128 BYTES ONTO DISC.
        INC-SECTOR IDETERMINE THE NEXT SEQUENTIAL SECTOR.
        DECZP CNTR  
        BNE STOR6  IF ALL 8 SECTORS HAVEN'T BEEN
        .SP-SET TOS-H,TOS-L  ISTORED, THEN RESET THE SP.
RTS  IRETURN.

THE FOLLOWING SERVICE ROUTINE KILLS A FILE ON A SELECTED
FLOPPY DISK. THE OPERATOR CALLS THIS FUNCTION WHILE IN
COMMAND MODE AND AFTER PROMPTING WITH A QUESTION MARK, TYPES
THE DESIRED FILENAME THAT MUST BE DELETED.

KFILE: LDXIM /00  ICLEAR THE X-REG.
KFILE0: .MOVE AX,FN+ZF,CHAR  IPROMPT THE OPERATOR WITH "FILENAME?".
        XXT96
        CPXIM /0A
        BNE KFILE0
        GETNAME  IUSE 'GTNAM' TO FETCH FILENAME FROM CRT.
        SEARCH/TBL  ISEARCH DISC DIRECTORY FOR FILE.
        CMPR IM+ZP,FCNTR  IF NOT FOUND, RETURN WITH ERROR MESSAGE.
        BNE KFILE2  USE Y-REG TO COUNT TEN CHARACTERS.
        LDXIM /0A  ILEAVE "FILENAME" ON STACK.
KFILE1: .MOVE IM+NULL+AX,BUFFR  IWRITE NULLS OVER DESIRED FILENAME
        DEX  IN 'BUFFR'.
        DEY  
        BNE KFILE1  
        .SP-SET BUF-H,BUF-L  IRESET THE SP TO THE TOP OF 'BUFFR'.
        .WRITE  IRETURN 'BUFFR' TO DISC DIRECTORY.
        RTS  IRETURN FROM SUBROUTINE.
KFILE2: .MOVE IM+ZP,FCNTR  ISET ERROR '4.0' IN ERROR DATA BYTES.
        MOVE IM+ZP+1,AX,NUL  IWRITE NULLS OVER DESIRED FILENAME
        JMP TSTS6  IJUMP TO ERROR PRINTING ROUTINE.
THIS SERVICE ROUTINE RENAMES A FILE ON A CHOSEN DISK. THE OPERATOR IS PROMPTED WITH A QUESTION MARK, AND THEN MUST ENTER THE ORIGINAL FILENAME. IF NO NAME IS FOUND IDENTICAL TO THE NAME REQUESTED, AN ERROR CODE APPEARS. IF FOUND, THE OPERATOR IS AGAIN PROMPTED, THIS TIME WITH "R?". THE NEW FILENAME IS THEN ENTERED.

RNAM0: LDXIM /00 CLEAR THE X-REG. INX CPXIM /08 BNE RNAM0

.RNAME .MOVE AX,OLD,ZP,CHAR PROMPT THE OPERATOR WITH "OLD FN?".

.RNAME .XMT96 INX CPXIM /08 BNE RNAM0

.GETNAME USE 'GTNAM' TO FETCH THE DESIRED OLD FILENAME & SEARCH THE DIRECTORY.

.CMPR IM,E1,AB,FCNT11 IF NOT FOUND, PRINT AN ERROR.

.BLK RNAM0 STXAB RNAMX STORE THE X-REG OFFSET IN 'RNAMX'.

.LDXIM /00 CLEAR THE X-REG.

.RNAME .MOVE AX,NEW,ZP,CHAR PROMPT THE OPERATOR WITH "NEW FN?".

.RNAME .XMT96 INX CPXIM /08 BNE RNAM0

.GETNAME USE 'GTNAM' TO FETCH THE NEW FILENAME.

.LDXAB RNAMX RESET THE X-REG.

.LDYIM /09 USE THE Y-REG TO COUNT TEN CHARACTERS.

.RNAME .MOVE AX,FILNM,AX,BUFFR TRANSFER THE NEW FILENAME INTO THE DEX OLD FILENAME POSITION.

.DEX BPL RNAM2 LOOP UNTIL TEN CHARACTERS ARE CHANGED.

.SP-SET BUF-H,BUF-L SET SP INTO TOP OF 'BUFFR'.

.WRITE WRITE 'BUFFR' BACK ONTO DISC.

.RNAME3: RTS RETURN.

.RNAMEX: .BYTE /00 IX-REG OFFSET STORAGE.

.GTNAM: LDXIM /09 USE Y-REG TO COUNT THROUGH TEN-CHAR NAME.

.LDAIM BLANK PUT AN ASCII 'BLANK' INTO ACC.

.GTNAM1: STAXAB FILNM SET 'FILNM' TO CONTAIN ALL BLANKS.

.BEX BPL GTNAM2 LOOP IF 'FILNM' NOT COMPLETELY FILLED.

.LDXIM /09 USE X-REG AS 'FILNM' OFFSET POINTER.

.GTNAM2: .RCV96 RECEIVE CHAR FROM CRT.

.CMPR IM,ERR,ZP,CHAR BEQ GTNAM2

.CMPR IM,'D',ZP,CHAR ITEST FOR A DELETE FUNCTION. BEQ GTNAM4 IF TRUE, TRANSFER TO DELETE ROUTINE.

.CMPR IM,'C',ZP,CHAR ITEST IF TRANSMITTED CHAR IS A 'CR'. BNE GTNAM3 IF NOT, ADD CHAR TO 'FILNM' BUFFER.

.XMTLF IIF TRUE, TERMINATE THE FILENAME FETCH.

.RTS RETURN.

.GTNAM3: CPXIM /FF ITEST IF X-REG HAS EXCEEDED 'FILNM' BOUNDS. BNE GTNAM2

.MOVE ZP,CHAR,AX,FILNM IPUT 'CHAR' INTO 'FILNM' POSITION.

.DEX DECREMENT OFFSET POINTER X-REG.

.XMT96 ECHO CHARACTER ON CRT.

.HBP GTNAM2 RETURN TO RECEPTION ROUTINE.

.GTNAM4: CPXIM /09 IIF X-REG = /09, NOTHING LEFT TO DELETE.

.BNE GTNAM2 IIF TRUE, RETURN TO RECEPTION ROUTINE.

.INX REPOSITION OFFSET POINTER.

.MOVE IM,BLANK,AX,FILNM IPUT A BLANK INTO PREVIOUS 'FILNM' SPOT.

.XMT ERASE IRESET THE CRT OUTPUT.

.JMP GTNAM2 RETURN TO CHARACTER RECEPTION ROUTINE.
THE FOLLOWING ROUTINE PERMITS AN OPERATOR TO ACCESS ANY DESIRED ABSOLUTE TRACK AND SECTOR ADDRESS ON A GIVEN DISC. THE DATA IS TRANSFERRED BY SECTOR BLOCKS OF 128 BYTES AT A TIME.

ABSL1: LDXIM /00 ; CLEAR THE X-REG.
ABSL1: .MOVE AX,SCR,ZP,CHAR ; PROMPT THE OPERATOR WITH "SECTOR?".
INX
CPXIM /08
BNE ABSL1 ; LOOP UNTIL THE MESSAGE IS TRANSMITTED.

ABSL2: .MOVE IM,FF,ZP,MODE ; USE THE X-REG TO COUNT
ABSL2: /0C ; USE 'MODE' TO COUNT 2 RECEIVED DIGITS.
ABSL2: .RCV96
ABSL2: .CMPR IM,ERR,ZP,CHAR
ABSL2: BEQ ABSL3
ABSL2: CMLH2 ZP,CHAR ; CONVERGE THE CHARACTER FROM ASCII TO
ABSL2: .CLD ; BINARY FORMAT.
ABSL2: .SEC
ABSL2: .SBCIM /30
ABSL2: .STAAP CHAR ; STORE BINARY CHARACTER IN 'CHAR'.
ABSL2: INCZP ; OF THE 2-DIGIT DECIMAL NUMBER AT THE
ABSL2: SEC
ABSL2: SEC
ABSL2: SBCIM ; END OF THIS LOOP.
ABSL2: CLD
ABSL2: .DEY
ABSL2: .JMP ABSL5

ABSL3: .STAA X CNTR ; STORE BINARY RESULT AT (CNTR + X-REG).
INX
.XHTL
.XIM CR
CPXIM /06
BNE ABSL7 ; AND WE CONTINUE WITH READ/WRITE CHOICE.

