PDP-11
Paper Tape
Software Handbook
Order No. DEC-11-XPTSA-B-D
The information in this document is subject to change without notice and should not be construed as a commitment by Digital Equipment Corporation. Digital Equipment Corporation assumes no responsibility for any errors that may appear in this document.

The software described in this document is furnished under a license and may be used or copied only in accordance with the terms of such license.

Digital Equipment Corporation assumes no responsibility for the use or reliability of its software on equipment that is not supplied by DIGITAL.


The postage prepaid READER'S COMMENTS form on the last page of this document requests the user's critical evaluation to assist us in preparing future documentation.

The following are trademarks of Digital Equipment Corporation:

DIGITAL  DEC  DECsystem-10  MASSBUS
PDP  DECtape  OMNIBUS
DECUS  DIBOL  OS/8
UNIBUS  EDUSYSTEM  PHA
COMPUTER LABS  FLIP CHIP  RSTS
COMTEX  FOCAL  RSX
DDT  INDAC  TYPESET-8
DECCOMM  LAB-8  TYPESET-10

7/77-34
CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAL-11S ASSEMBLY LANGUAGE AND ASSEMBLER</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>CHARACTER SET</td>
<td>1-2</td>
</tr>
<tr>
<td>1.2</td>
<td>STATEMENTS</td>
<td>1-2</td>
</tr>
<tr>
<td>1.2.1</td>
<td>Label</td>
<td>1-3</td>
</tr>
<tr>
<td>1.2.2</td>
<td>Operator</td>
<td>1-3</td>
</tr>
<tr>
<td>1.2.3</td>
<td>Operand</td>
<td>1-4</td>
</tr>
<tr>
<td>1.2.4</td>
<td>Comments</td>
<td>1-4</td>
</tr>
<tr>
<td>1.2.5</td>
<td>Format Control</td>
<td>1-4</td>
</tr>
<tr>
<td>1.3</td>
<td>SYMBOLS</td>
<td>1-5</td>
</tr>
<tr>
<td>1.3.1</td>
<td>Permanent Symbols</td>
<td>1-5</td>
</tr>
<tr>
<td>1.3.2</td>
<td>User-Defined Symbols</td>
<td>1-5</td>
</tr>
<tr>
<td>1.3.3</td>
<td>Direct Assignment</td>
<td>1-6</td>
</tr>
<tr>
<td>1.3.4</td>
<td>Register Symbols</td>
<td>1-6</td>
</tr>
<tr>
<td>1.4</td>
<td>EXPRESSIONS</td>
<td>1-7</td>
</tr>
<tr>
<td>1.4.1</td>
<td>Numbers</td>
<td>1-8</td>
</tr>
<tr>
<td>1.4.2</td>
<td>Arithmetic and Logical Operators</td>
<td>1-8</td>
</tr>
<tr>
<td>1.4.3</td>
<td>ASCII Conversion</td>
<td>1-8</td>
</tr>
<tr>
<td>1.4.4</td>
<td>Mode of Expressions</td>
<td>1-9</td>
</tr>
<tr>
<td>1.5</td>
<td>ASSEMBLY LOCATION COUNTER</td>
<td>1-10</td>
</tr>
<tr>
<td>1.6</td>
<td>RELOCATION AND LINKING</td>
<td>1-11</td>
</tr>
<tr>
<td>1.7</td>
<td>ADDRESSING</td>
<td>1-12</td>
</tr>
<tr>
<td>1.7.1</td>
<td>Register Mode</td>
<td>1-12</td>
</tr>
<tr>
<td>1.7.2</td>
<td>Deferred Register Mode</td>
<td>1-13</td>
</tr>
<tr>
<td>1.7.3</td>
<td>Autoincrement Mode</td>
<td>1-13</td>
</tr>
<tr>
<td>1.7.4</td>
<td>Deferred Autoincrement Mode</td>
<td>1-13</td>
</tr>
<tr>
<td>1.7.5</td>
<td>Autodecrement Mode</td>
<td>1-14</td>
</tr>
<tr>
<td>1.7.6</td>
<td>Deferred Autodecrement Mode</td>
<td>1-14</td>
</tr>
<tr>
<td>1.7.7</td>
<td>Index Mode</td>
<td>1-14</td>
</tr>
<tr>
<td>1.7.8</td>
<td>Deferred Index Mode</td>
<td>1-14</td>
</tr>
<tr>
<td>1.7.9</td>
<td>Immediate Mode and Deferred Immediate</td>
<td>1-14</td>
</tr>
<tr>
<td></td>
<td>(Absolute) Mode</td>
<td>1-15</td>
</tr>
<tr>
<td>1.7.10</td>
<td>Relative and Deferred Relative Modes</td>
<td>1-15</td>
</tr>
<tr>
<td>1.7.11</td>
<td>Table of Mode Forms and Codes (6-bit(A)</td>
<td>1-16</td>
</tr>
<tr>
<td></td>
<td>format only - see Section 1.7.12)</td>
<td></td>
</tr>
<tr>
<td>1.7.12</td>
<td>Instruction Forms</td>
<td>1-17</td>
</tr>
<tr>
<td>1.8</td>
<td>ASSEMBLER DIRECTIVES</td>
<td>1-18</td>
</tr>
<tr>
<td>1.8.1</td>
<td>.TITLE</td>
<td>1-18</td>
</tr>
<tr>
<td>1.8.2</td>
<td>.GLOBL</td>
<td>1-18</td>
</tr>
<tr>
<td>1.8.3</td>
<td>Program Section Directives (.ASECT and</td>
<td>1-19</td>
</tr>
<tr>
<td></td>
<td>.CSECT)</td>
<td></td>
</tr>
<tr>
<td>1.8.4</td>
<td>.EOT</td>
<td>1-20</td>
</tr>
<tr>
<td>1.8.5</td>
<td>.EVEN</td>
<td>1-20</td>
</tr>
<tr>
<td>1.8.6</td>
<td>.END</td>
<td>1-20</td>
</tr>
<tr>
<td>1.8.7</td>
<td>.WORD</td>
<td>1-21</td>
</tr>
<tr>
<td>1.8.8</td>
<td>.BYTE</td>
<td>1-21</td>
</tr>
<tr>
<td>1.8.9</td>
<td>.ASCII</td>
<td>1-22</td>
</tr>
<tr>
<td>1.8.10</td>
<td>.RAD50</td>
<td>1-23</td>
</tr>
<tr>
<td>1.8.11</td>
<td>.LIMIT</td>
<td>1-23</td>
</tr>
<tr>
<td>1.8.12</td>
<td>Conditional Assembly Directives</td>
<td>1-23</td>
</tr>
</tbody>
</table>

iii
CONTENTS (CONT.)

1.9 OPERATING PROCEDURES
   1.9.1 Introduction
   1.9.2 Loading PAL-11S
   1.9.3 Initial Dialogue
   1.9.4 Assembly Dialogue
   1.9.5 Assembly Listing
   1.9.6 Object Module Output
   1.9.6.1 Global Symbol Directory
   1.9.6.2 Text Block
   1.9.6.3 Relocation Directory
   1.10 ERROR CODES
   1.11 SOFTWARE ERROR HALTS

CHAPTER 2 WRITING PAL-11A ASSEMBLY LANGUAGE PROGRAMS

2.1 CHARACTER SET
2.2 STATEMENTS
   2.2.1 Label
   2.2.2 Operator
   2.2.3 Operand
   2.2.4 Comments
   2.2.5 Format Control
2.3 SYMBOLS
   2.3.1 Permanent Symbols
   2.3.2 User-Defined Symbols
   2.3.3 Direct Assignment
   2.3.4 Register Symbols
2.4 EXPRESSIONS
   2.4.1 Numbers
   2.4.2 Arithmetic and Logical Operators
   2.4.3 ASCII Conversion
2.5 ASSEMBLY LOCATION COUNTER
2.6 ADDRESSING
   2.6.1 Register Mode
   2.6.2 Deferred Register Mode
   2.6.3 Autoincrement Mode
   2.6.4 Deferred Autoincrement Mode
   2.6.5 Autodecrement Mode
   2.6.6 Deferred Autodecrement Mode
   2.6.7 Index Mode
   2.6.8 Deferred Index Mode
   2.6.9 Immediate Mode and Deferred Immediate
      (Absolute) Mode
   2.6.10 Relative and Deferred Relative Modes
   2.6.11 Table of Mode Forms and Codes (6-bit (A)
      format only - see Section 3.7)
2.7 INSTRUCTION FORMS
2.8 ASSEMBLER DIRECTIVES
   2.8.1 .EOT
   2.8.2 .EVEN
   2.8.3 .END
   2.8.4 .WORD
   2.8.5 .BYTE
   2.8.6 .ASCII
2.9 OPERATING PROCEDURES
   2.9.1 Introduction
   2.9.2 Loading PAL-11A
## CONTENTS (CONT.)

<table>
<thead>
<tr>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4-11</td>
</tr>
<tr>
<td>4-12</td>
</tr>
<tr>
<td>4-17</td>
</tr>
</tbody>
</table>

### CHAPTER 5  DEBUGGING OBJECT PROGRAMS ON-LINE  5-1

<table>
<thead>
<tr>
<th>5.1</th>
<th>INTRODUCTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1</td>
<td></td>
</tr>
<tr>
<td>5.1.1</td>
<td>ODT-11 and ODT-11X</td>
</tr>
<tr>
<td>5-1</td>
<td></td>
</tr>
<tr>
<td>5.1.2</td>
<td>ODT's Command Syntax</td>
</tr>
<tr>
<td>5-2</td>
<td></td>
</tr>
<tr>
<td>5.2</td>
<td>COMMANDS AND FUNCTIONS</td>
</tr>
<tr>
<td>5-3</td>
<td></td>
</tr>
<tr>
<td>5.2.1</td>
<td>Opening, Changing, and Closing Locations</td>
</tr>
<tr>
<td>5-4</td>
<td></td>
</tr>
<tr>
<td>5.2.1.1</td>
<td>The Slash (/)</td>
</tr>
<tr>
<td>5-4</td>
<td></td>
</tr>
<tr>
<td>5.2.1.2</td>
<td>The LINE FEED Key</td>
</tr>
<tr>
<td>5-4</td>
<td></td>
</tr>
<tr>
<td>5.2.1.3</td>
<td>The Up-Arrow (↑)</td>
</tr>
<tr>
<td>5-5</td>
<td></td>
</tr>
<tr>
<td>5.2.1.4</td>
<td>The Back-Arrow (↑)</td>
</tr>
<tr>
<td>5-5</td>
<td></td>
</tr>
<tr>
<td>5.2.1.5</td>
<td>Accessing General Registers 0-7</td>
</tr>
<tr>
<td>5-5</td>
<td></td>
</tr>
<tr>
<td>5.2.1.6</td>
<td>Accessing Internal Registers</td>
</tr>
<tr>
<td>5-6</td>
<td></td>
</tr>
<tr>
<td>5.2.2</td>
<td>Breakpoints</td>
</tr>
<tr>
<td>5-6</td>
<td></td>
</tr>
<tr>
<td>5.2.2.1</td>
<td>Setting the Breakpoint(n;B)</td>
</tr>
<tr>
<td>5-6</td>
<td></td>
</tr>
<tr>
<td>5.2.2.2</td>
<td>Locating the Breakpoint($)B</td>
</tr>
<tr>
<td>5-7</td>
<td></td>
</tr>
<tr>
<td>5.2.3</td>
<td>Running the Program(n;G and n;P)</td>
</tr>
<tr>
<td>5-7</td>
<td></td>
</tr>
<tr>
<td>5.2.4</td>
<td>Searches</td>
</tr>
<tr>
<td>5-8</td>
<td></td>
</tr>
<tr>
<td>5.2.4.1</td>
<td>Word Search(n;W)</td>
</tr>
<tr>
<td>5-8</td>
<td></td>
</tr>
<tr>
<td>5.2.4.2</td>
<td>Effective Address Search(n;E)</td>
</tr>
<tr>
<td>5-9</td>
<td></td>
</tr>
<tr>
<td>5.2.5</td>
<td>Calculating Offsets(n;O)</td>
</tr>
<tr>
<td>5-9</td>
<td></td>
</tr>
<tr>
<td>5.2.6</td>
<td>ODT's Priority Level($)P</td>
</tr>
<tr>
<td>5-10</td>
<td></td>
</tr>
<tr>
<td>5.3</td>
<td>ODT-11X</td>
</tr>
<tr>
<td>5-10</td>
<td></td>
</tr>
<tr>
<td>5.3.1</td>
<td>Opening, Changing and Closing Locations</td>
</tr>
<tr>
<td>5-10</td>
<td></td>
</tr>
<tr>
<td>5.3.1.1</td>
<td>Open the Addressed Location(@)</td>
</tr>
<tr>
<td>5-11</td>
<td></td>
</tr>
<tr>
<td>5.3.1.2</td>
<td>Relative Branch Offset(&gt;)</td>
</tr>
<tr>
<td>5-11</td>
<td></td>
</tr>
<tr>
<td>5.3.1.3</td>
<td>Return to Previous Sequence(&lt;)</td>
</tr>
<tr>
<td>5-11</td>
<td></td>
</tr>
<tr>
<td>5.3.2</td>
<td>Calculating Offsets(n;O)</td>
</tr>
<tr>
<td>5-11</td>
<td></td>
</tr>
<tr>
<td>5.3.3</td>
<td>Breakpoints</td>
</tr>
<tr>
<td>5-12</td>
<td></td>
</tr>
<tr>
<td>5.3.4</td>
<td>Single-Instruction Mode</td>
</tr>
<tr>
<td>5-12</td>
<td></td>
</tr>
<tr>
<td>5.4</td>
<td>ERROR DETECTION</td>
</tr>
<tr>
<td>5-13</td>
<td></td>
</tr>
<tr>
<td>5.5</td>
<td>PROGRAMMING CONSIDERATIONS</td>
</tr>
<tr>
<td>5-14</td>
<td></td>
</tr>
<tr>
<td>5.5.1</td>
<td>Functional Organization</td>
</tr>
<tr>
<td>5-14</td>
<td></td>
</tr>
<tr>
<td>5.5.2</td>
<td>Breakpoints</td>
</tr>
<tr>
<td>5-14</td>
<td></td>
</tr>
<tr>
<td>5.5.3</td>
<td>Search</td>
</tr>
<tr>
<td>5-19</td>
<td></td>
</tr>
<tr>
<td>5.5.4</td>
<td>Teletype Interrupt</td>
</tr>
<tr>
<td>5-19</td>
<td></td>
</tr>
<tr>
<td>5.6</td>
<td>OPERATING PROCEDURES</td>
</tr>
<tr>
<td>5-20</td>
<td></td>
</tr>
<tr>
<td>5.6.1</td>
<td>Linking Procedures (LSI-11 Systems Only)</td>
</tr>
<tr>
<td>5-20</td>
<td></td>
</tr>
<tr>
<td>5.6.2</td>
<td>Loading Procedures (non-LSI-11 Systems Only)</td>
</tr>
<tr>
<td>5-20</td>
<td></td>
</tr>
<tr>
<td>5.6.3</td>
<td>Starting and Restarting</td>
</tr>
<tr>
<td>5-21</td>
<td></td>
</tr>
</tbody>
</table>

### CHAPTER 6  LOADING AND DUMPING MEMORY  6-1

<table>
<thead>
<tr>
<th>6.1</th>
<th>PAPER TAPE BOOTSTRAPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>6-2</td>
<td></td>
</tr>
<tr>
<td>6.1.1</td>
<td>BM792-YA Paper Tape Bootstrap ROM</td>
</tr>
<tr>
<td>6-2</td>
<td></td>
</tr>
<tr>
<td>6.1.2</td>
<td>BM873-YA Bootstrap Loader ROM</td>
</tr>
<tr>
<td>6-2</td>
<td></td>
</tr>
<tr>
<td>6.1.3</td>
<td>LSI-11 Firmware Paper Tape Bootstrap</td>
</tr>
<tr>
<td>6-3</td>
<td></td>
</tr>
<tr>
<td>6.1.4</td>
<td>M9301-YB Bootstrap Loader</td>
</tr>
<tr>
<td>6-3</td>
<td></td>
</tr>
<tr>
<td>6.1.5</td>
<td>M9301-YA Bootstrap Loader</td>
</tr>
<tr>
<td>6-4</td>
<td></td>
</tr>
<tr>
<td>6.1.6</td>
<td>Other Bootstrap Loaders</td>
</tr>
<tr>
<td>6-4</td>
<td></td>
</tr>
<tr>
<td>6.1.6.1</td>
<td>Loading the Loader into Core</td>
</tr>
<tr>
<td>6-5</td>
<td></td>
</tr>
</tbody>
</table>
## CONTENTS (CONT.)