ABSL4: LDXAP CHAR1 ; TEN IS ADDED TO THE ACCUMULATOR FOR
ABSL4: BEQ ABSL6 ; EACH SINGLE COUNT OF 'CHAR1'. THUS,
ABSL4: .CLD ; THE ACC CONTAINS THE BINARY EQUIVALENT
ABSL4: .SEC ; OF THE 2-DIGIT DECIMAL NUMBER AT THE
ABSL4: .SBCIM /30 ; END OF THIS LOOP.
ABSL4: .DEY
ABSL4: .JMP ABSL5

ABSL6: STAAX CNTR ; STORE BINARY RESULT AT (CNTR + X-REG).
INX
.XHTL
.XIM CR
CPXIM /07
BNE ABSL3 ; WHEN DONE PROMPTING, RESET THE X-REG.

ABSL7: .MOVE AX,SCR,ZP,CHAR ; PROMPT THE OPERATOR WITH "TRACK?".
.XMT 96
INX
CPXIM /08
BNE ABSL8 ; CONTINUE TO LOOP IF MORE CHAR REMAIN.

ABSL8: .MOVE AX,SCR,ZP,CHAR
.XMT 96
INX
CPXIM /0A
BNE ABSL8 ; PROMPT THE OPERATOR WITH "READ OR WRITE?"

ABSL9: .RCV96
ABSL9: .CMPR IM,ERR,ZP,CHAR
BEQ ABSL9
.XMT 96
.LDXAP CHAR ; ECHO THE CHARACTER TO THE CRT.
CPXIM /0C
BEQ ABSL9
.XMT 96
.RCFI ; RETURN TO MONITOR VIA 'ASBL9'.
.XMT 96
.RCFI ; TEST FOR A READ OPERATION.
HERE, A READ OPERATION WAS DECODED, AND A SECTOR READ OF 128 BYTES IS PERFORMED. DATA IS ENTERED INTO THE STACK FROM THE CURRENT SP LOCATION ONWARDS.

**ABSLA:**

- **SEEK**: POSITION THE DISC READ/WRITE HEAD.
- **READ**: READ 128 BYTES FROM DISC.
- **MOVE**: IM, EOT, ZP, CHAR
  - ATTACH AN 'END-OF-TEXT' CHAR ONTO.
- **PUSH**: THE END OF THE DATA.

**ABSLB:**

- **RTS**: RETURN.

---

**EDITOR PACKAGE**

THIS NEXT SECTION CONTAINS A LOGIC LOOP THAT WATCHES THE CRT KEYBOARD FOR AN INCOMING CHARACTER. DECODING THEN Follows PROVIDED THAT NO ERROR WAS ENCOUNTERED DURING RECEPTION BY INSTRUCTION '*RCV96*'.

**EDIT**: .RCV96

- **LDAZP**: CHAR
- **ANDIM**: /EO
- **REQ**: EDIT1
- **LIAZP**: CHAR
- **CMPIM**: ^D
- **REQ**: EDIT1
- **JSR**: INSTR
- **JSR**: RSTSC
- **JMP**: EDIT

**EDIT1**: .CMPR

- **ZP, CHAR, IM, C-UP**: TEST FOR CURSOR-UP COMMAND.
- **BNE**: EDIT2
- **JSR**: CSRU
- **JMP**: EDIT

**EDIT2**: .CMPR

- **ZP, CHAR, IM, C-DN**: TEST FOR CURSOR-DOWN COMMAND.
- **BNE**: EDIT3
- **JSR**: CSRD
- **JMP**: EDIT

**EDIT3**: .CMPR

- **ZP, CHAR, IM, C-RIGHT**: TEST FOR CURSOR-RIGHT COMMAND.
- **BNE**: EDIT4
- **JSR**: CSR
- **JMP**: EDIT

**EDIT4**: .CMPR

- **ZP, CHAR, IM, C-LIGHT**: TEST FOR CURSOR-LEFT COMMAND.
- **BNE**: EDIT5
- **JSR**: CSRBL
- **JMP**: EDIT

**EDIT5**: .CMPR

- **ZP, CHAR, IM, CHR**: TEST FOR CURSOR-RIGHT(X5) CMD.
- **BNE**: EDIT6
- **JSR**: CSR8R5
- **JMP**: EDIT

**EDIT6**: .CMPR

- **ZP, CHAR, IM, CLR**: TEST FOR CURSOR-LEFT(X5) CMD.
- **BNE**: EDIT7
- **JSR**: CSRBL5
- **JMP**: EDIT
EDIT7: .CMPR ZP,CHAR,IM,CR
  BNE EDITA   ; Branch around if a CR wasn't sent.
  .XMT ERASE   ; Clear the CRT from the cursor onwards.
  .XMLF LINE   ; Send a LF & increment the line count.
  INCPZ LINE   ; Test if cursor was at bottom of screen.
EDIT8: .MOVE IM,CR,ZP,CHAR
  JSR INSRT   ; Insert a CR into the stack.
EDIT9: .CRT-RST
  JMP EDIT   ; Reset the screen.
  .CMPR IM,CR,IM,CR
  BNE EDITA   ; Test for a CHAR-DELETE command.
  BNE EDITC   ; Loop around if not equal.
  JMP EDIT    ; Jump to delete routine.
EDIT10: .CMPR IM,CR,CR
  BNE EDITC   ; Command, branch around if not.
  JMP EDIT    ; Jump to the 'ERASE-STACK' routine.
EDITC: .CMPR IM,CR,IM,CR
  BNE EDITC   ; Test if EDIT-MODE is to be cancelled.
  BNE EDITD   ; Branch around otherwise.
  JMP EDITS   ; End of all legal EDIT-MODE commands.
  JMP EDIT    ; Other characters will be ignored.

CURSOR
UTILITY SUBROUTINES

RELOCATE THIS SECTION OF CODE ON THE EPROM'S OF THE EXPANDED I/O
BOARD OF THE KIM-1 PROTOTYPING UNIT.

CSRU: .CMPR ZP,LINE,IM,01
  BNE CSRU4   ; Test if cursor at top of screen.
  .CLEAR CNTR  ; Cursor is at top; initialize 'CNTR'.
  .XMT CR   ; Position cursor at left edge.
CSRU1: .NULL   ; Fetch next character from stack.
  .CMPR ZP,CHAR,IM,ERR
  BNE CSRU2   ; Test if top of stack is reached.
  .CMPR ZP,CHAR,IM,CR
  BNE CSRU3   ; Test if next char is a carriage return.
  INCPZ CNTR   ; Loop until end of line or end of stack.
  INCZP CNTR   ; Count number of lines.
  .CMPR ZP,CNTR,IM,01
  BNE CSRU4   ; Test if 13 CR's have been found.
  .DOWN   ; Continue to loop until top of previous
          ; page is determined.
  .MOVE IM,01,IM,CR   ; Reposition stack pointer.
  .XMT CLEAR   ; Cursor line position is redefined.
CSRU2: .CRT-RST
  RTS  ; Use subroutine 'RST4' to reset the screen.
  RTS  ; Return.
THIS PORTION OF THE ROUTINE SIMPLY MOVES THE CURSOR UP FROM ITS PRESENT POSITION AND POSITIONS IT AT THE BEGINNING OF THE HIGHER LINE:

```
CSRU4: DECZP LINE ; CURSOR NOT AT TOP LINE--
    XMT CR ; DECREMENT THE CURSOR'S LINE POSITION.
    XMT CR-UP ; POSITION CURSOR TO LEFT EDGE OF CRT.
    MOVE IM/02,ZP,CNTR ; PUT CURSOR ON NEXT HIGHER CRT LINE.
CSRU5: .FULL CMPR IM,ERR,ZP,CHAR ; TEST IF THE STACK WAS EXCEEDED.
    BEQ CSRU6 ; STOP LOOPING IF TOP OF STACK IS FOUND.
    CMPR IM,CR,ZP,CHAR ; TEST FOR A 'CR'.
    BNE CSRU5 ; LOOP THROUGH STACK TO A CR.
    DECZP CNTR ; 'CNTR' COUNTS TWO 'CR' CHARACTERS.
    BNE CSRU5 ; BRANCH THROUGH STACK UNTIL 2 CR'S ARE FOUND.
    .DOWN ; POSITION SP JUST BELOW THE LAST CR.
CSRU6: RTS ; OPERATION IS COMPLETE.
```