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.6.2 Loading Bootstrap Tapes</td>
<td>6-6</td>
</tr>
<tr>
<td>6.1.6.3 Bootstrap Loader Operation</td>
<td>6-8</td>
</tr>
<tr>
<td>6.2 THE ABSOLUTE LOADER</td>
<td>6-10</td>
</tr>
<tr>
<td>6.2.1 Loading the Loader into Core</td>
<td>6-11</td>
</tr>
<tr>
<td>6.2.2 Using the Absolute Loader</td>
<td>6-11</td>
</tr>
<tr>
<td>6.2.3 Absolute Loader Operation</td>
<td>6-13</td>
</tr>
<tr>
<td>6.3 CORE MEMORY DUMPS</td>
<td>6-14</td>
</tr>
<tr>
<td>6.3.1 Operating Procedures</td>
<td>6-14</td>
</tr>
<tr>
<td>6.3.1.1 Using DUMPAB on Systems without Switch Registers</td>
<td>6-15</td>
</tr>
<tr>
<td>6.3.1.2 Using DUMPAB and DUMPTT on Systems with Switch Registers</td>
<td>6-16</td>
</tr>
<tr>
<td>6.3.2 Output Formats</td>
<td>6-17</td>
</tr>
<tr>
<td>6.3.3 Storage Maps</td>
<td>6-17</td>
</tr>
</tbody>
</table>

## CHAPTER 7 INPUT/OUTPUT PROGRAMMING

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1 INTRODUCTION</td>
<td>7-1</td>
</tr>
<tr>
<td>7.1.1 Using IOX with the LSI-ll Processor</td>
<td>7-3</td>
</tr>
<tr>
<td>7.1.2 Using IOX with Unibus PDP-ll Processors</td>
<td>7-3</td>
</tr>
<tr>
<td>7.1.3 IOX Interrupt and Trap Vectors</td>
<td>7-3</td>
</tr>
<tr>
<td>7.2 THE DEVICE ASSIGNMENT TABLE</td>
<td>7-3</td>
</tr>
<tr>
<td>7.2.1 Reset</td>
<td>7-4</td>
</tr>
<tr>
<td>7.2.2 Initialization</td>
<td>7-4</td>
</tr>
<tr>
<td>7.3 BUFFER ARRANGEMENT IN DATA TRANSFER COMMANDS</td>
<td>7-4</td>
</tr>
<tr>
<td>7.3.1 Buffer Size</td>
<td>7-5</td>
</tr>
<tr>
<td>7.3.2 Mode Byte</td>
<td>7-5</td>
</tr>
<tr>
<td>7.3.3 Status Byte</td>
<td>7-6</td>
</tr>
<tr>
<td>7.3.3.1 Non-Fatal Error Codes (Octal)</td>
<td>7-6</td>
</tr>
<tr>
<td>7.3.3.2 Done Bit</td>
<td>7-7</td>
</tr>
<tr>
<td>7.3.3.3 End-Of-Medium Bit</td>
<td>7-7</td>
</tr>
<tr>
<td>7.3.3.4 End-Of-File Bit</td>
<td>7-7</td>
</tr>
<tr>
<td>7.3.4 Byte Count</td>
<td>7-8</td>
</tr>
<tr>
<td>7.4 MODES</td>
<td>7-8</td>
</tr>
<tr>
<td>7.4.1 Formatted ASCII</td>
<td>7-8</td>
</tr>
<tr>
<td>7.4.2 Unformatted ASCII</td>
<td>7-10</td>
</tr>
<tr>
<td>7.4.3 Formatted Binary</td>
<td>7-10</td>
</tr>
<tr>
<td>7.4.4 Unformatted Binary</td>
<td>7-11</td>
</tr>
<tr>
<td>7.5 DATA TRANSFERS</td>
<td>7-11</td>
</tr>
<tr>
<td>7.5.1 Read</td>
<td>7-11</td>
</tr>
<tr>
<td>7.5.2 Write</td>
<td>7-12</td>
</tr>
<tr>
<td>7.5.3 Device Conflicts In Data Transfer Commands</td>
<td>7-12</td>
</tr>
<tr>
<td>7.5.4 Waitr (Wait, Return)</td>
<td>7-13</td>
</tr>
<tr>
<td>7.5.5 Waitr vs. Testing the Buffer Done Bit</td>
<td>7-13</td>
</tr>
<tr>
<td>7.5.6 Single Buffer Transfer on One Device</td>
<td>7-14</td>
</tr>
<tr>
<td>7.5.7 Double Buffering</td>
<td>7-15</td>
</tr>
<tr>
<td>7.5.8 Readr (Real-time Read)</td>
<td>7-15</td>
</tr>
<tr>
<td>7.5.9 Writr (Real-time Write)</td>
<td>7-16</td>
</tr>
<tr>
<td>7.6 REENABLE THE READER AND RESTARTING</td>
<td>7-16</td>
</tr>
<tr>
<td>7.6.1 Seek</td>
<td>7-16</td>
</tr>
<tr>
<td>7.6.2 Restart</td>
<td>7-16</td>
</tr>
<tr>
<td>7.7 FATAL ERRORS</td>
<td>7-17</td>
</tr>
<tr>
<td>7.8 EXAMPLE OF PROGRAM USING IOX</td>
<td>7-17</td>
</tr>
<tr>
<td>7.9 IOX INTERNAL INFORMATION</td>
<td>7-19</td>
</tr>
<tr>
<td>7.9.1 Conflict Byte/Word</td>
<td>7-19</td>
</tr>
<tr>
<td>7.9.2 Device Interrupt Table (DIT)</td>
<td>7-20</td>
</tr>
<tr>
<td>7.9.3 Device Status Table (DST)</td>
<td>7-21</td>
</tr>
</tbody>
</table>
## CONTENTS (CONT.)

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.9.4</td>
<td>Teletype Hardware Tab Facility</td>
<td>7-21</td>
</tr>
<tr>
<td>7.9.5</td>
<td>Adding Devices To IOX</td>
<td>7-21</td>
</tr>
<tr>
<td>7.9.5.1</td>
<td>Device Codes</td>
<td>7-21</td>
</tr>
<tr>
<td>7.9.5.2</td>
<td>Table Modification</td>
<td>7-22</td>
</tr>
<tr>
<td>7.9.5.3</td>
<td>Interrupt Routines</td>
<td>7-23</td>
</tr>
</tbody>
</table>

### CHAPTER 8

**FLOATING POINT MATH PACKAGE OVERVIEW**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>

### CHAPTER 9

**PROGRAMMING TECHNIQUES**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>9.1</td>
<td>WRITING POSITION INDEPENDENT CODE</td>
<td>9-1</td>
</tr>
<tr>
<td>9.1.1</td>
<td>Position Independent Modes</td>
<td>9-2</td>
</tr>
<tr>
<td>9.1.2</td>
<td>Absolute Modes</td>
<td>9-3</td>
</tr>
<tr>
<td>9.1.3</td>
<td>Writing Automatic PIC</td>
<td>9-3</td>
</tr>
<tr>
<td>9.1.4</td>
<td>Writing Non-Automatic PIC</td>
<td>9-4</td>
</tr>
<tr>
<td>9.1.4.1</td>
<td>Setting Up The Stack Pointer</td>
<td>9-4</td>
</tr>
<tr>
<td>9.1.4.2</td>
<td>Setting Up A Trap or Interrupt Vector</td>
<td>9-4</td>
</tr>
<tr>
<td>9.1.4.3</td>
<td>Relocating Pointers</td>
<td>9-5</td>
</tr>
<tr>
<td>9.2</td>
<td>LOADING UNUSED TRAP VECTORS</td>
<td>9-5</td>
</tr>
<tr>
<td>9.3</td>
<td>CODING TECHNIQUES</td>
<td>9-6</td>
</tr>
<tr>
<td>9.3.1</td>
<td>Altering Register Contents</td>
<td>9-6</td>
</tr>
<tr>
<td>9.3.2</td>
<td>Subroutines</td>
<td>9-7</td>
</tr>
</tbody>
</table>

### APPENDIX A

**ASCII CHARACTER SET**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
</table>

### APPENDIX B

**PAL-11S ASSEMBLY LANGUAGE AND ASSEMBLER**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>B.1</td>
<td>TERMINATORS</td>
<td>B-1</td>
</tr>
<tr>
<td>B.2</td>
<td>ADDRESS MODE SYNTAX</td>
<td>B-2</td>
</tr>
<tr>
<td>B.3</td>
<td>INSTRUCTIONS</td>
<td>B-2</td>
</tr>
<tr>
<td>B.3.1</td>
<td>Double Operand Instructions OP A,A</td>
<td>B-4</td>
</tr>
<tr>
<td>B.3.2</td>
<td>Single Operand Instructions OP A</td>
<td>B-4</td>
</tr>
<tr>
<td>B.3.3</td>
<td>Rotate/Shift</td>
<td>B-5</td>
</tr>
<tr>
<td>B.3.4</td>
<td>Operation Instructions OP</td>
<td>B-5</td>
</tr>
<tr>
<td>B.3.5</td>
<td>Branch Instructions Op E Where</td>
<td>B-6</td>
</tr>
<tr>
<td></td>
<td>[-128_{10} \leq (E_{-2})/2 &lt; 127_{10}]</td>
<td></td>
</tr>
<tr>
<td>B.3.6</td>
<td>Subroutine Call JSR ER,A</td>
<td>B-7</td>
</tr>
<tr>
<td>B.3.7</td>
<td>Subroutine Return</td>
<td>B-7</td>
</tr>
<tr>
<td>B.3.8</td>
<td>Extensions for the LSI-11 Version Of PAL-11S</td>
<td>B-7</td>
</tr>
<tr>
<td>B.4</td>
<td>ASSEMBLER DIRECTIVES</td>
<td>B-8</td>
</tr>
<tr>
<td>B.5</td>
<td>ERROR CODES</td>
<td>B-10</td>
</tr>
<tr>
<td>B.6</td>
<td>INITIAL OPERATING PROCEDURES</td>
<td>B-11</td>
</tr>
</tbody>
</table>