THE FOLLOWING ROUTINE PERFORMS A 'CURSOR-DOWN' OPERATION UPON RECEPTION OF THE CORRESPONDING CONTROL CHARACTER. THE PROGRAMMING TAKES ADVANTAGE OF THE ROLL-UP FEATURE PRESENT IN THE CRT UNIT BEING USED:

```
CSRD: .DOWN CMPR IM,END,ZP,CHAR ; GOING DOWN THE STACK REQUIRES
    BEQ CSRD4 ; A DECREMENTAL COUNTER.
    CMPR IM,CR,ZP,CHAR ; TEST FOR A CARRIAGE RETURN.
    BNE CSRD ; IF NOT, CONTINUE TO LOOP.
    XMT CR ; XMT A LINE FEED CHAR TO CRT.
    XMTLF LINE ; XMT A CARRIAGE RETURN TO CRT.
    CMPR IM,/OD,ZP,LINE ; INC 'LINE'; TO MATCH NEW CURSOR
    BEQ CSRD0 ; POSITION, THEN TEST IF LINE
    RTS ; COUNT EXCEEDS /OC.
    RETURN.
CSRD0: DECZP LINE ; IF AT BOTTOM OF CRT, RESET LINE
CSRD1: .DOWN CMPR IM,CR,ZP,CHAR ; COUNT & FETCH NEXT CHAR IN FILE.
    BEQ CSRD3 ; IF NEXT CHAR IS A CR, GO TO 'CSRD2'.
    CMPR IM,END,ZP,CHAR ; IF BOTTOM OF STACK IS REACHED, RETURN.
    BEQ CSRD5 ; SP & RETURN.
    CMPR IM,CR,ZP,CHAR ; IF END OF TEXT IS FOUND, REPOSITION
    BNE CSRD2 ; PRINT CHAR ON CRT.
    XMT96 ; LOOP UNTIL NEXT 'CR' IS FOUND.
    JMP CSRD1
CSRD2: .FULL ; 'CR'. THEN RETURN.
CSRD3: RTS

CSRD4: .FULL CMPR IM,ERR,ZP,CHAR ; REPOSITION THE SP UP ONE LOCATION.
    BEQ CSRDS ; TEST IF AT TOP OF FILE; & RETURN
    ; IF SO.
    CMPR IM,CR,ZP,CHAR ; LOOK FOR A 'CR' & LOOP BACK IF NOT
    BNE CSRD4 ; FOUND.
    .DOWN ; POSITION SP JUST BEYOND THE LOCATED
    CSRDS: RTS ; 'CR'. THEN RETURN.
```

THE FOLLOWING SUBROUTINE MOVES THE SCREEN CURSOR ONE PLACE TO THE LEFT AND SUBSEQUENTLY MOVES THE STACK POINTER 'UP' ONE POSITION IN MEMORY. IF THE CURSOR IS ALREADY AT THE EXTREME LEFT OF A LINE ON SCREEN, THE ROUTINE PLACES THE CURSOR AT THE FINAL POSITION ON THE LINE ABOVE. THE STACK POINTER IS MOVED TO POINT AT THE EXPECTED LINE FEED. IF THE CURSOR IS LOCATED AT THE EXTREME UPPER LEFT CORNER, NO ACTION IS PERFORMED BY THIS SUBROUTINE. TO POSITION THE CURSOR ON THE NEXT LINE (OFF THE SCREEN IN THIS CASE), A 'CURSOR-UP' COMMAND MUST BE EXECUTED.
This routine calls the subroutine 'CSRL.RHM' and executes a cursor-left command five times. Error control in the subroutine prevents illegal or impossible actions.

The following subroutine moves the cursor to the right one space on the memory screen and 'down' the stack one location in memory. If an end-of-line is reached, the cursor is automatically cycled to the beginning of the next line, and the stack pointer is positioned after the expected line feed and carriage return ASCII characters. If a 'cursor-right' is attempted at the lower right corner of the text, no action will result. To proceed to the next page, a 'cursor-down' must be executed.

CSRR:  LDYIM  00  ;TEST IF SP POINTS TO END OF TEXT.
  MOVE  D+SP-L,ZP,CHAR ;IF SO, DO NOTHING AND RETURN TO
  CMPR  ZP,CHAR,IM,CR  ;POURING ROUTINE.
  BNE  CSRR2  ;RETURN TO POLLING IF AT BOTTOM.
  CMPR  ZP,CHAR,IM,ER  ;TEST FOR A CR AND BRANCH IF NOT FOUND.
  BNE  CSRR1  ;RETURN TO POLLING IF AT BOTTOM.
  JMP  CSRL5  ;REPOSITION THE SP AFTER THE CR.
  INCZP  LINE  ;INCREMENT THE SCREEN LINE COUNTER.
  RTS  ;RETURN.
CSRR1:  XMT  C-RT  ;XMT A 'C-RT' TO CRT.
  DOWN  ;POINT THE SP AT THE NEW LOCATION.
  RTS  ;RETURN.
CSRR2:  XMT  C-LFT  ;REPOSITION CURSOR BACK ONE SPACE.
  RTS  ;RETURN.
THIS ROUTINE CALLS THE SUBROUTINE ‘CSRR.RHM’ AND EXECUTES A ‘CURSOR-RIGHT’ COMMAND FIVE TIMES. IF ILLEGAL OR IMPOSSIBLE ACTIONS ARE REQUESTED BY THIS COMMAND, THE BUILT-IN SAFEGUARDS OF ‘CSRR.RHM’ WILL PROVIDE ERROR CONTROL.

CSRR5

CSRR6:

JSR CSRR

JUMP TO SUBROUTINE THAT EXECUTES A ‘CURSOR-RIGHT’ COMMAND.

DECZP

CNTRL

DECREMENT THE LOOP COUNTER.

BNE CSRR6

CONTINUE TO LOOP IF NOT COMPLETE.

RTS RETURN

THE FOLLOWING ROUTINE DELETES A CHARACTER AT THE CURRENT CURSOR POSITION AND MOVES ALL OTHER CHARACTERS UP ONE POSITION IN THE DATA FILE. THIS WILL KILL INDIVIDUAL CHARACTERS, AND LINE REORGANIZATION IS AUTOMATICALLY INCLUDED.

DELETE:

SP-SAVE

SAVE THE SP.

CLEAR MODE

USE ‘MODE’ TO FLAG A CR LATER.

DELETE1:

LDYIM /00

LOAD Y-INDEX.

MOVE IMP,SP-L,ZP,CHAR

MOVE CHAR FROM STACK AT POINTER.

CMPR IMP,CR,ZP,CHAR

TEST FOR CR.

BNE DELET2

MOVE IMP,FF,ZP,MODE

SET THE FLAG FOR A CR.

DELET2:

CMPR IMP,EOT,ZP,CHAR

TEST FOR THE END OF THE TEXT.

BNE DELET3

RTS RETURN

DELET3:

DOWN

POSITION THE SP DOWN ONE PLACE.

LDYIM /00

CLEAR THE Y-INDEX.

LDADY SP-L

LOAD CHAR FROM STACK AT POINTER.

INY

USE Y-REG TO POINT AT ONE LOCATION HIGHER THAN SP. STORE CHAR THERE.

CMPIM EOT

TEST THE ACC AGAINST ‘EOT’.

BEC DLET4

REDONE.

JMP DLET3

CONTINUE LOOPSING THROUGH STACK.

DELET4:

SP-RESTORE

RESTORE THE SP TO ORIGINAL POSITION.

CMPIMP,FF,ZP,MODE

TEST TO SEE IF ONLY A LINE NEEDS TO BE REDONE.

BNE DLET5

CRT-RST

RESET CRT SCREEN WITHOUT SP ON A CR.

RTS RETURN.

DELET5:

JSR RSTSC

RESET THE CRT DISPLAY.

RTS RETURN.

THE FOLLOWING GENERAL CURSOR UTILITY ROUTINE RESETS THE SCREEN FROM THE EXISTING STACK POINTER POSITION ONWARDS.

RSTSC:

SP-SAVE

SAVE THE SP IN LOCATION TSP+IL

CLEAR CNTR

USE ‘CNTR’ TO COUNT CHAR TO END OF LINE.

DOWN

FETCH CHAR FROM STACK & MOVE SP DOWN.

CMPIM CR,ZP,CHAR

FETCH CHAR FROM STACK AT END OF LINE.

BEC RST2

TEST FOR END OF THE LINE.

CMPIM EOT,ZP,CHAR

FETCH FOR END OF TEXT.

BEC RST2

TEST IF STACK ARRAY HAS BEEN EXCEEDED.

BEC RST2

XMT96 TxMt CHAR UNTIL END OF LINE IS FOUND.

INCZP CNTR

LOOP BACK UNTIL END OF LINE IS FOUND.
RST2: .MOVE IM+,/20,ZP CHAR; END OF LINE IS FOUND & A BLANK IS SENT
.XMT96 ; TO ERASE A POSSIBLE EXTRA CHARACTER.
INCZP CNTR
RST3: .MOVE IM+C-LFT,ZP CHAR; READY TO PUT CURSOR BACK IN ORIGINAL SPOT.
DECZP CNTR
DNE RST3 ; DECREMENT CNTR UNTIL ORIG POSITION.
.SP-RESTORE
RTS

; THIS PORTION OF THE ROUTINE ASSUMES THAT THE ENTIRE SCREEN MUST
; BE REMADE.
RST4: .SP-SAVE ; SAVE THE CURRENT SP LOCATION.
.XMT ERASE ; ERASE CRT FROM CURSOR ONWARDS.
.RST; MOVE ZP+LINE,ZP,TLINE ; SAVE THE LINE COUNT FOR LATER.
RST5: ; FETCH THE CURRENT CHAR FROM STACK.
.CMPR IM,CR,ZP,CHAR ; TEST IF A LINE HAS BEEN REACHED.
BEQ RST6
.CMPR IM,EDT,ZP,CHAR ; TEST IF THE END OF TEXT WAS REACHED.
BEQ RSTB
.CMPR IM,ERR,ZP,CHAR ; TEST IF BOTTOM OF FILE IS REACHED.
BEQ RST5
.XMT96 ; PRINT THE CHAR ON CRT
JMP RST5

RST6: ; A 'CR' WAS FOUND & LINE COUNT IS
.CMPR IM+/OC,ZP+LINE ; TESTED.
BEQ RST5
.INCZP LINE ; IF NOT, SEND A 'CR', 'LF', &
.XMTLF ; CONTINUE TOO LOOP.
.XMT CR
JMP RST5

RST8: ; POSITION THE SP AT THE PREVIOUS CHAR.
.RST; FETCH THE CHARACTER & POSITION THE SP
.CMPR IM,CR,ZP,CHAR ; HIGHER IN THE FILE. TEST IF THE SP
BEQ RSTB ; HAS EXCEEDED THE FILE.
.CMPR IM,CR,ZP,CHAR ; TEST IF THE CHAR WAS A 'CR'.
BEQ RST9
.CMPR ZP+LINE,ZP+LINE ; IF A 'CR', TEST IF THE LINE COUNT HAS
BEQ RSTA ; TO ITS ORIGINAL VALUE (& POSITION).
DECZP LINE ; IF NOT, DECREMENT THE LINE COUNT &
.XMT CR ; THE CURSOR ON NEXT HIGHER LINE AT
.XMT C-UP ; LEFT MARGIN.
JMP RST9 ; LOOP 'TIL CURSOR IS ON ORIGINAL LINE.

RSTA: ; PLACE CURSOR AT LEFT MARGIN.
.RST; FETCH NEXT CHAR & DECREMENT SP.
.RSTC: ; LOWER BYTE OF SP MATCHES THE
.CMPR ZP,SP-L,ZP,TSP-L ; TEST IF LOWER BYTE OF SP MATCHES THE
BEQ RSTD ; LOWER BYTE OF THE ORIGINAL SP.
.XMT C-RT ; MOVE CURSOR RIGHT UNTIL IT DOES.
.DONW ; FETCH NEXT CHAR & DECREMENT SP.
.JMP RSTD ; LOOP UNTIL LOWER BYTES MATCH.

RSTD: ; RETURN.

; THIS NEXT UTILITY ROUTINE PROVIDES THE ABILITY OF INSERTING
; ANY LEGAL, TYPED CHARACTER (NOT INCLUDING CONTROL CHARACTERS)
; INTO THE STACK AT THE CURRENT STACK POSITION. THE CURSOR
; POSITION IS FILLED WITH THE NEW CHARACTER; AND ALL CHARACTERS
; BELOW THE CURRENT POSITION ARE MOVED DOWN THE STACK ONE PLACE.
; THE CURRENT POSITION IS NOW FOUND
; MOVED ONE PLACE DOWN THE STACK.
INSRT: .SF-SAVE ♦ STORE THE STACK POINTER.
.XMT96

INSR1: LDYIM /00 ♦ CLEAR Y-REG SO NO OFFSET OCCURS.
.MOVE DY,SP-L,ZF,CHAR1 ♦ REMOVE CHAR FROM STACK POSITION.
.PUSH ♦ PUT NEW CHAR INTO STACK.
.MOVE ZP,CHAR1,ZF,CHAR ♦ TRANSFER 'EOT' TO 'CHAR'.
.CMPR ZP,CHAR,IM,EOT ♦ TEST IF END-OF-TEXT IS REACHED.
BNE INSR1
.PUSH ♦ STORE THE STACK POINTER.
.SF-RESTORE ♦ MOVE SP TO POINT AT ORIG. CHAR.
.RTS ♦ RETURN

THE NEXT ROUTINE ERASES ALL CONTENTS OF THE STACK FROM
THE CURRENT CURSOR LOCATION ONWARDS TO THE END OF THE STACK.
AND PRINTS IT OUT ON THE TTY. A DUPLICATE OF THE CHARACTER
LOCATED AT CURSOR LOCATION ON THE CRT IS
ALSO ERASED. THE CRT ERASES ALL CHARACTERS FROM THE CURRENT
CURSOR LOCATION TO THE END OF THE SCREEN.

EFILE: .MOVE IM,EOT,ZP,CHAR ♦ PUSH AN 'EOT' CHAR ONTO STACK.
.PUSH ♦ RESET THE STACK POINTER.
PULL ♦ THE 'ERASE' CHAR CLEARS THE CRT FROM
.XMT ERASE ♦ THE PRESENT CURSOR POSITION.
.RTS ♦ RETURN.

THE PRINT ROUTINE TAKES THE MEMORY CONTENTS OF THE STACK
PRINTS IT OUT ON THE TTY. A DUPLICATE OF THE CHARACTER
SENT TO THE TTY IS ECHOED ON THE CRT.

PRINT: .CLEAR CMD1 ♦ USE 'CMD1' TO TEST FOR A 'BREAK'.
.XMT CLEAR ♦ SIGNAL FROM CRT. CLEAR SCREEN.
.MOVE IM,CR,ZF,CHAR ♦ REPOSITION TTY CARRIAGE TO THE LEFT.
.XMT11
PRIN1: .DOWN ♦ TEST IF BOTTOM OF STACK IS EXCEEDED.
.CMPR ZP,CHAR,IM,ERR BEQ PRIN2 ♦ TEST IF END-OF-TEXT WAS ENCOUNTERED.
.CMPR ZP,CHAR,IM,EOT BEQ PRIN2 ♦ TEST IF A CR (& LF) MUST BE SENT.
.CMPR IM,CR,ZF,CHAR BNE PRIN4 ♦ TEST IF 'CMD1' WAS SET EARLIER.
.MOVE IM,LF,ZF,CHAR ♦ IF NOT SET, SET 'CMD1' ON THIS PASS.
.PRIN1 ♦ LOOP BACK FOR ANOTHER CHARACTER.
.XMT11 ♦ XMT A CR TO TTY & CRT.
.XMTLF
PRIN4: .MOVE IM,CR,ZF,CHAR ♦ XMT ECHO TO CRT.
.XMT11 ♦ XMT CHAR TO TTY.
.LDAAB CRT ♦ TEST FOR AN INCOMING CHAR.
.ANDIM /00 ♦ IF NO CHAR START BIT, LOOP BACK.
.BEQ PRIN1 ♦ TEST IF 'CMD1' WAS SET EARLIER.
.LDAZP CMD1 ♦ IF SET, TERMINATE PRINTING.
.BNE PRIN2 ♦ IF NOT SET, SET 'CMD1' ON THIS PASS.
.STAQP CMD1 ♦ JUMP TO PRIN1.
.JMP ♦ LOOP BACK FOR ANOTHER CHARACTER.
PRIN2: .MOVE IM,CR,ZF,CHAR ♦ XMT A CR TO TTY & CRT.
.XMT96 ♦ TRANSMIT A CR AND 3 LF'S TO TTY.
.XMT11 ♦ AND CRT.
.XMT11 ♦ XMT11
PRIN3: RTS ♦ THE PRINTING IS DONE.