### APPENDIX C

**PAL-11A ASSEMBLY LANGUAGE AND ASSEMBLER**

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.1</td>
<td>SPECIAL CHARACTERS</td>
<td>C-1</td>
</tr>
<tr>
<td>C.2</td>
<td>ADDRESS MODE SYNTAX</td>
<td>C-2</td>
</tr>
<tr>
<td>C.3</td>
<td>INSTRUCTIONS</td>
<td>C-3</td>
</tr>
<tr>
<td>C.3.1</td>
<td>Double-Operand Instructions OP A,A</td>
<td>C-4</td>
</tr>
<tr>
<td>C.3.2</td>
<td>Single-Operand Instructions OP A</td>
<td>C-4</td>
</tr>
<tr>
<td>C.3.3</td>
<td>Rotate/Shift Instructions OP A</td>
<td>C-5</td>
</tr>
<tr>
<td>C.3.4</td>
<td>Operate Instructions Op</td>
<td>C-5</td>
</tr>
<tr>
<td>C.3.5</td>
<td>Trap Instructions Op or Op E Where (0 \leq E &lt; 377_{8})</td>
<td>C-6</td>
</tr>
<tr>
<td>C.3.6</td>
<td>Branch Instructions Op E where</td>
<td>C-7</td>
</tr>
<tr>
<td></td>
<td>(-128_{10} \leq (E_{-2})/2 &lt; 127_{10})</td>
<td></td>
</tr>
<tr>
<td>CONTENTS (CONT.)</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>C.3.7 Subroutine Call Op ER, A</td>
<td>C-7</td>
<td></td>
</tr>
<tr>
<td>C.3.8 Subroutine Return Op ER</td>
<td>C-8</td>
<td></td>
</tr>
<tr>
<td>C.4 ASSEMBLER DIRECTIVES</td>
<td>C-8</td>
<td></td>
</tr>
<tr>
<td>C.5 ERROR CODES</td>
<td>C-8</td>
<td></td>
</tr>
<tr>
<td>C.6 INITIAL OPERATING PROCEDURES</td>
<td>C-9</td>
<td></td>
</tr>
<tr>
<td>APPENDIX D TEXT EDITOR, ED-11</td>
<td>D-1</td>
<td></td>
</tr>
<tr>
<td>D.1 INPUT/OUTPUT COMMANDS</td>
<td>D-1</td>
<td></td>
</tr>
<tr>
<td>D.2 POINTER-POSITIONING COMMANDS</td>
<td>D-2</td>
<td></td>
</tr>
<tr>
<td>D.3 SEARCH COMMANDS</td>
<td>D-2</td>
<td></td>
</tr>
<tr>
<td>D.4 COMMANDS TO MODIFY THE TEXT</td>
<td>D-3</td>
<td></td>
</tr>
<tr>
<td>D.5 SYMBOLS</td>
<td>D-3</td>
<td></td>
</tr>
<tr>
<td>D.6 GROUPING OF COMMANDS</td>
<td>D-4</td>
<td></td>
</tr>
<tr>
<td>D.7 OPERATING PROCEDURES</td>
<td>D-4</td>
<td></td>
</tr>
<tr>
<td>D.7.1 Loading</td>
<td>D-4</td>
<td></td>
</tr>
<tr>
<td>D.7.2 Storage Requirements</td>
<td>D-4</td>
<td></td>
</tr>
<tr>
<td>D.7.3 Starting</td>
<td>D-4</td>
<td></td>
</tr>
<tr>
<td>D.7.4 Initial Dialogue</td>
<td>D-4</td>
<td></td>
</tr>
<tr>
<td>D.7.5 Restarting</td>
<td>D-4</td>
<td></td>
</tr>
<tr>
<td>APPENDIX E DEBUGGING OBJECT PROGRAMS ON-LINE</td>
<td>E-1</td>
<td></td>
</tr>
<tr>
<td>AND ODT-11 AND ODT-11X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E.1 SUMMARY OF CONTENTS</td>
<td>E-1</td>
<td></td>
</tr>
<tr>
<td>APPENDIX F LOADING AND DUMPING CORE MEMORY</td>
<td>F-1</td>
<td></td>
</tr>
<tr>
<td>F.1 THE BOOTSTRAP LOADER</td>
<td>F-1</td>
<td></td>
</tr>
<tr>
<td>F.1.1 Loading The Bootstrap Loader</td>
<td>F-1</td>
<td></td>
</tr>
<tr>
<td>F.2 THE ABSOLUTE LOADER</td>
<td>F-3</td>
<td></td>
</tr>
<tr>
<td>F.3 CORE MEMORY DUMPS</td>
<td>F-4</td>
<td></td>
</tr>
<tr>
<td>APPENDIX G INPUT/OUTPUT PROGRAMMING, IOX</td>
<td>G-1</td>
<td></td>
</tr>
<tr>
<td>G.1 INSTRUCTION SUMMARY</td>
<td>G-1</td>
<td></td>
</tr>
<tr>
<td>G.2 PROGRAM FLOW SUMMARY</td>
<td>G-1</td>
<td></td>
</tr>
<tr>
<td>G.3 FATAL ERRORS</td>
<td>G-2</td>
<td></td>
</tr>
<tr>
<td>APPENDIX H SUMMARY OF FLOATING POINT MATH</td>
<td>H-1</td>
<td></td>
</tr>
<tr>
<td>PACKAGE, FPMP-11</td>
<td></td>
<td></td>
</tr>
<tr>
<td>H.1 OTS ROUTINES</td>
<td>H-2</td>
<td></td>
</tr>
<tr>
<td>H.2 NON-OTS ROUTINES</td>
<td>H-7</td>
<td></td>
</tr>
<tr>
<td>H.3 ROUTINES ACCESSED VIA TRAP HANDLER</td>
<td>H-7</td>
<td></td>
</tr>
<tr>
<td>APPENDIX I TAPE DUPLICATION</td>
<td>I-1</td>
<td></td>
</tr>
<tr>
<td>APPENDIX J ASSEMBLY AND LINKING INSTRUCTIONS</td>
<td>J-1</td>
<td></td>
</tr>
<tr>
<td>J.1 SYSTEMS WITHOUT SWITCH REGISTERS</td>
<td>J-1</td>
<td></td>
</tr>
<tr>
<td>J.1.1 IOX/IOXLPT</td>
<td>J-1</td>
<td></td>
</tr>
<tr>
<td>J.1.1.1 Assembling IOX</td>
<td>J-1</td>
<td></td>
</tr>
<tr>
<td>J.1.1.2 Assembling IOXLPT</td>
<td>J-1</td>
<td></td>
</tr>
<tr>
<td>J.1.1.3 Linking IOX and IOXLPT</td>
<td>J-1</td>
<td></td>
</tr>
<tr>
<td>J.1.2 ODT11X</td>
<td>J-1</td>
<td></td>
</tr>
</tbody>
</table>
CONTENTS (CONT.)

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>J.1.2.1</td>
<td>Assembling ODT11X</td>
<td>J-1</td>
</tr>
<tr>
<td>J.1.2.2</td>
<td>Linking ODT11X</td>
<td>J-2</td>
</tr>
<tr>
<td>J.1.3</td>
<td>ED-11</td>
<td>J-2</td>
</tr>
<tr>
<td>J.1.3.1</td>
<td>Assembling ED-11</td>
<td>J-2</td>
</tr>
<tr>
<td>J.1.3.2</td>
<td>Linking ED-11</td>
<td>J-2</td>
</tr>
<tr>
<td>J.1.4</td>
<td>PAL-11S</td>
<td>J-2</td>
</tr>
<tr>
<td>J.1.4.1</td>
<td>Assembling PAL-11S</td>
<td>J-2</td>
</tr>
<tr>
<td>J.1.4.2</td>
<td>Linking PAL-11S</td>
<td>J-3</td>
</tr>
<tr>
<td>J.1.5</td>
<td>LINK-11S</td>
<td>J-3</td>
</tr>
<tr>
<td>J.1.5.1</td>
<td>Assembling LINK-11S</td>
<td>J-4</td>
</tr>
<tr>
<td>J.1.5.2</td>
<td>Linking LINK-11S</td>
<td>J-4</td>
</tr>
<tr>
<td>J.2</td>
<td>SYSTEMS WITH SWITCH REGISTERS</td>
<td>J-5</td>
</tr>
<tr>
<td>J.2.1</td>
<td>Assembling PAL-11A</td>
<td>J-5</td>
</tr>
<tr>
<td>J.2.2</td>
<td>Assembling ED-11</td>
<td>J-6</td>
</tr>
<tr>
<td>J.2.3</td>
<td>ODT-11/ODT-11X</td>
<td>J-6</td>
</tr>
<tr>
<td>J.2.4</td>
<td>Assembling IOX/IOXLPT</td>
<td>J-7</td>
</tr>
<tr>
<td>J.2.5</td>
<td>Assembling and Linking PAL-11S</td>
<td>J-8</td>
</tr>
<tr>
<td>J.2.6</td>
<td>Assembling and Linking LINK-11S</td>
<td>J-11</td>
</tr>
</tbody>
</table>

APPENDIX K

STANDARD PDP-11 ABBREVIATIONS

K-1

APPENDIX L

CONVERSION TABLES

L-1

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>L.1</td>
<td>OCTAL-DECIMAL INTEGER CONVERSIONS</td>
<td>L-1</td>
</tr>
<tr>
<td>L.2</td>
<td>POWERS OF TWO</td>
<td>L-5</td>
</tr>
<tr>
<td>L.3</td>
<td>SCALES OF NOTATION</td>
<td>L-6</td>
</tr>
<tr>
<td>L.3.1</td>
<td>$2^x$ In Decimal</td>
<td>L-6</td>
</tr>
<tr>
<td>L.3.2</td>
<td>$10^n$ In Octal</td>
<td>L-6</td>
</tr>
<tr>
<td>L.3.3</td>
<td>n Log 2 and 10 In Decimal</td>
<td>L-6</td>
</tr>
<tr>
<td>L.3.4</td>
<td>Addition and Multiplication, Binary and Octal</td>
<td>L-6</td>
</tr>
<tr>
<td>L.3.5</td>
<td>Mathematical Constants In Octal</td>
<td>L-7</td>
</tr>
</tbody>
</table>

APPENDIX M

NOTE TO USERS OF SERIAL LA30 AND 600, 1200, AND 2400 BAUD VT05'S

M-1

APPENDIX N

USING THE ABSOLUTE LOADER ON PDP-11'S WITHOUT SWITCH REGISTERS

N-1

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>N.1</td>
<td>LSI-11</td>
<td>N-1</td>
</tr>
<tr>
<td>N.2</td>
<td>M9301-YB BOOTSTRAP LOADER</td>
<td>N-3</td>
</tr>
<tr>
<td>N.3</td>
<td>M9301-YA BOOTSTRAP LOADER</td>
<td>N-4</td>
</tr>
</tbody>
</table>

INDEX

Index-1
CONTENTS (CONT.)

FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-1</td>
<td>Communication and Data Flow</td>
</tr>
<tr>
<td>6-1</td>
<td>Bootstrap Loader Instructions</td>
</tr>
<tr>
<td>6-2</td>
<td>Loading and Verifying the Bootstrap Loader</td>
</tr>
<tr>
<td>6-3</td>
<td>Loading Bootstrap Tapes Into Core</td>
</tr>
<tr>
<td>6-4</td>
<td>The Bootstrap Loader Program</td>
</tr>
<tr>
<td>F-1</td>
<td>Loading and Verifying the Bootstrap Loader</td>
</tr>
<tr>
<td>F-2</td>
<td>Loading Bootstrap Tapes into Core</td>
</tr>
<tr>
<td>F-3</td>
<td>Loading with the Absolute Loader</td>
</tr>
<tr>
<td>F-4</td>
<td>Dumping Using DUMPAB or DUMPTT</td>
</tr>
</tbody>
</table>

TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Instruction Operand Fields</td>
</tr>
<tr>
<td>2-1</td>
<td>Instruction Operand Fields</td>
</tr>
</tbody>
</table>
CHAPTER 1
PAL-11S ASSEMBLY LANGUAGE AND ASSEMBLER

PAL-11S Assembly (Program Assembly Language for the PDP-11, Relocatable, Stand Alone Version) enables you to write source (symbolic) programs using letters, numbers, and symbols which are meaningful to you. The source programs, generated either on-line using the Text Editor (ED-11), or off-line, are then assembled into object modules which are processed by the PDP-11 linker, LINK-11S. LINK-11S produces a load module which is loaded by the Absolute Loader for execution. Object modules may contain absolute and/or relocatable code and separately assembled object modules may be linked with global symbols. The object module is produced after two passes through the Assembler; an optional third pass produces a complete octal/symbolic listing of the assembled program. This listing is especially useful for documentation and debugging purposes.

This chapter not only explains how to write PAL-11S programs but also how to assemble the source programs into object modules. All facets of the assembly language are explained and illustrated with many examples, and the chapter concludes with assembling procedures. In explaining how to write PAL-11S source programs, it is necessary, especially at the outset, to make frequent forward references. Therefore, we recommend that you first read through the entire chapter to get a "feel" for the language, and then reread the chapter, this time referring to appropriate sections as indicated, for a thorough understanding of the language and assembling procedures.

Some notable features of PAL-11S are:

1. Selective assembly pass functions.
2. Device specification for pass functions.
3. Optional error listing on the teleprinter.
4. Double buffered and concurrent I/O (provided by IOXLPT).
5. Alphabetized, formatted symbol table listing.
6. Relocatable object modules.
7. Global symbols for linking between object modules.

The PAL-11S Assembler requires 8K of memory and provides for about 900 user-defined symbols (see Section 1.3.2). In addition, it allows a line printer to be used for program listing and/or symbol table listing.
The following discussion of the PAL-11S Assembly Language assumes that
you have read the PDP-11 Processor Handbook with emphasis on those
sections which deal with the PDP-11 instruction repertoire, formats,
and timings -- a thorough knowledge of these is vital to efficient
assembly language programming.

1.1 CHARACTER SET
A PAL-11S source program is composed of symbols, numbers, expressions,
symbolic instructions, assembler directives, argument separators, and
line terminators written using the following ASCII\(^1\) characters.

1. The letters A through Z. (Upper and lower case letters are
   acceptable, although upon input, lower case letters will be
   converted to upper case letters.)

2. The numbers 0 through 9.

3. The characters . and $. (These characters are reserved for
   systems use.)

4. The separating or terminating symbols:
   
   
   
   carriage return tab space line feed form feed

1.2 STATEMENTS
A source program is composed of a sequence of statements, where each
statement is on a single line. The statement is terminated by a
 carriage return character which must be immediately followed by either
a line feed or form feed character. Should a carriage return character be present
and not be followed by a line feed or form feed, the Assembler will generate
a Q error (Section 1.10), and that portion of the line following the carriage return
will be ignored. Since the carriage return terminator is a required statement terminator, a line
feed or form feed not immediately preceded by a carriage return will have one inserted by the Assembler.

It should be noted that, if the Editor (ED-11) is being used to create
the source program, a typed carriage return (RETURN key) automatically
generates a line feed character.

A statement may be composed of up to four fields which are identified
by their order of appearance and by specified terminating characters
as explained below and summarized in Appendix B. The four fields are:

Label Operator Operand Comment

The label and comment fields are optional. The operator and operand
fields are inter-dependent -- either may be omitted depending upon the
contents of the other.

\(^1\)ASCII stands for American Standard Code for Information Interchange.
1.2.1 Label

A label is a user-defined symbol (see Section 1.3.2) which is assigned the value of the current location counter. This value may be either absolute or relocatable depending on whether the location counter value is absolute or relocatable. In the latter case, the final absolute value is assigned by the Linker, i.e., the value + the relocation constant. A label is a symbolic means of referring to a specific location within a program. If present, a label always occurs first in a statement and must be terminated by a colon. For example, if the current location is absolute 100 the statement:

    ABCD:    MOV A,B

will assign the value 100 to the label ABCD so that subsequent reference to ABCD will be to location 100. In the above case if the location counter were relocatable then the final value of ABCD would be 100 + K, where K is the location of the beginning of the relocatable section in which the label ABCD appears. More than one label may appear within a single label field; each label within the field will have the same value. For example, if the current location counter is 100, multiple labels in the statement:

    ABC: $DD: A7.7: MOV A,B

will equate each of the three labels ABC, $DD, and A7.7 with the value 100 ($ and . are reserved for system software).

The error code M (multiple definition of a symbol) will be generated during assembly if two or more labels have the same first six characters.

1.2.2 Operator

An operator follows the label field in a statement, and may be an instruction mnemonic or an assembler directive (see Section 1.8 and Appendix B). When it is an instruction mnemonic, it specifies what action is to be performed on any operand(s) which follows it. When it is an assembler directive, it specifies a certain function or action to be performed during assembly.

The operator may be preceded only by one or more labels and may be followed by one or more operands and/or a comment. An operator is legally terminated by a space, tab, or any of the following characters:

    # + - @ ( " ' % ! & , ;

        line feed    form feed    carriage return

The use of each character above will be explained in this chapter.

Consider the following examples:

    MOV A,B ; (TAB) terminates operator MOV
    MOV@A,B ; @ terminates operator MOV

When the operator stands alone without an operand or comment, it is terminated by a carriage return followed by a line feed or form feed character.
1.2.3 Operand

An operand is that part of a statement which is operated on by the operator -- an instruction mnemonic or assembler directive. Operands may be symbols, expressions, or numbers. When multiple operands appear within a statement, each is separated from the next by a comma. An operand may be preceded by an operator and/or label, and followed by a comment.