; END
### APPENDIX B

**OPERATING INSTRUCTIONS FOR THE PROTOTYPE RESIDENT MONITOR AND TEXT EDITOR**

<table>
<thead>
<tr>
<th>Key</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Monitor Commands</strong></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>Enter the &quot;typewriter&quot; mode of operation.</td>
</tr>
<tr>
<td>D</td>
<td>Choose a disc for data storage uses.</td>
</tr>
<tr>
<td>L</td>
<td>Load a 1024-byte file from disc into the KIM.</td>
</tr>
<tr>
<td>S</td>
<td>Store a 1024-byte file onto a chosen disc.</td>
</tr>
<tr>
<td>R</td>
<td>Rename a desired file on disc.</td>
</tr>
<tr>
<td>K</td>
<td>Kill a file already stored on a common disc.</td>
</tr>
<tr>
<td>F</td>
<td>Format a pre-selected disc.</td>
</tr>
<tr>
<td><strong>Editor Commands</strong></td>
<td></td>
</tr>
<tr>
<td>(Up-Arrow)</td>
<td>Moves cursor up one line.</td>
</tr>
<tr>
<td>(Down-Arrow)</td>
<td>Moves cursor down one line.</td>
</tr>
<tr>
<td>(Right-Arrow)</td>
<td>Moves cursor right one position.</td>
</tr>
<tr>
<td>(Left-Arrow)</td>
<td>Places cursor left one position.</td>
</tr>
<tr>
<td>Control-L</td>
<td>Move cursor left five times.</td>
</tr>
<tr>
<td>Control-R</td>
<td>Move cursor right five times.</td>
</tr>
<tr>
<td>RUBOUT</td>
<td>Delete a character at the cursor location.</td>
</tr>
<tr>
<td>Control-C</td>
<td>Return to monitor.</td>
</tr>
</tbody>
</table>
APPENDIX C

THE DOWNLOADING PROCESS: PDP-11 SOURCE CODE, LINKED SUBROUTINES, AND KIM-1 PROGRAMS

<table>
<thead>
<tr>
<th>Address</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>00</td>
</tr>
<tr>
<td>0001</td>
<td>00</td>
</tr>
<tr>
<td>0002</td>
<td>02</td>
</tr>
<tr>
<td>1703</td>
<td>00</td>
</tr>
<tr>
<td>17C0</td>
<td>A0</td>
</tr>
<tr>
<td>17C1</td>
<td>00</td>
</tr>
<tr>
<td>17C2</td>
<td>AD</td>
</tr>
<tr>
<td>17C3</td>
<td>02</td>
</tr>
<tr>
<td>17C4</td>
<td>17</td>
</tr>
<tr>
<td>17C5</td>
<td>AA</td>
</tr>
<tr>
<td>17C6</td>
<td>29</td>
</tr>
<tr>
<td>17C7</td>
<td>04</td>
</tr>
<tr>
<td>17C8</td>
<td>F0</td>
</tr>
<tr>
<td>17C9</td>
<td>FB</td>
</tr>
<tr>
<td>17CA</td>
<td>8A</td>
</tr>
<tr>
<td>17CB</td>
<td>38</td>
</tr>
<tr>
<td>17CC</td>
<td>29</td>
</tr>
<tr>
<td>17CD</td>
<td>02</td>
</tr>
<tr>
<td>17CE</td>
<td>D0</td>
</tr>
<tr>
<td>17CF</td>
<td>01</td>
</tr>
<tr>
<td>17D0</td>
<td>18</td>
</tr>
<tr>
<td>17D1</td>
<td>26</td>
</tr>
<tr>
<td>17D2</td>
<td>00</td>
</tr>
<tr>
<td>17D3</td>
<td>8A</td>
</tr>
<tr>
<td>17D4</td>
<td>29</td>
</tr>
<tr>
<td>17D5</td>
<td>08</td>
</tr>
<tr>
<td>17D6</td>
<td>F0</td>
</tr>
<tr>
<td>17D7</td>
<td>05</td>
</tr>
<tr>
<td>17D8</td>
<td>A5</td>
</tr>
<tr>
<td>17D9</td>
<td>00</td>
</tr>
<tr>
<td>17DA</td>
<td>91</td>
</tr>
<tr>
<td>17DB</td>
<td>01</td>
</tr>
<tr>
<td>17DC</td>
<td>CB</td>
</tr>
<tr>
<td>17DD</td>
<td>AD</td>
</tr>
<tr>
<td>17DE</td>
<td>02</td>
</tr>
<tr>
<td>17DF</td>
<td>17</td>
</tr>
<tr>
<td>17E0</td>
<td>29</td>
</tr>
<tr>
<td>17E1</td>
<td>04</td>
</tr>
<tr>
<td>17E2</td>
<td>D0</td>
</tr>
<tr>
<td>17E3</td>
<td>F9</td>
</tr>
<tr>
<td>17E4</td>
<td>4C</td>
</tr>
<tr>
<td>17E5</td>
<td>C2</td>
</tr>
<tr>
<td>17E6</td>
<td>17</td>
</tr>
</tbody>
</table>

Figure C.1. Hand-Entered KIM-1 Download Program (Machine Code)
MAIN DOWNLOADER PROGRAM

AUTHOR: R. H. MC CARTNEY (AFTER F. J. YESKEL'S "LOADF.JY")
DATE: AUGUST 26, 1977
MACHINE: PDP-11/40 W/RT-11FB MONITOR
PROJECT: EE 410 MASTER'S THESIS
FILE: LDR.RHM

THIS FILE CONTAINS THE MAIN PROGRAM, WRITTEN IN FORTRAN, WHICH,
TOGETHER WITH ONE ASSEMBLY LANGUAGE SUBROUTINE, ALLOWS THE USER
TO DOWNLOAD ASSEMBLED MACHINE CODE FROM THE PDP-11/40 DIRECTLY
TO THE KIM-1. THIS DOWNLOADING PROCESS TAKES PLACE VIA THE DR-11C
INTERFACE BOARD ON THE PDP-11. THE KIM-1 ADDRESSES AND DATA ARE
DOWNLOAD SERIALLY.

THE MACHINE CODE DESIRED TO BE DOWNLOADED MUST BE IN THE FILE:
"MACHIN.RHM"
AND WILL BE READ FROM DISK USING THE FOLLOWING FORTRAN FORMAT:
5X,06,5X,2A1
WHICH REPRESENTS A 2-BYTE HEXADECIMAL ADDRESS OF 4 ASCII-CODED
CHARACTERS. THESE ARE FOLLOWED BY A SINGLE DATA BYTE IN HEX OF
TWO ASCII-CODED CHARACTERS.

TRANSMISSION OF DATA THROUGH THE DR-11C I/O CARD REQUIRES USE OF
ABSOLUTE ADDRESSES. THUS AN ASSEMBLY-LEVEL SUBROUTINE WAS REQUIRED
TO COMPLETE THE HARDWARE/SOFTWARE INTERFACING. THE SUBROUTINE
SOURCE CODE WILL BE FOUND IN THE FILE TITLED:
LDRM.RHM
AND THE FORTRAN STATEMENT NECESSARY TO IMPLEMENT THIS SUBROUTINE IS
CALL OUT (SUMCHK,CNTRL,ARG)
WITH THE WORD 'CALL' BEGINNING IN THE SEVENTH SPACE FROM THE LEFT.

THE THREE TRANSMISSION LINES USED BETWEEN THE PDP-11 AND THE
KIM-1 ARE CONNECTED TO THE DR-11C THROUGH BITS $2, 3, AND
OF THE 16-BIT OUTPUT PORT. NOTE: THE BIT NUMBERING CONVENTION
IS FROM THE LSB ($0) TO THE MSB ($15).

A LINKED SUBROUTINE TITLED 'BNHEX' MUST BE LINKED TO THIS MAIN
PROGRAM. IT MAY BE FOUND UNDER THE FILE NAME OF 'BNHEX.RHM'.

SPECIAL NOTE:
THIS MAIN PROGRAM IS DESIGNED TO DOWNLOAD THE MAIN KIM
LOADER ROUTINE (SEE FILE "KLDR.RHM") USING ONLY THE
HAND-LOADED MINILOADER (SEE FILE "MINIL.RHM"). NO
ADDITIONAL MODIFICATIONS ARE REQUIRED TO THIS OR EITHER
OF THE OTHER TWO PROGRAMS.

MAIN SOURCE CODE
THE MAIN PROGRAM BEGINS BY CREATING THE NECESSARY ARRAYS AND
INTEGERS FOR DATA MANIPULATION.
DIMENSION HEX(4),DATAA(2)
INTEGER SUMCHK,LOW,HI,DATAA,ADDR,BIN,HEX,CNTRL
INITIALIZATION
SUMCHK=0
ADDR=511
READ HIGH ADDRESS, LOW ADDRESS, AND DATA
CALL ASSIGN (7,'OKOIMACHIN.RHM',14)
10 READ (7,END=100) BIN,DATAA
FORMATT (5X,06,5X,2A1)
CNTRL=1
NOW CONVERT THE BINARY ADDRESS TO HEXADECIMAL FORM.