The operand field is terminated by a semicolon when followed by a comment, or by a carriage return followed by a line feed or form feed character when the operand ends the statement. For example,

```
LABEL: MOV GEORGE,BOB ;THIS IS A COMMENT
```

where the space between MOV and GEORGE terminated the operator field and began the operand field; the comma separated the operands GEORGE and BOB; the semicolon terminated the operand field and began the comment.

1.2.4 Comments

The comment field is optional and may contain any ASCII character except null, rubber, carriage return, line feed or form feed. All other characters, even those with special significance are ignored by the Assembler when used in the comment field.

The comment field may be preceded by none, any, or all of the other three fields. It must begin with the semicolon and end with a carriage return followed by a line feed or form feed character. For example,

```
LABEL: CLR HERE ;THIS IS A $1.00 COMMENT
```

Comments do not affect assembly processing or program execution, but they are useful in program listings for later analysis, checkout or documentation purposes.

1.2.5 Format Control

The format is controlled by the space and tab characters. They have no effect on the assembling process of the source program unless they are embedded within a symbol, number, or ASCII text; or are used as the operator field terminator. Thus, they can be used to provide a neat, readable program. A statement can be written:

```
LABEL: MOV(SP)+,TAG;POP VALUE OFF STACK
```

or, using formatting characters it can be written:

```
LABEL: MOV (SP)+,TAG ;POP VALUE OFF STACK
```

which is much easier to read.

Page size is controlled by the form feed character. A page of n lines is created by inserting a form feed (CTRL/FORM keys on the keyboard) after the nth line. If no form feed is present, a page is automatically terminated after 56 lines.
1.3 SYMBOLS

There are two types of symbols, permanent and user-defined. Both are stored in the Assembler's symbol table. Initially, the symbol table contains the permanent symbols, but as the source program is assembled, user-defined symbols are added to the table.

1.3.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics (see Appendix B.3) and assembler directives (see Section 1.8). These symbols are a permanent part of the Assembler's symbol table and need not be defined before being used in the source program.

1.3.2 User-Defined Symbols

User-defined symbols are those defined as labels (see Section 1.2.1) or by direct assignment (see Section 1.3.3). These symbols are added to the symbol table as they are encountered during the first pass of the assembly. They can be composed of alphanumeric characters, dollar signs, and periods only; again $'s and .'s are reserved for system software. Any other character is illegal and, if used, will result in the error message I or QU (see Section 1.10). I is a low priority error which may be flagged as QU first. The following rules also apply to user-defined symbols:

1. The first character must not be a number.

2. Each symbol must be unique within the first six characters.

3. A symbol may be written with more than six legal characters but the seventh and subsequent characters are only checked for legality, and are not otherwise recognized by the Assembler.

4. Spaces and tabs must not be embedded within a symbol.

A user-defined symbol may duplicate a permanent symbol. The value associated with a permanent symbol that is also user-defined depends upon its use:

1. A permanent symbol encountered in the operator field is associated with its corresponding machine op-code.

2. If a permanent symbol in the operand field is also user-defined, its user-defined value is associated with the symbol. If the symbol is not found to be user-defined, then the corresponding machine op-code value is associated with the symbol.

User-defined symbols are either internal or global. All symbols are internal unless they are explicitly typed as global with the .GLOBL assembler directive (see Section 1.8.2). Global symbols are used to provide links between object modules. A global symbol which is defined (as a label or by direct assignment) in a program is called an entry symbol or entry point. Such symbols may be referred to from other object modules or assemblies. A global symbol which is not defined in the current assembly is called an external symbol. Some other assembly must define the same symbol as an entry point.

1-5
1.3.3 Direct Assignment

A direct assignment statement associates a symbol with a value. When a direct assignment statement defines a symbol for the first time, that symbol is entered into the Assembler's symbol table and the specified value is associated with it. A symbol may be redefined by assigning a new value to a previously defined symbol. The newly assigned value will replace the previous value assigned to the symbol.

The symbol takes on the relocatable or absolute attribute of the defining expression. However, if the defining expression is global, the defined symbol will not be global unless previously defined as such (see Section 1.4).

The general format for a direct assignment statement is:

    symbol = expression.

The following conventions apply:

1. An equal sign (=) must separate the symbol from the expression defining the symbol.

2. A direct assignment statement may be preceded by a label and may be followed by a comment.

3. Only one symbol can be defined by any one direct assignment statement.

4. Only one level of forward referencing is allowed.

Example of two levels of forward referencing (illegal):

    X = Y
    Y = Z
    Z = 1

X and Y are both undefined throughout pass 1 and will be listed on the teleprinter as such at the end of that pass. X is undefined throughout pass 2, and will cause a U error message.

Examples:

    A=1                   ;THE SYMBOL A IS EQUATED WITH THE VALUE 1
    B='A-1&MASKLOW       ;THE SYMBOL B IS EQUATED WITH THE EXPRESSION'S
                          ;VALUE
    C: D=3               ;THE SYMBOL D IS EQUATED WITH 3. THE
    E: MOV #1,ABLE       ;LABELS C AND E ARE EQUATED WITH THE
                          ;NUMERICAL MEMORY ADDRESS OF THE MOV
                          ;COMMAND

1.3.4 Register Symbols

The eight general registers of the PDP-11 are numbered 0 through 7. These registers may be referenced by use of a register symbol; that is, a symbolic name for a register. A register symbol is defined by means of a direct assignment, where the defining expression contains at least one term preceded by a $ or at least one term previously defined as a register symbol. In addition, the defining expression of a register symbol must be absolute. For example:

1-6
PAL-11S ASSEMBLY LANGUAGE AND ASSEMBLER

R0=%0 ;DEFINE R0 AS REGISTER 0
R3=R0+3 ;DEFINED R3 AS REGISTER 3
R4=1+R3 ;DEFINE R4 AS REGISTER 4
THERE=%2 ;DEFINE "THERE" AS REGISTER 2

It is important to note that all register symbols must be defined before they are referenced. A forward reference to a register symbol will generally cause phase errors (see Section 1.10).

The % may be used in any expression thereby indicating a reference to a register. Such an expression is a register expression. Thus, the statement:

CLR %6

will clear register 6 while the statement:

CLR 6

will clear the word at memory address 6. In certain cases a register can be referenced without the use of a register symbol or register expression. These cases are recognized through the context of the statement and are thoroughly explained in Sections 1.7.11 and 1.7.12. Two obvious examples of this are:

JSR 5, SUBR ;THE FIRST OPERAND FIELD MUST ALWAYS
;BE A REGISTER

CLR X(2) ;ANY EXPRESSION ENCLOSED IN ( ) MUST BE
;A REGISTER. IN THIS CASE, INDEX REGISTER
;2

1.4 EXPRESSIONS

Arithmetic and logical operators (see Section 1.4.2) may be used to form expressions. A term of an expression may be a permanent or user-defined symbol (which may be absolute, relocatable or global), a number, ASCII data, or the present value of the assembly location counter represented by the period (see Section 1.5). Expressions are evaluated from left to right. Parenthetical grouping is not allowed.

Expressions are evaluated as word quantities. The operands of a .BYTE directive (Section 1.8.8) are evaluated as word expressions before truncation to the low-order eight bits. The evaluation of an expression includes the evaluation of the mode of the resultant expression; that is, absolute, relocatable or external. The definition of the modes of expression are given below in Section 1.4.4.

A missing term, expression or external symbol will be interpreted as 0. A missing operator will be interpreted as +. The error code Q (Questionable syntax) will be generated for a missing operator. For example,

A + -100 ;OPERAND MISSING

will be evaluated as A + 0 - 100, and

TAG ! LA 177777 ;OPERATOR MISSING

will be evaluated as TAG ! LA+177777.

1-7
The value of an external expression will be the value of the absolute part of the expression; e.g., EXT+A will have a value of A. This will be modified by the linker to become EXT+A.

1.4.1 Numbers

The Assembler accepts both octal and decimal numbers. Octal numbers consist of the digits 0 through 7 only. Decimal numbers consist of the digits 0 through 9 followed by a decimal point. If a number contains an 8 or 9 and is not followed by a decimal point, the N error code (see Section 1.10) will be printed and the number will be interpreted as decimal. Negative numbers may be expressed as a number preceded by a minus sign rather than in a two's complement form. Positive numbers may be preceded by a plus sign although this is not required.

If a number is too large to fit into 16 bits, the number is truncated from the left. In the assembly listing the statement will be flagged with a Truncation (T) error. Numbers are always considered to be absolute quantities (that is, not relocatable).

1.4.2 Arithmetic And Logical Operators

The arithmetic operators are:

+ indicates addition or a positive number

- indicates subtraction or a negative number

The logical operators are:

& indicates the logical AND operation

! indicates the logical inclusive OR operation

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &amp; 0 = 0</td>
<td>0 ! 0 = 0</td>
</tr>
<tr>
<td>0 &amp; 1 = 0</td>
<td>0 ! 1 = 1</td>
</tr>
<tr>
<td>1 &amp; 0 = 0</td>
<td>1 ! 0 = 1</td>
</tr>
<tr>
<td>1 &amp; 1 = 1</td>
<td>1 ! 1 = 1</td>
</tr>
</tbody>
</table>

1.4.3 ASCII Conversion

When preceded by an apostrophe, any ASCII character (except null, rubout, carriage return, line feed, or form feed) is assigned the 7-bit ASCII value of the character (see Appendix A). For example,

'A

is assigned the value 1010.

When preceded by a quotation mark, two ASCII characters (not including null, rubout, carriage return, line feed, or form feed) are assigned the 7-bit ASCII values of each of the characters to be used. Each 7-bit value is stored in an 8-bit byte and the bytes are combined to form a word. For example "AB will store the ASCII value of A in the low-order (even) byte and the value of B in the high-order (odd) byte:
1.4.4 Mode of Expressions

The mode of an expression may be absolute, relocatable or external as defined below:

A term of an expression is absolute, relocatable or external depending on whether its definer (i.e., number, symbol, etc.) is absolute, relocatable or external. Numbers, permanent symbols and generated data are always treated as absolute.

An absolute expression is defined as:

1. Absolute term (one whose value is defined at assembly time) preceded optionally by a single plus or minus sign, or

2. Relocatable expression minus a relocatable term, or

3. Absolute expression followed by an operator followed by an absolute expression.

A relocatable expression is defined as:

1. Relocatable term (one whose value is not known until link time), or

2. Relocatable expression followed by an arithmetic operator followed by an absolute expression, or

3. Absolute expression followed by a plus operator followed by a relocatable expression.

An external expression is defined as:

1. External term (one whose value is defined outside the program), or

2. External expression followed by an arithmetic operator followed by an absolute term, or

3. Absolute expression followed by a plus operator followed by an external expression.

In the following examples:

ABS is an absolute symbol,

REL is a relocatable symbol,

EXT is an external symbol.
Examples:

The following are valid expressions:

- `EXT + ABS` ; External expression
- `REL+REL-REL` ; Relocatable expression
- `ABS+REL-REL & ABS` ; Absolute expression

The following are illegal expressions:

- `EXT+REL`
- `REL+REL`
- `ABS-EXT`

1.5 ASSEMBLY LOCATION COUNTER

The period (.) is the symbol for the assembly location counter. (Note difference of Program Counter. #PC. See Section 1.7.) When used in the operand field of an instruction, it represents the address of the first word of the instruction. When used in the operand field of an assembler directive, it represents the address of the current byte or word. For example,

```
A: MOV #.,R0 ; refers to location A, ;i.e., the address of the ;MOV instruction
```

(# is explained in Section 1.7.9.)

At the beginning of each assembly pass, the Assembler clears the location counter. Normally, consecutive memory locations are assigned to each byte of object data generated. However, the location where the object data is stored may be changed by a direct assignment altering the location counter:

```
=expression
```

Similar to other symbols, the location counter symbol "." has a mode associated with it. However, the mode cannot be external. Neither can one change the existing mode of the location counter by using a defining expression of a different mode.

The mode of the location counter symbol can be changed by the use of the .ASECT or .CSECT directive as explained in Section 1.8.3.

The expression defining the location counter must not contain forward references or symbols that vary from one pass to another.

Examples:

```
.ASECT
.,=500 ;SET LOCATION COUNTER TO ABSOLUTE 500
FIRST: MOV .+10,COUNT ; THE LABEL FIRST HAS THE VALUE 500
 ; (OCTAL) ,+10 EQUALS 510 (OCTAL), THE CONTENTS OF LOCATION 510 (OCTAL) WILL
 ; BE DEPOSITED IN LOCATION COUNT.
.,=520 ; THE ASSEMBLY LOCATION COUNTER NOW
 ; HAS A VALUE OF ABSOLUTE 520 (OCTAL).
```
1.6 RELOCATION AND LINKING

The output of the relocatable assembler is an object module which must be processed by the PDP-11 Linker, LINK-11S, before loading and execution. The Linker essentially fixes (i.e., makes absolute) the values of external or relocatable symbols and creates another module (load module) which contains the binary data to be loaded and executed.

To enable the Linker to fix the value of an expression the assembler issues certain directives to the Linker together with the required parameters. In the case of relocatable expressions the Linker adds the base of the relocatable section (the location in memory of relocatable 0) to the value of the relocatable expression provided by the Assembler. In the case of an external expression the value of the external term in the expression is determined by the Linker (since the external symbol must be defined in one of the other object modules being linked and adds it to the value of the external expression provided by the Assembler.

All instructions that are to be modified as described above will be marked by a single apostrophe in the assembly listing. Thus the binary text output will look as follows for the given examples:

```
005065' CLR EXTERNAL(5) ; VALUE OF EXTERNAL SYMBOL
000000  ; ASSUMED ZERO; WILL BE
005065' CLR EXTERNAL+6(5) ; MODIFIED BY THE LINKER,
000006
005065' CLR RELOCATABLE(5) ; ASSUMING WE ARE IN THE
000040  ; ABSOLUTE SECTION AND
          ; THE VALUE OF RELOCATABLE
          ; IS RELOCATABLE 40
```
1.7 ADDRESSING

The Program Counter (register 7 of the eight general registers) always contains the address of the next word to be fetched; i.e., the address of the next instruction to be executed, or the second or third word of the current instruction.

In order to understand how the address modes operate and how they assemble, the action of the Program Counter must be understood. The key rule is:

Whenever the processor implicitly uses the Program Counter to fetch a word from memory, the Program Counter is automatically incremented by two after the fetch.

That is, when an instruction is fetched, the PC is incremented by two, so that it is pointing to the next word in memory; and, if an instruction uses indexing (see sections 1.7.7, 1.7.8 and 1.7.10), the processor uses the Program Counter to fetch the base from memory. Hence, using the rule above, the PC increments by two, and now points to the next word.

The following conventions are used in this section:

1. Let E be any expression as defined in Section 1.4.

2. Let R be a register expression. This is any expression containing a term preceded by a % character of a symbol previously equated to such a term.