CALL B2HEX(BIN,HEX)

CHECK TO SEE IF ADDRESS IS IN SERIES OR IS A NEW ADDRESS
IF (BIN.EQ.(ADDR+1)) CNTRL=0

CONVERT HIGH ADDRESS, LOW ADDRESS, AND DATA TO BINARY NUMBERS
ADDR=BIN
M=HEX(3)
H=HEX(4)
HEX(3)=HEX(1)
HEX(4)=HEX(2)
HEX(1)=8240
HEX(2)=8240
CALL HEXTB (HEX,HI)
HEX(3)=M
HEX(4)=N
CALL HEXTB (HEX,LOW)
HEX(3)=DATAA(1)
HEX(4)=DATAA(2)
CALL HEXTB (HEX,DATA)

CALL ROUTINE 'OUT' TO SEND DATA TO KIM; THEN READ NEXT LINE
OF SOURCE FILE.
IF (CNTRL.EQ.0) GO TO 30
CALL OUT (SUMCHK,CNTRL,LOW)

CNTRL=2
CALL OUT (SUMCHK,CNTRL,HI)

CNTRL=0
GO TO 10

THE END OF SOURCE FILE HAS BEEN REACHED. SEND THE CONTROL
BYTE '03' TO THE KIM INDICATING A 'STOP' COMMAND FOLLOWED BY
A COMPLETION MESSAGE TO THE OPERATOR.

DATA=SUMCHK
CALL OUT (SUMCHK,CNTRL,DATA)
WRITE (5,110) DATA

110 FORMAT (1X,'SUMCHECK=',I4)

120 FORMAT (1X,'******** DOWNLOADING PROCESS IS COMPLETE ********')
CALL CLOSE(7)
STOP
END

ASCII-TO-BINARY SUBROUTINE

THIS SUBROUTINE IS A HEXADECIMAL-TO-BINARY CONVERSION ROUTINE. IT
CONVERTS A HEXADECIMAL NUMBER, COMPOSED OF FOUR ASCII CHARACTERS IN
A1 FORMAT, INTO A 16-BIT BINARY NUMBER.

SUBROUTINE HEXTB (HEX,HI)
DIMENSION HEX(4)
INTEGER HEX,BIN
BIN=0
DO 20 I=1,4
IF (HEX(I).GT.6256) GO TO 10
BIN=BIN+(HEX(I)-8240)*16**((4-I))
GO TO 20
20 CONTINUE
RETURN
END
**BINARY TO ASCII—FORM HEXADECIMAL**

**SUBROUTINE**

```plaintext
SUBROUTINE RNHEx(B,H)
DIMENSION H(4)
INTEGER B,H,I,E,F
I=1
E=8
DO 220 I=4,1,-1
    F=E/(16**(I-1))
    IF (F.LT.10) GO TO 200
    F=F+7
200  H(5-I)=F#240
          E=E-(F*(16**(I-1)))
220 CONTINUE
RETURN
END
```

**ASSEMBLY-LANGUAGE SUBROUTINE**

**PDP-11 & KIM-1 DATA TRANSMISSION**

**AUTHOR:** R. H. MC CARTNEY (AFTER F. J. YESKEL'S *LOADM.FJY*)
**DATE:** AUGUST 26, 1977
**MACHINE:** PDP-11/40 W/RT-11FB MONITOR
**PROJECT:** EE 410 MASTER'S THESIS
**FILE:** LDRM.RHM

**THIS FILE CONTAINS AN ASSEMBLY LANGUAGE SUBROUTINE THAT**
**IS CALLED BY THE FORTRAN DOWNLOADER PROGRAM IN THE FILE**
"LDRM.RHM".

**THE FUNCTION OF THIS SUBROUTINE IS TO CONTROL COMMUNICATION**
**BETWEEN THE MAIN FORTRAN DOWNLOADER PROGRAM AND THE PDP-11**
**DR-11C INTERFACE BOARD OUTPUT REGISTER. THE FORTRAN CALL**
**SUBROUTINE STATEMENT IS OF THE FORM:**

CALL OUT (SUMCHK,CNTRL,ARG)

**WHERE THE ARGUMENTS ARE AS FOLLOWS:**

**SUMCHK** — THE EXCLUSIVE OR'D 'SUM' OF ALL DATA & ADDRESS BYTES
TRANSMITTED. USED LIKE A PARTIALLY CHECK AT END OF
TRANSMISSION.

**CNTRL** — THE UPPER BYTE OF EACH WORD SENT TO THE KIM,
CONTAINS INFORMATION ON HOW THE LOWER BYTE IS
TO BE INTERPRETED.

**ARG** — THE LOWER BYTE OF EACH WORD SENT TO THE KIM,
CONTAINS EITHER DATA OR ADDRESS INFORMATION.
SUBROUTINE SOURCE CODE

```assembly
.MCALL .REGDEF
.REGDEF
.GLOBL OUT
.DEFIN "OUT" AS GLOBAL WITH FORTRAN CODE

OUT:
MOV B2(R5),R0    ;R0=MUNCHK
MOV B6(R5),R3    ;PUT DATA BYTE INTO R3
XOR R3,R0        ;EXCLUSIVE OR WITH ARGUMENT
BIC @177400,R3   ;CLEAR UPPER 8 BITS (NOT SEEN BY KIM)
MOV R0,B2(R5)    ;RETURN MUNCHK TO MAIN PROGRAM LOCATION
MOV $000,$167772 ;CLEAR ALL OUTPUT PORT BITS

MOV $000,$167770 ;CLEAR ALL STATUS PORT BITS
MOV B6(R5),R3    ;PLACE DATA BYTE IN R3 AGAIN
MOV B4(R5),R1    ;PUT CONTROL BYTE IN R1
BIC @177400,R3   ;CLEAR UPPER 6 BITS OF R3
ASL R1           ;SHIFT R1 LEFT 6 BITS
BIC *177400,R3   ;CLEAR ALL OUTPUT PORT BITS
ASL R1
ASL R1
ASL R1
ASL R1
ASL R1
ASL R1
ASL R1

JUMP0: DEC R2     ;DECREMENT THE BIT COUNTER
BCE JUMP1       ;BRANCH TO "JUMP1" IF DATA BIT=0
MOV @1,R3       ;DATA =1: SET BIT IN OUTPUT BUFFER
MOV R3,$167772  ;SET BUFFER TO "DATA BIT"
JSR R5,DELAY    ;DELAY 100 MICROSEC FOR KIM
JSR @0,$167772  ;CLEAR STROBE AND DATA FOR NEXT BIT

JUMP2: MOV @1,R3  ;UNCONDITIONALLY SET DATA BIT IN BUFFER
JSR @1,$167770  ;SET STATUS REG BIT #7 HIGH (EDO)
ROL R1           ;SHIFT LAST BIT INTO THE CARRY
BCE JUMP3       ;IF CARRY=0, CLEAR BIT IN R3 BUFFER
MOV $000,R3      ;CLEAR DATA BIT IN BUFFER

JUMP3: BTS @2,R3  ;SET DATA STROBE BIT
MOV R3,$167772   ;SET BITS ON OUTPUT PORT
JSR R5,DELAY     ;DELAY FOR THE KIM
JSR @0,$167772   ;CLEAR OUTPUT PORT
JSR @0,$167770   ;CLEAR STATUS PORT & EDW BIT
JSR R5,DELAY     ;DELAY WHILE STROBE LINE REMAINS LOW
RTS PC           ;RETURN TO MAIN FORTRAN PROGRAM

THE FOLLOWING ASSEMBLY-LEVEL SUBROUTINE CAUSES A DELAY OF
APPROXIMATELY 100 MICROSECONDS.

DELAY: MOV $30,R3  ;SET R3 AS DELAY COUNTER
DELAY1: DEC R3    ;DECREMENT THE DELAY COUNTER
BNE DELAY1       ;CONTINUE TO LOOP IF DELAY ISN'T DONE
RTS R5           ;RETURN TO INSTRUCTIONS ABOVE
.END
```
KIM-1 ASSEMBLY LANGUAGE
LOADING ROUTINE

AUTHOR: R. H. MC CARTNEY
DATE: AUGUST 26, 1977
MACHINE: PDP-11/40 W/RT-11FB MONITOR
PROJECT: EE 410 MASTER'S THESIS
FILE: KLD.RHM

THE FOLLOWING ASSEMBLY-LEVEL PROGRAM IS WRITTEN FOR THE
MOSTEK 6502 MICROPROCESSOR-BASED KIM-1 UNIT. THE PURPOSE
OF THIS PROGRAM IS TO READ SERIAL DATA AND CORRECTLY
INSERT THE RESULTING INFORMATION INTO ASSIGNED MEMORY
LOCATIONS ON THE KIM UNIT.