Examples:

\[
\begin{align*}
R0 &= \%o \\
R1 &= R0+1 \\
R2 &= 1+\%1
\end{align*}
\]

\$GENERAL REGISTER 0

\$GENERAL REGISTER 1

\$GENERAL REGISTER 2

3. Let ER be a register expression or an expression in the range 0 to 7 inclusive.

4. Let A be a general address specification which produces a 6-bit mode address field as described in a PDP-11 Processor Handbook.

The addressing specifications, A, may now be explained in terms of E, R, and ER as defined above. Each will be illustrated with the single operand instruction CLR or double operand instruction MOV.

1.7.1 Register Mode

The register contains the operand.

Format:  R

Example:

\[
\begin{align*}
R0 &= \%0 \\
CLR & R0 \$CLEAR REGISTER 0
\end{align*}
\]
1.7.2 Deferred Register Mode

The register contains the address of the operand.

Format: @R or (ER)

Example:

```
CLR@R1 ; CLEAR THE WORD AT THE
or ; ADDRESS CONTAINED IN
CLR(1) ; REGISTER 1
```

1.7.3 Autoincrement Mode

The contents of the register are incremented immediately after being used as the address of the operand.

Format: (ER)+

Examples:

```
CLR (R0)+ ; CLEAR WORDS AT ADDRESSES
CLR (R0+3)+ ; CONTAINED IN REGISTERS 0, 3, AND 2
CLR (2)+ ; AND INCREMENT REGISTER CONTENTS
; BY TWO.
```

NOTE

Both JMP and JSR instructions using mode 2 (non-deferred autoincrement mode), execute differently on different PDP-11 processors. Avoid use of these instructions with mode 2 addressing.

Double operand instructions of the addressing form @R, (R)+ or @R, -(R) where the source and destination registers are the same, give different results on different PDP-11 processors, and should be avoided.

1.7.4 Deferred Autoincrement Mode

The register contains the pointer to the address of the operand. The contents of the register are incremented after being used.

Format: @(ER)+

Example:

```
CLR @3+ ; CONTENTS OF REGISTER 3 POINT
; TO ADDRESS OF WORD TO BE CLEARED
; BEFORE BEING INCREMENTED BY TWO
```

1-13
1.7.5 Autodecrement Mode

The contents of the register are decremented before being used as the address of the operand (see note in Section 1.7.3).

Format:  -(ER)

Examples:

CLR -(R0)  ;DECREMENT CONTENTS OF REGISTERS
CLR -(R0+3) ;0, 3 AND 2 BEFORE USING
CLR -(2)   ;AS ADDRESSES OF WORDS TO BE CLEARED

1.7.6 Deferred Autodecrement Mode

The contents of the register are decremented before being used as the pointer to the address of the operand.

Format:  @-(ER)

Example:

CLR @-(2)      ;DECREMENT CONTENTS OF REG. 2
               ;BEFORE USING AS POINTER TO ADDRESS
               ;OF WORD TO BE CLEARED.

1.7.7 Index Mode

Format:  E(ER)

The value of an expression E is stored as the second or third word of the instruction. The effective address is calculated as the value of E plus the contents of register ER. The value E is called the base.

Examples:

CLR X+2(R1)    ;EFFECTIVE ADDRESS IS X+2 PLUS
               ;THE CONTENTS OF REGISTER 1
CLR -(2)(3)    ;EFFECTIVE ADDRESS IS -2 PLUS
               ;THE CONTENTS OF REGISTER 3

1.7.8 Deferred Index Mode

An expression plus the contents of a register gives the pointer to the address of the operand.

Format:  @E(ER)

Example:

CLR @14(4)  ;IF REGISTER 4 HOLDS 100, AND LOCATION
            ;114 HOLDS 2000, LOC.2000 IS CLEARED.
1.7.9 Immediate Mode and Deferred Immediate (Absolute) Mode

The immediate mode allows the operand itself to be stored as the second or third word of the instruction. It is assembled as an autoincrement of register 7, the PC.

Format: \#E

Examples:

MOV \#100,R0  \#MOVE AN OCTAL 100 TO REGISTER 0
MOV \#X,R0   \#MOVE THE VALUE OF SYMBOL X TO
              \#REGISTER 0.

The operation of this mode is explained as follows:

The statement MOV \#100,R3 assembles as two words. These are:

```
0 1 2 7 0 3
0 0 0 1 0 0
```

Just before this instruction is fetched and executed, the PC points to the first word of the instruction. The processor fetches the first word and increments the PC by two. The source operand mode is 27 (autoincrement the PC). Thus the PC is used as a pointer to fetch the operand (the second word of the instruction) before being incremented by two, to point to the next instruction.

If the \#E is preceded by @, E specifies an absolute address.

1.7.10 Relative and Deferred Relative Modes

Relative mode is the normal mode for memory references.

Format:  E

Examples:

CLR 100  \#CLEAR LOCATION 100
MOV X,Y  \#MOVE CONTENTS OF LOCATION X TO
          \#LOCATION Y.

This mode is assembled as Index mode, using 7, the PC, as the register. The base of the address calculation, which is stored in the second or third word of the instruction, is not the address of the operand. Rather, it is the number which, when added to the PC, becomes the address of the operand. Thus, the base is X-PC. The operation is explained as follows:

If the statement MOV 100,R3 is assembled at absolute location 20 then the assembled code is:

```
Location 20: 0 1 6 7 0 3
Location 22: 0 0 0 0 5 4
```

The processor fetches the MOV instruction and adds two to the PC so that it points to location 22. The source operand mode is 67; that is; indexed by the PC. To pick up the base, the processor fetches the word pointed to by the PC and adds two to the PC. The PC now points to location 24. To calculate the address of the source operand, the base is added to the designated register. That is, BASE+PC=54+24=100, the operand address.
Since the Assembler considers "." as the address of the first word of the instruction, an equivalent statement would be

\[ \text{MOV 100 .-. 4(\text{PC}),R3} \]

This mode is called relative because the operand address is calculated relative to the current PC. The base is the distance (in bytes) between the operand and the current PC. If the operator and its operand are moved in memory so that the distance between the operator and data remains constant, the instruction will operate correctly.

If \( E \) is preceded by \( @ \) the expression's value is the pointer to the address of the operand.

1.7.11 Table of Mode Forms and Codes (6-bit(A) format only - see Section 1.7.12)

Each instruction takes at least one word. Operands of the first six forms listed below, do not increase the length of an instruction. Each operand in one of the other modes, however, increases the instruction length by one word.

<table>
<thead>
<tr>
<th>Form</th>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>these</td>
<td></td>
<td></td>
</tr>
<tr>
<td>forms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>increases</td>
<td></td>
<td></td>
</tr>
<tr>
<td>the</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction length</td>
<td></td>
<td></td>
</tr>
<tr>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>these</td>
<td></td>
<td></td>
</tr>
<tr>
<td>forms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>increases</td>
<td></td>
<td></td>
</tr>
<tr>
<td>the</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Form</th>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any of these</td>
<td></td>
<td></td>
</tr>
<tr>
<td>forms adds a</td>
<td></td>
<td></td>
</tr>
<tr>
<td>word to the</td>
<td></td>
<td></td>
</tr>
<tr>
<td>instruction</td>
<td></td>
<td></td>
</tr>
<tr>
<td>length.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>None</td>
<td></td>
<td></td>
</tr>
<tr>
<td>of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>these</td>
<td></td>
<td></td>
</tr>
<tr>
<td>forms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>increases</td>
<td></td>
<td></td>
</tr>
<tr>
<td>the</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Notes:

1. An alternate form for \( @R \) is \( (ER) \). However, the form \( @0(ER) \) is equivalent to \( @0(ER) \).

2. The form \( @#E \) differs from the form \( E \) in that the second or third word of the instruction contains the absolute address of the operand rather than the relative distance between the operand and the PC. Thus, the statement \( CLR @#100 \) will clear location 100 even if the instruction is moved from the point at which it was assembled.

The Assembler is not particular about left and right and dangling + and - signs in address fields. The following are some examples of incorrect syntax that assemble as indicated, without an error indication.
PAL-11S ASSEMBLY LANGUAGE AND ASSEMBLER

Form Assembles As: Form Assembles As:
(R2)A A(R2) (R2)- -(R2)
A-(R2) A(R2) or A-0(R2) @ (R2)A @ A(R2)
A(Rw)+ A(R2) A(R2)+B A+B(R2)
+(R2) (R2)+

1.7.12 Instruction Forms

The instruction mnemonics are given in Appendix B. This section defines the number and nature of the operand fields for these instructions.

In the table that follows, let R, E, and ER represent expressions as defined in Sections 1.4 and 1.7 and let A be a 6-bit address specification of the forms:

<table>
<thead>
<tr>
<th>R</th>
<th>ER</th>
<th>-(ER)</th>
<th>@ -(ER)</th>
</tr>
</thead>
<tbody>
<tr>
<td>@R or (R)</td>
<td>E(ER)</td>
<td>@ E(ER)</td>
<td></td>
</tr>
<tr>
<td>@ (ER)+</td>
<td>@#E</td>
<td>@ #E</td>
<td></td>
</tr>
</tbody>
</table>

Table 1-1
Instruction Operand Fields

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Operand</td>
<td>Op A,A</td>
<td>MOV (R6)+, @Y</td>
</tr>
<tr>
<td>Single Operand</td>
<td>Op A</td>
<td>CLR -(R2)</td>
</tr>
<tr>
<td>OPERATE</td>
<td>Op</td>
<td>HALT</td>
</tr>
<tr>
<td>Branch</td>
<td>Op E</td>
<td>BR X+2</td>
</tr>
<tr>
<td>where -128&lt;(E-.2)/2&lt;127</td>
<td></td>
<td>BLO .4</td>
</tr>
<tr>
<td>Subroutine Call</td>
<td>JSR ER,A</td>
<td>JSR PC, SUBR</td>
</tr>
<tr>
<td>Subroutine Return</td>
<td>RTS ER</td>
<td>RTS PC</td>
</tr>
<tr>
<td>EMT/TRAP</td>
<td>Op or Op E</td>
<td>EMT</td>
</tr>
<tr>
<td>where 0&lt;E&lt;377</td>
<td></td>
<td>EMT 31</td>
</tr>
</tbody>
</table>

The branch instructions are one word instructions. The high byte contains the op code and the low byte contains an 8-bit signe offset (7 bits plus sign) which specifies the branch address relative to the PC. The hardware calculates the branch address as follows:

1. Extend the sign of the offset through bits 8-15.
2. Multiply the result by 2. This creates a word offset rather than a byte offset.
3. Add the result to the PC to form the final branch address.

The Assembler performs the reverse operation to form the byte offset from the specified address. Remember that when the offset is added to the PC, the PC is pointing to the word following the branch instruction; hence the factor -2 in the calculation.

Byte offset = (E-PC)/2 truncated to eight bits.

Since PC = .+2, we have

Byte offset = (E+.2)/2 truncated to eight bits.
PAL-IIS ASSEMBLY LANGUAGE AND ASSEMBLER

NOTE

It is illegal to branch to a location specified as an external symbol, or to a relocatable symbol when within an absolute section, or to an absolute symbol when within a relocatable section.

The EMT and TRAP instructions do not use the low-order byte of the word. This allows information to be transferred to the trap handlers in the low-order byte. If EMT or TRAP is followed by an expression, the value is put into the low-order byte of the word. However, if the expression is too big (>3776) it is truncated to eight bits and a Truncation (T) error occurs.

Do not try to micro-program the condition code operators (see Appendix B, B.4). This makes sense in the PDP-IIS hardware; however, the current PAL-IIS Assembler does not support this capability. Thus:

```
CLC!CLV
```

results in a Q error (see Appendix B, B.5) and the statement is assembled as CLC.

Expressions in the Assembler do, however, allow logical operators and the use of instruction mnemonics. Thus, the proper ways to write the above statement:

```
.WORD CLC!           #Operand of .WORD
+CLC!CLV             #Operand of default .WORD
!CLC!CLV             #Operand of default .WORD
```

1.8 ASSEMBLER DIRECTIVES

Assembler directives (sometimes called pseudo-ops) direct the assembly process and may generate data.

Assembler directives may be preceded by a label and followed by a comment. The assembler directive occupies the operator field. Only one directive may be placed in any one statement. One or more operands may occupy the operand field or it may be void -- allowable operands vary from directive to directive.

1.8.1 .TITLE

The .TITLE directive is used to name the object module. The name is assigned by the first symbol following the directive. If there is no .TITLE statement the default name assigned is "MAIN".

1.8.2 .GLOBL

The .GLOBL directive is used to declare a symbol as being global. It may be an entry symbol, in which case it is defined in the program, or it may be an external symbol, in which case it should be defined in another program which will be linked with this program by the linker.

The form of the .GLOBL directive is

```
.GLOBL NAMA, NAMB,...,NAMN
```

1-18
NOTE

A symbol cannot be declared global by defining it as a global expression in a direct assignment statement.

If an illegal character is detected in the operand field of a .GLOBL statement, an error message is not generated; and the Assembler may ignore the remainder of the statement. Thus:

```
GLOBL A,B,@C,D
```

assembles without error as:

```
.GLOBL A,B
```

1.8.3 Program Section Directives (.ASECT and .CSECT)

The relocatable assembler provides for two program sections, an absolute section declared by an .ASECT directive and a relocatable section declared by a .CSECT directive. These directives therefore enable the programmer to specify that parts of his program be assembled in the absolute section and others in a relocatable section. The scope of each directive extends until a directive to the contrary is given. The Assembler initially starts in the relocatable section. Thus, if the first statement of a program were

```
A: .ASECT
```

the label "A" would be a relocatable symbol which is assigned the value of relocatable zero. The absolute value of A will be calculated by the Linker by adding the value of the base of the relocatable section.

Example:

```
.ASECT
=1000
A: CLR X
.CSECT
X: JMP A
.END
```

The absolute and/or relocatable section may be discontinued (by switching to the alternate section) and then continued where they left off by using another .ASECT or .CSECT statement.

Example:

```
.CSECT
.WORD 0,1,2
.ASECT
.WORD 0,1,2
.CSECT
.WORD 0
.END
```

If a label is defined twice, first in an absolute section and then in a relocatable section, the symbol will be relocatable but its value will be as defined in the absolute section.
1.8.4 .EOT

The .EOT directive indicates the physical End Of Tape though not the logical end of the program. If the .EOT is followed by a single line feed or form feed, the Assembler will still read to the end of the tape, but will not process anything past the .EOT directive. If .EOT is followed by at least two line feeds or form feeds, the Assembler will stop before the end of the tape. Either case is proper, but it should be understood that even though it appears as if the Assembler has read too far, it actually hasn't.

If a .EOT is embedded in a tape, and more information to be assembled follows it, .EOT must be immediately followed by at least two line feeds or form feeds. Otherwise, the first line following the .EOT will be lost.

Any operands following a .EOT directive will be ignored. The .EOT directive allows several physically separate tapes to be assembled as one program. The last tape should be terminated by a .END directive (see Section 1.8.6) but may be terminated with .EOT (see .END emulation in Section 1.9.4).

1.8.5 .EVEN

The .EVEN directive ensures that the assembly location counter is even by adding one if it is odd. Any operands following a .EVEN directive will be ignored.

1.8.6 .END

The .END directive indicates the logical and physical end of the source program. The .END directive may be followed by only one operand, an expression indicating the program's transfer address.

At load time, the load module will be loaded and program execution will begin at the transfer address indicated by the .END directive. If the address is not specified, the loader will halt after reading in the load module.

1.8.7 .WORD

The .WORD assembler directive may have one or more operands, separated by commas. Each operand is stored in a word of the object program. If there is more than one operand, they are stored in successive words. The operands may be any legally formed expression. For example,

```asm
E=1420
SAL=0
.WORD 177535, +4, SAL $STORED IN WORDS 1420, 1422 AND $1424 WILL BE 177535, 1426, AND 0
```

Values exceeding 16 bits will be truncated from the left, to word length.
A .WORD directive followed by one or more void operands separated by commas will store zeros for the void operands. For example,

```
,=1430 ;ZERO, FIVE, AND ZERO ARE STORED
,WORD ,5, ;IN WORDS 1430, 1432, AND 1434
```

An operator field left blank will be interpreted as the .WORD directive if the operand field contains one or more expressions. The first term of the first expression in the operand field must not be an instruction mnemonic or assembler directive unless preceded by a +, -, or one of the logical operators, ! or &. For example,

```
,=440 ;THE OP-CODE FOR MOV, WHICH IS 010000,
LABEL; +MOV+LABEL ;IS STORED IN LOCATION 440. 440 IS
;STORED IN LOCATION 442.
```

Note that the default .WORD will occur whenever there is a leading arithmetic or logical operator, or whenever a leading symbol is encountered which is not recognized as an instruction mnemonic or assembler directive. Therefore, if an instruction mnemonic or assembler directive is misspelled, the .WORD directive is assumed and errors will result. Assume that MOV is spelled incorrectly as MOR:

```
MOR A,B
```

Two error codes can result: A Q will occur because an expression operator is missing between MOR and A, and a U will occur if MOR is undefined. Two words will be generated; one for MOR A and one for B.

1.8.8 .BYTE

The .BYTE assembler directive may have one or more operands separated by commas. Each operand is stored in a byte of the object program. If multiple operands are specified, they are stored in successive bytes. The operands may be any legally formed expression with a result of 8 bits or less. For example,

```
SAM=5 ;STORED IN LOCATION 410 WILL BE
,=410 ;060 (THE OCTAL EQUIVALENT OF 48).
,BYTE 4B,,SAM ;IN 411 WILL BE 005.
```

If the expression has a result of more than 8 bits, it will be truncated to its low-order 8 bits and will be flagged as a T error. If an operand after the .BYTE directive is left void, it will be interpreted as zero. For example,

```
,=420 ;ZERO WILL BE STORED IN
,BYTE , , ;BYTES 420, 421 AND 422.
```

If the expression is relocatable, a warning flag, A, will be given.

1.8.9 .ASCII

The .ASCII directive translates strings of ASCII characters into their 7-bit ASCII codes with the exception of null, rubout, carriage return, line feed and form feed. The text to be translated is delimited by a character at the beginning and the end of the text. The delimiting character may be any printing ASCII character except colon and equal sign and those used in the text string. The 7-bit ASCII code generated for each character will be stored in successive bytes of the object program. For example,
PAL-11S ASSEMBLY LANGUAGE AND ASSEMBLER

```
=500  ;THE ASCII CODE FOR Y WILL BE
.ASCII /YES/ ;STORED IN 500, THE CODE FOR E
.ASCII /5+3/2/ ;IN 501, THE CODE FOR S IN 502.
.ASCII /DELIMITER OCCURS AMONG THE OPERANDS. THE ASCII CODES FOR S, +, AND 3 ARE
.ASCII /503, 504, AND 505, 2/ IS NOT ASSEMBLED.
```

The .ASCII directive may be terminated by any legal terminator except for = and ::. Note that if the text delimiter is also a terminator, the leading text delimiter can also serve as the .ASCII directive terminator. For example,

```
.ASCII /ABCD/ ;THE SPACE IS REQUIRED
.ASCII+ABCD+ ;BECAUSE / IS NOT A TERMINATOR.
```

1.8.10 .RAD50

PDP-11 system programs often handle symbols in a specially coded form called "RADIX 50" (this form is sometimes referred to as "MOD40"). This form allows 3 characters to be packed into 16 bits; therefore, any symbol can be held in two words, the form of the directive is:

```
.RAD50 /CCC/
```

The single operand is of the form /CCC/ where the slash (the delimiter) can be any printable character except for = and ::. The delimiters enclose the characters to be converted which may be A through Z, 0 through 9, dollar ($), dot (.) and space ( ). If there are fewer than 3 characters they are considered to be left-justified and trailing spaces are assumed. Any characters following the trailing delimiter are ignored and no error results.

Examples:

```
.RAD50 /ABC/ ;PACK ABC INTO ONE WORD
.RAD50 /AB/ ;PACK AB (SPACE) INTO ONE WORD
.RAD50 // ;PACK 3 SPACES INTO ONE WORD
```

The packing algorithm is as follows:

A. Each character is translated into its RADIX 50 equivalent as indicated in the following table:

<table>
<thead>
<tr>
<th>Character</th>
<th>RADIX 50 Equivalent (octal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(SPACE)</td>
<td>0</td>
</tr>
<tr>
<td>A-Z</td>
<td>1-32</td>
</tr>
<tr>
<td>$</td>
<td>33</td>
</tr>
<tr>
<td>.</td>
<td>34</td>
</tr>
<tr>
<td>0-9</td>
<td>36-47</td>
</tr>
</tbody>
</table>

Note that another character can be defined for code 35.

B. The RADIX 50 equivalents for characters 1 through 3 (C1,C2,C3) are combined as follows:

```
RESULT=((C1*50)+C2)*50+C3
```
1.8.11  .LIMIT

A program often wishes to know the boundaries of the relocatable code. The .LIMIT directive generates two words into which the linker puts the low and high addresses of the relocated code. The low address (inserted into the first word) is the address of the first byte of code. The high address is the address of the first free byte following the relocated code. These addresses will always be even since all relocatable sections are loaded at even addresses and if a relocatable section consists of an odd number of bytes the linker adds one to the size to make it even.

1.8.12 Conditional Assembly Directives

Conditional assembly directives provide the programmer with the capability to conditionally include or not include portions of his source code in the assembly process. In what follows, E denotes an expression and S(i) denotes a symbol. The conditional directives are:

- .IFZ E ;IF E=0
- .IFNZ E ;IF E#0
- .IFL E ;IF E<0
- .IFLE E ;IF E≤0
- .IFG E ;IF E>0
- .IFGE E ;IF E≥0
- .IFDF S (1) [!,&] S (2) [!,&]...[!,&] S(N) (!=OR, &=AND)
- .IFNDF S (1) [!,&] S (2) [!,&]...[!,&] S(N)

If the condition is met, all statements up to the matching .ENDC are assembled. Otherwise, the statements are ignored until the matching .ENDC is detected.

In the above,.IFDF and .IFNDF mean "if defined" and "if undefined" respectively. The scan is left to right, no parentheses permitted.

Example:

- .IFDF S!&T&U Means assemble if either S or T is defined and U is defined
- .IFNDF T&U!S Means assemble if both T and U are undefined or if S is undefined

General Remarks:

An errored or null expression takes the default value 0 for purposes of the conditional test. An error in syntax, e.g., a terminator other than ;, !, &, or CR results in the undefined situation for .IFDF and .IFNDF, as does an errored or null symbol.

All conditionals must end with the .ENDC directive. Anything in the operand field of .ENDC is ignored. Nesting is permitted up to a depth of 127 . Labels are permitted on conditional directives, but the scan is purely left to right. For example:

```
.A:   .IFZ 1
       .ENDC
```

A is ignored.

```
A:    .IFZ 1
       .ENDC
```

A is entered in the symbol table.
PAL-11S ASSEMBLY LANGUAGE AND ASSEMBLER

If a .END is encountered while inside a satisfied conditional, a Q flag will appear, but the .END directive will still be processed normally. If more .ENDC's appear than are required, Q flags appear on the extras.

1.9 OPERATING PROCEDURES

1.9.1 Introduction

The Assembler enables you to assemble an ASCII tape containing PAL-11 statements into a relocatable binary tape (object module). To do this, two or three passes are necessary. On the first pass, the Assembler creates a table of user-defined symbols and their associated values, and a list of undefined symbols is printed on the teleprinter. On the second pass the Assembler assembles the program and punches out an absolute binary tape and/or outputs an assembly listing. During the third pass (this pass is optional), the Assembler punches an absolute binary tape or outputs an assembly listing. The symbol table (and/or a list of errors) may be output on any of these passes. The input and output devices as well as various options are specified during the initial dialogue (see Section 1.9.3). The Assembler initiates the dialogue immediately after being loaded and after the last pass of an assembly.

1.9.2 Loading PAL-11S

PAL-11S is loaded by the Paper Tape Software Absolute Loader. Note that on systems with hardware switch registers, the start address of the Absolute Loader must be in the Switch Register when loading the Assembler. This is because the Assembler tape has an initial program which clears all of core up to the address specified in the Switch Register, and jumps to that address to start loading the Assembler.

1.9.3 Initial Dialogue

After being loaded, the Assembler prints its name and version and then initiates dialogue by printing on the teleprinter

*S

meaning "What is the Source symbolic input device?" The response may be

← use Low-speed reader (← denotes typing the RETURN key)
H meaning High-speed reader
L meaning Low-speed reader
T meaning Teleprinter keyboard

The device specification is terminated, as is all user response, by typing the RETURN key.

If an error is made in typing at any time, typing the RUBOUT key will erase the immediately preceding character if it is on the current line. Typing CTRL/U will erase the whole line on which it occurs.
After the *S question and response, the Assembler prints:

*B

meaning "What is the Binary output device?" The responses to *B are similar to those for *S:

H meaning High-speed punch
L meaning Low-speed punch
meaning do not output binary tape (\ denotes typing the RETURN key)

In addition to I/O device specification, various options may be chosen. The binary output will occur on the second pass unless /3 (indicating the third pass) is typed following the H or L. Errors will be listed on the same pass if /E is typed. If /E is typed in response to more than one inquiry, only the last occurrence will be honored. It is strongly suggested that the errors be listed on the same pass as the binary output, since errors may vary from pass to pass.

If both /3 and /E are typed, /3 must precede /E. The response is terminated by typing the RETURN key. Examples:

*B L/E Binary output on the low-speed punch and the errors on the teleprinter, both during the second pass.
*B H/3/E Binary output on the high-speed punch and the errors on the teleprinter during the third pass.
*B Meaning the RETURN key alone will cause the Assembler to omit binary output.

After the *B question and response, the Assembler prints:

*L

meaning "What is the assembly Listing output device?" The response to *L may be:

L meaning Low-speed punch
H meaning High-speed punch
T meaning Teleprinter
P meaning Line Printer
meaning do not output listing (\ denotes typing RETURN)

After the I/O device specification, pass and error list options similar to those for *B may be chosen. The assembly listing will be output on the third pass unless /2 (indicating the second pass) is typed following H, L, T, or P. Errors will be listed on the teleprinter during the same pass if /E is typed. If both /2 and /E are typed, /2 must precede /E. The response is terminated by typing the RETURN key. Examples:

*L L/2/E Listing on low-speed punch and errors on teleprinter during second pass.
*L H Listing on high-speed punch during third pass
*L Meaning the RETURN key alone will cause the Assembler to omit listing output.

1-25
After the *L question and response, the final question is printed on the teleprinter:

*T

meaning "What is the symbol Table output device?" The device specification is the same as for *L question. The symbol table will be output at the end of the first pass unless /2 or /3 is typed in response to *T. The first tape to be assembled should be placed in the reader before typing the RETURN key because assembly will begin upon typing RETURN to the *T question. The /E option is not a meaningful response to *T. Example

*T T/3 Symbol table output on teleprinter at end of third pass.

*T Typing the RETURN key alone will cause the Assembler to omit symbol table output.

The symbol table is printed alphabetically, three symbols per line. Each symbol printed is followed by its identifying characters and by its value. If the symbol is undefined, six asterisks replace its value. The identifying characters indicate the class of the symbol; that is, whether it is a label, direct assignment, register symbol, etc. The following examples show the various forms.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCDEF</td>
<td>001244</td>
<td>(Defined Label)</td>
</tr>
<tr>
<td>R3</td>
<td>%000003</td>
<td>(Register Symbol)</td>
</tr>
<tr>
<td>DIRASM</td>
<td>177777</td>
<td>(Direct Assignment)</td>
</tr>
<tr>
<td>XYZ</td>
<td>*****</td>
<td>(Undefined direct assignment)</td>
</tr>
<tr>
<td>R6</td>
<td>%*****</td>
<td>(Undefined register symbol)</td>
</tr>
<tr>
<td>LABEL</td>
<td>*****</td>
<td>(Undefined label)</td>
</tr>
</tbody>
</table>

Generally, undefined symbols and external symbols will be listed as undefined direct assignments. Multiply-defined symbols are not flagged in the symbol table printout but are flagged wherever they are used in the program.

If the symbol is relocatable or global or both, the symbol's value will be followed by an R, a G or both.

It is possible to output both the binary tape and the assembly listing on the same pass, thereby reducing the assembly process to two passes (see Example 1 below). This will happen automatically unless the binary device and the listing device are conflicting devices or the same device (see Example 2 below). The only conflicting devices are the teleprinter and the low-speed punch. Even though the Assembler deduces that three passes are necessary, the binary and listing can be forced on pass 2 by including /2 in the responses to *B and *L (see Example 3 below).

Example 1. Runs 2 passes:

*S H High-speed reader
*B H High-speed punch
*L P Line Printer
*T T Teleprinter

Example 2. Runs 3 passes:

*S H High-speed reader
*B H High-speed punch
*L H High-speed punch
*T T Teleprinter
Example 3. Runs 2 passes:

- **S** H High-speed reader
- **B** H/2 High-speed punch on pass 2
- **L** H/2 High-speed punch on pass 2
- **T** T Teleprinter

Note that there are several cases where the binary output can be intermixed with ASCII output:

a. **B** H/2 Binary and listing to punch on pass 2.

b. **L** H/2 Binary to low-speed punch and error listing to teleprinter (and low-speed punch).

c. **B** L/2/E Binary, error listing, and **L** T/2 listing to low speed punch.

The object module so generated is acceptable to the Linker as long as there are no CTRL/A characters in the source program. The start of every block on the binary tape is indicated by a 001 and the Linker ignores all information until a 001 is detected. Thus, all source and/or error messages will be ignored if they do not contain any CTRL/A characters (octal 001).