TRANSMISSION OF DATA FROM PDP-11 TO KIM IS ACCOMPLISHED BY
THREE DATA LINES:

LINE 1: DATA OUT (+5 VOLTS = LOGICAL '1')
ASSIGNED TO BIT #1 OF KIM ADDRESS /4000
LOCATED ON PIN #1 OF THE 26-PIN PDP-11 CABLE

LINE 2: DATA STROBE (+5 VOLTS = DATA READY)
ASSIGNED TO BIT #2 OF KIM ADDRESS /4000
LOCATED ON PIN #3 OF THE 26-PIN PDP-11 CABLE

LINE 3: END OF WORD (+5 VOLTS = LAST OF 16 BITS)
ASSIGNED TO BIT #3 OF KIM ADDRESS /4000
LOCATED ON PIN #13 OF THE 26-PIN PDP-11 CABLE

THE PROGRAM ORIGIN IS CHOSEN TO BE AT LOCATION /DC00
TO POSITION ITSELF AT THE END OF THE ALLOCATED ROM MEMORY
AREA.

.ORG /D60

THE MAIN ROUTINE NOW BEGINS WITH INITIALIZATION PROCEDURES.

START
LDX #0 CLEAR
STX #0 SUMCK CLEAR THE SUMCHECK BYTE
STX #0 CTRLA SET PORTA TO THE BI-DIR MODE
STX #0 PORTA SET ALL PORT LINES TO READ-MODE
TXA CLEAR THE ACCUMULATOR
TAY CLEAR THE Y-REG
LDX /04 Activate PORTA TO READ FROM PDP-11
STX #0 CTRLA
LOOP1
LDAB PORTA LOAD ACC WITH BYTE FROM PDP-11
AND #1 INCLUSIVE OF 16 (Except Strobe)
BNE JUMP1 LOOP IF STROBE NOT HIGH
LDAB PORTA AGAIN FETCH DATA PORT BYTE
STA STORE THE PORT DATA BYTE IN X-REG
ANDAB DATAB IBLANK ALL BITS EXCEPT DATA BIT
SEC SET CARRY BIT UNCONDITIONALLY
BCE JUMP2 IIF DATA BIT=1 BRANCH TO JUMP2
CLC CLEAR THE CARRY

JUMP2
ROL #0 DATAL SHIFT 'DATAL' LEFT, CARRY ENTERS LSR
ROL #0 DATHA COMPLETE A 2-BYTE SHIFT LEFT
TXA FETCH PORT DATA BYTE FROM X-REG
AND #1 INCLUSIVE OF 16 (Except 'EDW')
PCE JUMP3 'EDW' IS SET, ASSUME LAST BIT

LOOP3
LDAB PORTA FETCH PORT DATA BYTE FROM PDP-11
AND #1 INCLUSIVE OF 16 (Except Strobe)
PCE LOOP3 Mask all bits except the Strobe
JMP LOOP3 LOOP UNTIL STROBE BIT=0

JUMP4
LDAB DATHA LOAD ACC WITH CONTROL BYTE
STA STORE DUPLICATE IN X-REG
CPE /03 TEST FOR A STOP BYTE
BNE JUMP5 IF TRUE
LDAB DATAL LOAD DATA BYTE AND PERFORM AN
EORZP SUMCK ; 'EXCLUSIVE-OR' FOR SUMCHECK
STAZP SUMCK ; STORE THE EOR RESULT IN 'SUMCK'
TXA ; RESTORE CONTROL BYTE TO ACC
CMPIM /01 ; TEST FOR A LOW-BYTE OF AN ADDRESS
BED JUMPA ; JUMP TO ADDRESS ROUTINE IF TRUE
CMPIM /02 ; TEST FOR A HIGH-BYTE OF AN ADDRESS
BED JUMPB ; JUMP TO ADDRESS ROUTINE IF TRUE
JMP JUMPS ; ASSUME A DATA BYTE & GO TO 'JUMPS'

; IF THE PROGRAM CONTINUES WITH THE FOLLOWING CODE, WE ASSUME THAT
; A 'STOP' COMMAND WAS SENT.

JUMPS LDYM /EE ; PLACE SUMCHECK RESULT IN ACC
LDAZP SUMCK ; TEST IF IDENTICAL TO PDP-11 SUMCHECK
SNE JUMP4 ; ENTER ERROR DISPLAY IF NOT IDENTICAL
LDYM /CC ; ENTER A 'CORRECTLY-LOADED' CODE INTO LED'S
JUMP6 STYZP LED1 ; STORE SUMCHECK MATCH TEST RESULTS IN
STYZP LED2 ; LED DISPLAY OF KIM
JMF /1C4F ; JUMP TO KIM ROM DISPLAY ROUTINE 'START'

; THIS PORTION OF THE PROGRAM HANDLES A DATA BYTE.

JUMP6 LDYM CLEAR ; CLEAR THE X-REGISTER
CLD ; CLEAR THE DECIMAL MODE
LDAZP DATAL ; PLACE DATA BYTE IN ACC
STAZP ADDL ; STORE BYTE AT LOW ADDRESS BYTE LOCATION
INCZP ADDL ; REPOSITION LOW BYTE OF ADDR POINTER
SNE JUMPF ; PASS UPPER BYTE IF INCREMENT UNNECESSARY
INCZP ADDH ; INCREMENT UPPER ADDRESS BYTE
JUMPF JMF LOOP3 ; LOOP BACK FOR NEXT BYTE TRANSMISSION

; THE LOWER BYTE OF A NEW ADDRESS WAS SENT AND IS NOW PLACED
; IN ITS ZERO PAGE MEMORY LOCATION.

JUMPF LDAZP DATAL ; LOAD LOW ORDER ADDRESS BYTE AT 'ADDL'
STAZP ADDL
JMP LOOP3

; THE HIGHER BYTE OF A NEW ADDRESS WAS SENT AND IS NOW PLACED
; IN ITS ZERO PAGE MEMORY LOCATION.

JUMPF LDAZP DATAL ; LOAD HIGH ORDER ADDRESS BYTE AT 'ADDH'
STAZP ADDH
JMP LOOP3

; THE ASSIGNED MEMORY LOCATIONS FOR THE DATA FILES AND REQUIRED
; STORAGE REGISTERS ARE DEFINED IN THE FOLLOWING SECTION. THE
; NECESSARY ADDRESS LOCATIONS ARE RESERVED FOR STORAGE USES BY
; USE OF THE MACRO COMMAND '.EQU', PRECEDES BY THE BYTE VARIABLE,
; AND FOLLOWED BY THE TWO-BYTE ABSOLUTE ADDRESS.

LED1 .EQU /FA
LED2 .EQU /FB
ADDL .EQU /00
ADDH .EQU /01
DATAL .EQU /02
DATAH .EQU /03
SUMCK .EQU /04
PORTA .EQU /4000
CTRLA .EQU /4001
CLEAR .EQU /00
DATAB .EQU /02
STROE .EQU /04
EOW .EQU /08

.END
KIM-1 MINI-LOADER

A HAND-ENTERED PROGRAM FOR LOADING THE MAIN PDP-11/KIM-1 LOADER

AUTHOR: R. H. MC CARTNEY
DATE: AUGUST 26, 1977
MACHINE: PDP-11/40 W/RT-11FB MONITOR
PROJECT: EE 410, MASTER'S THESIS
FILE: MINILIURHM


THE PROGRAM IS PLACED IN THE SPECIAL 64-BYTE RAM OF THE MOS 6520 CHIP. THE STARTING ADDRESS IS 17C0 (HEX).