If a character other than those mentioned is typed in response to a question, the Assembler will ignore it and print the question again.

Example:

- **S** H High-speed reader
- **B** Q Q is not a valid response
- **B** The question is repeated

If at any time you wish to restart the Assembler, type CTRL/P. If the low-speed reader is the source input device, turn it off before typing CTRL/P.

When no passes are omitted or error options specified, the Assembler performs as follows:

**PASS 1:**

Assembler creates a table of user-defined symbols and their associated values to be used in assembling the source to object program. Undefined symbols (not including external globals) are listed on the teleprinter at the end of the pass. The symbol table is also listed at this time. If an illegal location statement of the form .=expression is encountered, the line and error code will be printed out on the teleprinter before the assembly proceeds. An error in a location statement is usually a fatal error in the program and should be corrected.

**PASS 2:**

Assembler punches the object module, and prints the pass error count and undefined location statements on the teleprinter.

**PASS 3:**

Assembler prints or punches the assembly program listing, undefined location statements, and the pass error count on the teleprinter.
The functions of passes 2 and 3 will occur simultaneously on pass 2 if
the binary and listing devices are different, and do not conflict with
each other (the low-speed punch and teleprinter conflict).
Furthermore, if the binary object module is not requested, the listing
will be produced on pass 2.

The following table summarizes the initial dialogue questions:

<table>
<thead>
<tr>
<th>PRINTOUT</th>
<th>INQUIRY</th>
</tr>
</thead>
<tbody>
<tr>
<td>*S</td>
<td>What is the input device of the Source symbolic tape?</td>
</tr>
<tr>
<td>*B</td>
<td>What is the output device of the Binary object tape?</td>
</tr>
<tr>
<td>*L</td>
<td>What is the output device of the assembly Listing?</td>
</tr>
<tr>
<td>*T</td>
<td>What is the output device of the symbol Table?</td>
</tr>
</tbody>
</table>

The following table summarizes the legal responses:

<table>
<thead>
<tr>
<th>CHARACTER</th>
<th>RESPONSE INDICATED</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Teleprinter keyboard</td>
</tr>
<tr>
<td>L</td>
<td>Low-speed reader or punch</td>
</tr>
<tr>
<td>H</td>
<td>High-speed reader or punch</td>
</tr>
<tr>
<td>P</td>
<td>Line Printer</td>
</tr>
<tr>
<td>/1</td>
<td>Pass 1</td>
</tr>
<tr>
<td>/2</td>
<td>Pass 2</td>
</tr>
<tr>
<td>/3</td>
<td>Pass 3</td>
</tr>
<tr>
<td>/E</td>
<td>Errors listed on same pass (not meaningful response to *S or *T)</td>
</tr>
<tr>
<td>↓</td>
<td>Omit function (except in response to *S).</td>
</tr>
</tbody>
</table>

Typical examples of complete initial dialogues:

For minimal PDP-11 configuration:

* S L Source input on low-speed reader
* B L/E Binary output on low-speed punch
  errors during same (second) pass
* L T Listing on teleprinter during pass 3
* T T Symbol table on teleprinter at end of pass 1

For a PDP-11 with high-speed I/O devices:

* S H Source input on high-speed reader
* B H/E Binary output on high-speed punch
  errors during same (second) pass
* L ↓ No listing
* T T/2 Symbol table on teleprinter at end of pass 2.

1.9.4 Assembly Dialogue

During assembly, the Assembler will pause to print on the teleprinter
various messages to indicate that you must respond in some way before
the assembly process can continue. You may also type CTRL/P, at any
time, if you wish to stop the assembly process and restart the initial
dialogue, as mentioned in the previous section.

When a .EOT assembler directive is read on the tape, the Assembler
prints

```
EOF ?
```

and pauses. During this pause, the next tape is placed in the reader,
and RETURN is typed to continue the assembly.
If the specified assembly listing output device is the high-speed punch and if it is out of tape, or if the device is the Line Printer and is out of paper, the Assembler prints on the teleprinter

EOM ?

and waits for tape or paper to be placed in the device. Type the RETURN key when the tape or paper has been replenished; assembly will continue.

Conditions causing the EOM? messages for an assembly listing device are:

<table>
<thead>
<tr>
<th>HSP</th>
<th>LPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>No power</td>
<td>No power</td>
</tr>
<tr>
<td>No tape</td>
<td>Printer drum gate open</td>
</tr>
<tr>
<td></td>
<td>Too hot</td>
</tr>
<tr>
<td></td>
<td>No paper</td>
</tr>
</tbody>
</table>

There is no EOM if the line printer is switched off-line, although characters may be lost for this condition as well as for an EOM.

If the binary output device is the high-speed punch and if it is out of tape, the Assembler prints:

EOM ?

The assembly process is aborted and the initial dialogue is begun again.

When a .END assembler directive is read on the tape, the Assembler prints:

END ?

and pauses. During the pause the first tape is placed in the reader, and the RETURN key is typed to begin the next pass. On the last pass, the .END directive causes the Assembler to begin the initial dialogue for the next assembly.

If you are starting the binary pass and the binary is to be punched on the low-speed punch, turn the punch on before typing the RETURN key for starting the pass. The carriage return and line feed characters will be punched onto the binary tape, but the Linker will ignore them.

If the last tape ends with a .EOT, the Assembler may be told to emulate a .END assembler directive by responding with E followed by the RETURN key. The Assembler will then print

END ?

and wait for another RETURN before starting the next pass. Example:

EOF ? E
END ?

Note that forcing a .END in this manner causes the error counter to be incremented by one.
1.9.5 Assembly Listing

PAL-11S produces a side-by-side assembly listing of symbolic source statements, their octal equivalents, assigned addresses, and error codes, as follows:

```
EELLLLLL O00000A0SSS.......S
000000
000000
```

The E's represent the error field. The L's represent the address. The O's represent the object data in octal. The S's represent the source statement. "A" represents a single apostrophe which indicates that either the second, third or both words of the instruction will be modified by the Linker. While the Assembler accepts 72 characters per line on input, the listing is reduced by the 16 characters to the left of the source statement.

The above represents a three-word statement. The second and third words of the statement are listed under the command word. No addresses precede the second and third words since the address order is sequential.

The third line is omitted for a two-word statement; both second and third lines are omitted for a one-word statement.

For a .BYTE directive, the object data field is three octal digits.

For a direct assignment statement, the value of the defining expression is given in the object code field although it is not actually part of the code of the object program.

The .ASECT and .CSECT directives cause the current value of the appropriate location counter (absolute or relocatable) to be printed.

Each page of the listing is headed by a page number (octal).

1.9.6 Object Module Output

The output of the assembler during the binary object pass is an object module which is meaningful only to the linker. What follows gives an overview of what the object module contains and at what stage each part of it is produced.

The binary object module consists of three main types of data block:

- a) Global symbol directory (GSD)
- b) Text blocks (TXT)
- c) Relocation Directory (RLD)

1.9.6.1 Global Symbol Directory - As the name suggests, the GSD contains a list of all the global symbols together with the name of the object module. Each symbol is in Radix-50 form and contains information regarding its mode and value whenever known.

The GSD is created at the start of the binary object pass.
1.9.6.2 **Text Block** — The text blocks consist entirely of the binary object data as shown in the listing. The operands are in the unmodified form.

1.9.6.3 **Relocation Directory** — The RLD blocks consist of directives to the Linker which may reference the text block preceding the RLD. These directives control the relocation and linking process.

Text and RLD blocks are constructed during the binary object pass. Outputting of each block is done whenever either the TXT or RLD buffer is full and whenever the location counter needs to be modified.

1.10 **ERROR CODES**

The error codes printed beside the octal and symbolic code in the assembly listing have the following meanings:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Addressing error. An address within the instruction is incorrect. Also may indicate a relocation error.</td>
</tr>
<tr>
<td>B</td>
<td>Bounding error. Instructions or word data are being assembled at an odd address in memory. The location counter is updated by +1.</td>
</tr>
<tr>
<td>D</td>
<td>Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.</td>
</tr>
<tr>
<td>I</td>
<td>Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing.</td>
</tr>
<tr>
<td>L</td>
<td>Line buffer overflow. Extra characters on a line (more than 72) are ignored.</td>
</tr>
<tr>
<td>M</td>
<td>Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a previously encountered label.</td>
</tr>
<tr>
<td>N</td>
<td>Number containing 8 or 9 has decimal point missing.</td>
</tr>
<tr>
<td>P</td>
<td>Phase error. A label's definition or value varies from one pass to another.</td>
</tr>
<tr>
<td>Q</td>
<td>Questionable syntax. There are missing arguments or the instruction scan was not completed or a carriage return was not immediately followed by a line feed or form feed.</td>
</tr>
<tr>
<td>R</td>
<td>Register-type error. An invalid use of or reference to a register has been made.</td>
</tr>
<tr>
<td>S</td>
<td>Symbol table overflow. When the quantity of user-defined symbols exceeds the allocated space available in the user's symbol table, the assembler outputs the current source line with the S error code, then returns to the initial dialogue.</td>
</tr>
</tbody>
</table>
Truncation error. A number generated more than 16
bits of significance or an expression generated more
than 8 bits of significance during the use of the
.YBYTE directive.

Undefined symbol. An undefined symbol was
encountered during the evaluation of an expression.
Relative to the expression, the undefined symbol is
assigned a value of zero.

1.11 SOFTWARE ERROR HALTS

PAL-11S loads all of its unused trap vectors with the code

.WORD .+2,HALT

so that if the trap does occur, the processor will halt in the second
word of the vector. The address of the halt, displayed in the console
address register, therefore indicates the cause of the halt.

Address of Halt (octal)  Meaning
12  Reserved instruction executed
16  Trace trap occurred
26  Power fail trap
32  EMT executed

A halt at address 40 indicates an IOXLPT detected error. R0
(displayed in the console lights) contains an identifying code:

Code in R0  Meaning
0  Illegal memory reference, SP overflow or
    illegal instruction.
1  Illegal IOX command.
2  Slot number out of range.
3  Device number illegal
4  Referenced slot not INITed.
5  Illegal Data Mode.

IOXLPT also sets R1 as follows:

If the error code is 0, R1 contains the PC at the time of the error.

If the error code is 1-5, R1 points to some element in the IOT
argument list or to the instruction following the argument list,
depending on whether IOXLPT has finished decoding all the arguments
when it detects the error.
CHAPTER 2
WRITING PAL-11A ASSEMBLY LANGUAGE PROGRAMS

PAL-11A (Program Assembly Language for the PDP-11's Absolute Assembler) enables you to write source (symbolic) programs using letters, numbers, and symbols which are meaningful to you. The source programs, generated either on-line using the Text Editor (ED-11), or off-line, are then assembled into object programs (in absolute binary) which are executable by the computer. The object program is produced after two passes through the Assembler; an optional third pass produces a complete octal/symbolic listing of the assembled program. This listing is especially useful for documentation and debugging purposes.

This chapter explains not only how to write PAL-11A programs but also how to assemble the source programs into computer-acceptable object programs. All facets of the assembly language are explained and illustrated with many examples, and the chapter concludes with assembling procedures. In explaining how to write PAL-11A source programs, it is necessary, especially at the outset, to make frequent forward references. Therefore, we recommend that you first read through the entire chapter to get a "feel" for the language, and then reread the chapter, this time referring to appropriate sections as indicated, for a thorough understanding of the language and assembling procedures.

Some notable features of PAL-11A are:

1. Selective assembly pass functions
2. Device specification for pass functions
3. Optional error listing on Teletype
4. Double buffered and concurrent I/O (provided by IOX)
5. Alphabetized, formatted symbol table listing

The PAL-11A Assembler is available in two versions: a 4K version and an 8K version.

The assembly language applies equally to both versions. The 4K version provides symbol storage for about 176 user-defined symbols, and the 8K version provides for about 1256 user-defined symbols (see Section 2.3).

In addition, the 8K version allows a line printer to be used for the program listing and/or symbol table listing.

PAL-11A is not currently available for PDP-11 systems without switch registers.
The following discussion of the PAL-11A Assembly Language assumes that you have read the PDP-11 Processor Handbook, with emphasis on those sections which deal with the PDP-11 instruction set, formats, and timings -- a thorough knowledge of these is vital to efficient assembly language programming.

2.1 CHARACTER SET

A PAL-11A source program is composed of symbols, numbers, expressions, symbolic instructions, assembler directives, arguments separators, and line terminators written using the following ASCII' characters.

1. The letters A through Z. (Upper and lower case letters are acceptable, although upon input, lower case letters will be converted to upper case letters.)

2. The numbers 0 through 9.

3. The characters . and $ (reserved for system software).

4. The separating or terminating symbols:
   := & # @ ( ) ; " ' + - & !
   carriage return    tab    space    line feed    form feed

2.2 STATEMENTS

A source program is composed of a sequence of statements, where each statement is on a single line. The statement is terminated by a carriage return character and must be immediately followed by either a line feed or form feed character. Should a carriage return character be present and not be followed by a line feed or form feed, the Assembler will generate a Q error (Section 2.10) and that portion of the line following the carriage return will be ignored. Since the carriage return is a required statement terminator, a line feed or form feed not immediately preceded by a carriage return will have one inserted by the Assembler.

It should be noted that, if the Editor (ED-11) is being used to create the source program (see Section 4.4.4), a typed carriage return (RETURN key) automatically generates a line feed character.

A statement may be composed of up to four fields which are identified by their order of appearance and by specified terminating characters as explained below and summarized in Appendix B. The four fields are:

   Label    Operator    Operand    Comment

The label and comment fields are optional. The operator and operand fields are interdependent -- either may be omitted depending upon the contents of the other.

ASCII stands for American Standard Code for Information Interchange.
2.2.1 Label

A label is a user-defined symbol (see Section 3.3.2) which is assigned the value of the current location counter. It is a symbolic means of referring to a specific location within a program. If present, a label always occurs first in a statement and must be terminated by a colon. For example, if the current location is 100(octal), the statement

```
ABCD:    MOV A,B
```

will assign the value 100(octal) to the label ABCD so that subsequent reference to ABCD will be to location 100(octal). More than one label may appear within a single label field; each label within the field will have the same value. For example, if the current location is 100, multiple labels in the statement

```
ABC:    $DD:    A7.7:    MOV A,B
```

will equate each of the three labels ABC, $DD, and A7.7 with the value 100(octal). ($ and . are reserved for system software.)

The error code M (multiple definition of a symbol) will be generated during assembly if two or more labels have the same first six characters.

2.2.2 Operator

An operator follows the label field in a statement, and may be an instruction mnemonic or an assembler directive (see Appendix B). When it is an instruction mnemonic, it specifies what action is to be performed on any operand(s) which follows it. When it is an assembler directive, it specifies a certain function or action to be performed during assembly.