.ORG /17C0

FOUR SYMBOLIC VARIABLES OF ONE BYTE EACH ARE NOW ASSIGNED ABSOLUTE ADDRESSES.
DATA: .EQU /0000 ;TEMPORARY STORAGE OF INCOMING BYTES
ADDL: .EQU /0001 ;LOW ADDRESS BYTE FOR DATA DESTINATION
ADDH: .EQU /0002 ;HIGH ADDRESS BYTE
PORT: .EQU /1702 ;DOWNLOAD PORT CONNECTED TO PDP-11
DDR3B: .EQU /1703 ;DATA DIRECTION REGISTER FOR ADDRESS 1702

THE BIT FUNCTIONS OF THE DOWNLOAD PORT ON THE KIM (ADDRESS 1702) ARE AS FOLLOWS:

BIT 0: RESERVED FOR OTHER USES
BIT 1: DATA BIT
BIT 2: DATA STROBE (DATA BIT VALID WHEN STROBE GOES HIGH)
BIT 3: END-OF-WORD (EOW) GOES HIGH WITH 16TH BIT
BIT 4: UNUSED
BIT 5: UNUSED
BIT 6: NOT AVAILABLE ON KIM-1
BIT 7: RESERVED FOR OTHER USES

THE KIM REQUIRES THE FOLLOWING BYTE INITIALIZATION:

0001 00
0002 02
1703 00

THE SOURCE CODE OF THIS MINI-PROGRAM NOW BEGINS.
START:  LDYIM /00  USE Y-REG AS ADDRESS POINTER
LOOP1:  LDAAB PORT  LOAD DATA BYTE FROM PDP-11
         TAX  STORE DATA IN X-REG
         ANDIM /04  CLEAR ALL BITS EXCEPT STROBE
         BEQ LOOP1  LOOP TIL STROBE GOES HIGH
         TXA  PLACE PORT BYTE IN ACC.
         SEC  UNCONDITIONALLY SET CARRY
         ANDIM /02  BLANK ALL BITS EXCEPT DATA
         BNE JUMP1  BYPASS CLEARING CARRY IF DATA=1
         CLC  CLEAR CARRY IF DATA=0
JUMP1:  ROLZP DATA  SHIFT LEFT! LSB <- CARRY
         TXA  PUT PORT BYTE INTO ACC
         ANDIM /08  CLEAR ALL BITS EXCEPT 'EOU'
         BEQ LOOP2  IF 'EOU' IS SET, ENTER DATA
         LDAZP DATA  PLACE ASSEMBLED DATA BYTE IN ACC.
         STADY  STORE DATA DEFERRED-Y
         INY  INCREMENT Y POINTER
LOOP2:  LDAAB PORT  TEST IF STROBE GOES LOW
         ANDIM /04  CLEAR ALL BITS EXCEPT STROBE
         BNE LOOP2  LOOP TIL STROBE DROPS
         JMP LOOP1  LOOP BACK FOR NEXT BIT

THE MEMORY INITIALIZATION IS NOW INCLUDED IN THE LISTING.

.ORG /0000
.BYTE /00
.BYTE /02

.ORG /1703
.BYTE /00

.END
APPENDIX D

DISKETTE DATA FORMAT AND FILE STRUCTURE

Diskette File Storage

Sector Format Sequence:

<table>
<thead>
<tr>
<th>File Number</th>
<th>Sector Sequence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2, 8, 14, 20, 3, 9, 15, 21</td>
</tr>
<tr>
<td>2</td>
<td>4, 10, 16, 22, 5, 11, 17, 23</td>
</tr>
<tr>
<td>3</td>
<td>6, 12, 18, 24, 7, 13, 19, 25</td>
</tr>
</tbody>
</table>

Reserved Sectors:

SECTOR #01: May contain Track Identification and special data.

SECTOR #26: Reserved for future expansion.
APPENDIX E

EPROM CONTENTS LISTINGS

0000 A9 01 C5 0F D0 30 A9 00 35 00 A0 0D 20 18 01 20
0010 2F 00 A9 80 C5 03 F0 11 A9 0D C5 03 D0 F1 E6 00
0020 A9 0D C5 00 D0 E9 20 4D 00 A9 01 85 0F A0 1C 20
0030 18 01 20 CE 0D 60 C6 0F A0 0D 20 18 01 A0 0B 20
0040 18 01 A9 02 35 00 20 2F 00 A5 03 C9 80 F0 0D A9
0050 0D C5 03 D0 F1 C6 00 D0 ED 20 4D 00 60 20 4D 00
0060 A5 03 C9 0E F0 38 A5 03 C9 0D D0 F1 A0 0D 20 18
0070 01 20 6C 01 E6 0F A5 0F C9 0D F0 01 60 C6 0F 20
0080 4D 00 A5 03 C9 0D F0 12 A5 03 C9 80 F0 0F A5 03
0090 C9 0E F0 06 20 1A 01 4C 7F 0C 20 2F 00 60 20 2F
00A0 00 A5 03 C9 80 F0 09 A5 03 C9 0D D0 F1 20 4D 00
00B0 60 20 2F 00 A9 80 C5 03 F0 4A A9 0D C5 03 D0 45
00C0 A9 01 C5 0F F0 3E C6 0F 20 2F 00 A5 03 C9 80 F0
00D0 09 A9 0D C5 03 D0 F1 20 4D 00 A0 0D 20 18 01 A0
00E0 0B 20 18 01 20 4D 00 A5 03 C9 0D F0 14 A5 03 C9
00F0 80 F0 01 1A A5 03 C9 0E F0 0B A0 1F 20 18 01 4C E4

Erasable/Programmable Read-Only Memory

(EPROM) Contents: Addresses /0C00 to /OCFF

198
0100 0C 20 2F 00 60 A0 03 20 18 01 60 A9 05 85 01 20
0110 B1 0C C6 01 D0 F9 60 A0 00 B1 08 85 03 A9 0E C5
0120 03 F0 22 A9 0D C5 03 D0 14 A9 0C C5 0F F0 16 20
0130 6C 01 A0 0D 20 18 01 20 4D 00 60 A9 05 85 01 20 17 0D C6 01 D0
0140 18 01 20 4D 00 60 A9 05 85 01 20 17 0D C6 01 D0
0150 F9 60 20 5A 01 A9 00 85 11 A0 00 B1 08 85 03 A5
0160 03 C9 0D D0 04 A9 FF 85 11 A5 03 C9 0E D0 01 60
0170 20 4D 00 A0 00 B1 08 C8 91 08 C9 0E F0 03 4C 70
0180 0D 20 63 01 A5 11 C9 FF D0 04 20 CE 0D 60 20 92
0190 0D 60 20 5A 01 A9 00 85 00 20 4D 00 A5 03 C9 0D
01A0 F8 14 A5 03 C9 OF F0 OF A5 03 C9 80 F0 0B 20 1A
01B0 01 E6 00 4C 99 0D A9 20 85 03 20 1A 01 E6 00 A9
01C0 0B 85 03 20 1A 01 C6 00 D0 F9 20 63 01 60 20 5A
01D0 01 A0 0C 20 18 01 A5 0F 85 10 20 4D 00 A5 03 C9
01E0 0D F0 12 A5 03 C9 OF F0 1F A5 03 C9 80 F0 1C 20
01F0 1A 01 4C DA 0D A5 OF C9 OF F0 0D E6 OF 20 6C 01

Erasable/Programmable Read-Only-Memory

(EPROM) Contents: Addresses /0D00 to /0DFF
Erasable/Programmable Read-Only-Memory

(PROM) Contents: Addresses /0E00 to /0EFF
Erasable/Programmable Read-Only-Memory

(EPROM) Contents: Addresses /0F00 to /OFFF
SELECTED BIBLIOGRAPHY

Chinnock, D. Interview conducted at the Computer Science Research Laboratory, Department of Electrical Engineering, University of Arizona, Tucson, October 23, 1977.


PWB ASSY, Data, and Control. California Computer Products, Inc.,
Drawing #18830-001, December 10, 1975.

RT-11 and CAPS-11 Text Editor. Digital Equipment Corporation,
Sunnyvale, CA, 1974.

Signetics Digital/Linear/MOS Catalog. Signetics Corporation,
Sunnyvale, CA, 1972.

Signetics Full Product Line. Signetics Corporation, Sunnyvale, CA,
May, 1977.

Software Application Manual. MOS Technology, Inc., Norristown, PA,
1976.

Supplement 3: Installation-Information. California Computer Products,

Timesharing User's Manual. Publication #UM-2c, University of Arizona


Williams, T. L. Bufferboard Design Booklet: KIM-1 Expansion. Depart-
ment of Electrical Engineering, University of Arizona, Tucson,
1976a.

Williams, T. L. PPS Display Board: KIM-1 Expansion Design. Department

Williams, T. L. PPS 1/0 Board: KIM-1 Expansion Design. Department
of Electrical Engineering, University of Arizona, Tucson, 1976c.

Williams, T. L. PPS Memory Board: KIM-1 Expansion Design. Department
of Electrical Engineering, University of Arizona, Tucson, 1976d.

Wilson, D. Interview conducted at the Computer Science Research
Laboratory, Department of Electrical Engineering, University of

Yeskel, F. J. PDP-11/KIM-1 Downloading Program. Department of