The operator may be preceded only by one or more labels and may be followed by one or more operands and/or a comment. An operator is legally terminated by a space, tab, or any of the following characters.

```
#  +  -  @  (  "  '  %  !  &  ,  
line feed   form feed  carriage return
```

The use of each character above will be explained in this chapter.

Consider the following examples:

```
MOV A,B    ;⇒(TAB) terminates operator MOV
MOV@A,B    ;⇒ terminates operator MOV
```

When the operator stands alone without an operand or comment, it is terminated by a carriage return followed by a line feed or form feed character.

2.2.3 Operand

An operand is that part of a statement which is operated on by the operator -- an instruction mnemonic or assembler directive. Operands may be symbols, expressions, or numbers. When multiple operands appear within a statement, each is separated from the next by a comma.
An operand may be preceded by an operator and/or label, and followed by a comment.

The operand field is terminated by a semicolon when followed by a comment, or by a carriage return followed by a line feed or form feed character when the operand ends the statement. For example,

```
LABEL: MOV GEORGE,BOB ; THIS IS A COMMENT
```

where the space between MOV and GEORGE terminated the operator field and began the operand field; the comma separated the operands GEORGE and BOB; the semicolon terminated the operand field and began the comment.

### 2.2.4 Comments

The comment field is optional and may contain any ASCII character except null, rubout, carriage return, line feed or form feed. All other characters, even those with special significance are ignored by Assembler when used in the comment field.

The comment field may be preceded by none, any, or all of the other three fields. It must begin with the semicolon and end with a carriage return followed by a line feed or form feed character. For example,

```
LABEL: CLR HERE ; THIS IS A $1.00 COMMENT
```

Comments do not affect assembly processing or program execution, but they are useful in program listings for later analysis, checkout or documentation purposes.

### 2.2.5 Format Control

The format is controlled by the space and tab characters. They have no effect on the assembling process of the source program unless they are embedded within a symbol, number, or ASCII text; or are used as the operator field terminator. Thus, they can be used to provide a neat, readable program. A statement can be written

```
LABEL: MOV(SP)+,TAG; POP VALUE OFF STACK
```

or, using formatting characters it can be written

```
LABEL: MOV (SP)+,TAG ; POP VALUE OFF STACK
```

which is much easier to read.

Page size is controlled by the form feed character. A page of n lines is created by inserting a form feed (CTRL/FORM keys on the keyboard) after the nth line. If no form feed is present, a page is terminated after 56 lines.
2.3 SYMBOLS

There are two types of symbols, permanent and user-defined. Both are stored in the Assembler's symbol table. Initially, the symbol table contains the permanent symbols, but as the source program is assembled, user-defined symbols are added to the table.

2.3.1 Permanent Symbols

Permanent symbols consist of the instruction mnemonics (see Appendix B.3) and assembler directives (see Section 2.8). These symbols are a permanent part of the Assembler's symbol table and need not be defined before being used in the source program.

2.3.2 User-Defined Symbols

User-defined symbols are those defined as labels (see Section 2.2.1) or by direct assignment (see Section 2.3.3). These symbols are added to the symbol table as they are encountered during the first pass of the assembly. They can be composed of alphanumeric characters, dollar signs, and periods only; again, dollar signs and periods are reserved for use by the system software. Any other character is illegal and, if used, will result in the error message I (see Section 2.11). The following rules also apply to user-defined symbols:

1. The first character must not be a number.
2. Each symbol must be unique within the first six characters.
3. A symbol may be written with more than six legal characters but the seventh and subsequent characters are only checked for legality, and are not otherwise recognized by the Assembler.
4. Spaces and tabs must not be embedded within a symbol.

A user-defined symbol may duplicate a permanent symbol. The value associated with a permanent symbol that is also user-defined depends upon its use:

1. A permanent symbol encountered in the operator field is associated with its corresponding machine op-code.
2. If a permanent symbol in the operand field is also user-defined, its user-defined value is associated with the symbol. If the symbol is not found to be user-defined, then the corresponding machine op-code value is associated with the symbol.

2.3.3 Direct Assignment

A direct assignment statement associates a symbol with a value. When a direct assignment statement defines a symbol for the first time, that symbol is entered into the Assembler's symbol table and the specified value is associated with it. A symbol may be redefined by assigning a new value to a previously defined symbol. The newly assigned value will replace the previous value assigned to the symbol.
WRITING PAL-11A ASSEMBLY LANGUAGE PROGRAMS

The general format for a direct assignment statement is

\[
\text{symbol} = \text{expression}
\]

The following conventions apply:

1. An equal sign (=) must separate the symbol from the expression defining the symbol.

2. A direct assignment statement may be preceded by a label and may be followed by a comment.

3. Only one symbol can be defined by any one direct assignment statement.

4. Only one level of forward referencing is allowed.

Example of the two levels of forward referencing (illegal):

\[
\begin{align*}
X &= Y \\
Y &= Z \\
Z &= 1
\end{align*}
\]

X and Y are both undefined throughout pass 1 and will be listed on the printer as such at the end of that pass. X is undefined throughout pass 2, and will cause a U error message.

Examples:

\[
\begin{align*}
A &= 1 & \text{\texttt{\textasciitilde}THE SYMBOL A IS EQUATED WITH THE VALUE 1} \\
B &= 'A-1\&\\text{MASKLOW} & \text{\texttt{\textasciitilde}THE SYMBOL B IS EQUATED WITH THE EXPRESSION'S VALUE} \\
C\text{\texttt{\textasciitilde}D} &= 3 & \text{\texttt{\textasciitilde}THE SYMBOL D IS EQUATED WITH 3. THE LABELS C AND E ARE EQUATED WITH THE \texttt{MOV} \texttt{\\#1,ABLE} \texttt{\textasciitilde}NUMERICAL MEMORY ADDRESS OF THE MOV COMMAND.}
\end{align*}
\]

2.3.4 Register Symbols

The eight general registers of the PDP-11 are numbered 0 through 7. These registers may be referenced by use of a register symbol, that is, a symbolic name for a register. A register symbol is defined by means of a direct assignment, where the defining expression contains at least one term preceded by a % or at least one term previously defined as a register symbol.

\[
\begin{align*}
R0 &= \%0 & \text{\texttt{\textasciitilde}DEFINE RO AS REGISTER 0} \\
R3 &= R0+3 & \text{\texttt{\textasciitilde}DEFINE R3 AS REGISTER 3} \\
R4 &= 1+\%3 & \text{\texttt{\textasciitilde}DEFINE R4 AS REGISTER 4} \\
\text{THERE} &= \%2 & \text{\texttt{\textasciitilde}DEFINE \"THERE\" AS REGISTER 2}
\end{align*}
\]

It is important to note that all register symbols must be defined before they are referenced. A forward reference to a register symbol will generally cause phase errors (see Section 2.10).
The % may be used in any expression thereby indicating a reference to a register. Such an expression is a register expression. Thus, the statement

CLR %6

will clear register 6 while the statement

CLR 6

will clear the word at memory address 6. In certain cases a register can be referenced without the use of a register symbol or register expression. These cases are recognized through the context of the statement and are thoroughly explained in Sections 2.6 and 2.7. Two obvious examples of this are:

\begin{verbatim}
JSR 5:SUBR ;THE FIRST OPERAND FIELD MUST
\end{verbatim}

\begin{verbatim}
\end{verbatim}

\begin{verbatim}
CLR X(2) ;ANY EXPRESSION ENCLOSED IN \\
\end{verbatim}

\begin{verbatim}
(); MUST BE A REGISTER. IN \\
\end{verbatim}

\begin{verbatim}
;THIS CASE, INDEX REGISTER 2.
\end{verbatim}

2.4 EXPRESSIONS

Arithmetic and logical operators (see Section 2.4.2) may be used to form expressions. A term of an expression may be a permanent or user-defined symbol, a number, ASCII data, or the present value of the assembly location counter represented by the period. Expressions are evaluated from left to right. Parenthetical grouping is not allowed.

Expressions are evaluated as word quantities. The operands of a .BYTE directive (Section 2.8.5) are evaluated as word expressions before truncation to the low-order eight bits.

A missing term or expression will be interpreted as 0. A missing operator will be interpreted as +. The error code Q (Questionable syntax) will be generated for a missing operator. For example,

\begin{verbatim}
A + -100 ;OPERAND MISSING
\end{verbatim}

will be evaluated as A + 0 - 100, and

\begin{verbatim}
TAG ! LA 177777 ;OPERATOR MISSING
\end{verbatim}

will be evaluated as TAG ! LA+177777.

2.4.1 Numbers

The Assembler accepts both octal and decimal numbers. Octal numbers consist of the digits 0 through 7 only. Decimal numbers consist of the digits 0 through 9 followed by a decimal point. If a number contains an 8 or 9 and is not followed by a decimal point, the N error code (see Section 2.10) will be printed and the number interpreted as decimal. Negative numbers may be expressed as a number preceded by a minus sign rather than in a two's complement form. Positive numbers may be preceded by a plus sign although this is not required.

If a number is too large to fit into 16 bits, the number is truncated from the left. In the assembly listing the statement will be flagged with a Truncation (T) error.
2.4.2 Arithmetic and Logical Operators

The arithmetic operators are:

+  indicates addition or a positive number
-
  indicates subtraction or a negative number

The logical operators are defined and illustrated below.

&  indicates the logical AND operation
!
  indicates the logical inclusive OR operation

<table>
<thead>
<tr>
<th>AND</th>
<th>OR</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 &amp; 0 = 0</td>
<td>0 ! 0 = 0</td>
</tr>
<tr>
<td>0 &amp; 1 = 0</td>
<td>0 ! 1 = 1</td>
</tr>
<tr>
<td>1 &amp; 0 = 0</td>
<td>1 ! 0 = 1</td>
</tr>
<tr>
<td>1 &amp; 1 = 1</td>
<td>1 ! 1 = 1</td>
</tr>
</tbody>
</table>

2.4.3 ASCII Conversion

When preceded by an apostrophe, any ASCII character (except null, rubout, carriage return, line feed, or form feed) is assigned the 7-bit ASCII value of the character (see Appendix A). For example,

'A'

is assigned the value 101 (octal).

When preceded by a quotation mark, two ASCII characters (not including null, rubout, carriage return, line feed, or form feed) are assigned the 7-bit ASCII values of each of the characters to be used. Each 7-bit value is stored in an 8-bit byte and the bytes are combined to form a word. For example, "AB will store the ASCII value of A in the high-order (even) byte and the value of B in the high-order (odd) byte:

high-order byte  |  low-order byte

B's value = 1 0 2  | 1 0 1 = A's value

0 100 001 001 000 001

0 4 1 1 0 1

"AB = 041101"

2.5 ASSEMBLY LOCATION COUNTER

The period (.) is the symbol for the assembly location counter. (Note difference of Program Counter. . = PC. See Section 2.6.) When used in the operand field of an instruction, it represents the address of the first word of the instruction. When used in the operand field of an assembler directive, it represents the address of the current byte or word. For example,
WRITING PAL-11A ASSEMBLY LANGUAGE PROGRAMS

A: MOV $, #RO ; REFERS TO LOCATION A, I.E.,
; THE ADDRESS OF THE MOV INSTRUCTION

(#) is explained in Section 2.6.9).

At the beginning of each assembly pass, the Assembler clears the
location counter. Normally, consecutive memory locations are assigned
to each byte of object data generated. However, the location where
the object data is stored may be changed by a direct assignment
altering the location counter.

.=expression

The expression defining the period must not contain forward references
or symbols that vary from one pass to another. Examples:

.=500

FIRST: MOV ,+10,COUNT ;THE LABEL FIRST HAS THE VALUE (OCTAL)
; +10 EQUALS 510 (OCTAL), THE CONTENTS
; OF THE LOCATION 510 (OCTAL) WILL BE DE-
; POSITED IN LOCATION COUNT.

.=520

; THE ASSEMBLY LOCATION COUNTER NOW
; HAS A VALUE OF 520 (OCTAL).

SECOND: MOV ,INDEX ; THE LABEL SECOND HAS THE VALUE 520 (OCTAL).
; THE CONTENTS OF LOCATION 520 (OCTAL),
; THAT IS, THE BINARY CODE FOR THE
; INSTRUCTION ITSELF, WILL BE DEPOSITED
; IN LOCATION INDEX.

Storage area may be reserved by advancing the location counter. For
example, if the current value of the location counter is 1000, the
direct assignment statement

.=.+100

will reserve 100 (octal) bytes of storage space in the program. The
next instruction will be stored at 1100.

2.6 ADDRESSING

The Program Counter (register 7 of the eight general registers) always
contains the address of the next word to be fetched; i.e., the
address of the next instruction to be executed, or the second or third
word of the current instruction.

In order to understand how the address modes operate and how they
assemble (see Section 2.6.11), the action of the Program Counter must
be understood. The key rule is:

Whenever the processor implicitly uses the Program Counter (PC) to
fetch a word from memory, the Program Counter is automatically
incremented by two after the fetch.

That is, when an instruction is fetched, the PC is incremented by two,
so that it is pointing to the next word in memory; and, if an
instruction uses indexing (see Sections 2.6.7, 2.6.8, and 2.6.10), the
processor uses the Program Counter to fetch the base from memory.
Hence, using the rule above, the PC increments by two, and now points
to the next word.
WRITING PAL-11A ASSEMBLY LANGUAGE PROGRAMS

The following conventions are used in this section:

a. Let E be any expression as defined in Section 3.4.

b. Let R be a register expression. This is any expression containing a term preceded by a % character or a symbol previously equated to such a term.

Examples:

R0 = %0 ;GENERAL REGISTER 0
R1 = R0 + 1 ;GENERAL REGISTER 1
R2 = 1 + %1 ;GENERAL REGISTER 2

c. Let ER be a register expression or an expression in the range 0 to 7 inclusive.

d. Let A be a general address specification which produces a 6-bit address field as described in the PDP-11 Handbook.

The addressing specification, A, may now be explained in terms of E, R, and ER as defined above. Each will be illustrated with the single operand instruction CLR or double operand instruction MOV.

2.6.1 Register Mode

The register contains the operand.

Format: R

Example:

R0 = %0 ;DEFINE R0 AS REGISTER 0
CLR R0 ;CLEAR REGISTER 0

2.6.2 Deferred Register Mode

The register contains the address of the operand.

Format: $R or {ER}

Example:

CLR $R1 ;CLEAR THE WORD AT THE
or
#ADDRESS CONTAINED IN
CLR (1) ;#REGISTER 1.

2.6.3 Autoincrement Mode

The contents of the register are incremented immediately after being used as the address of the operand.¹

¹a. Both JMP and JSR instructions using mode 2 may increment the register before or after its use, depending on what PDP-11 processor is being used. This mode should be avoided.

b. In double operand instructions of the addressing form $R, (R)+ or $R, -(R) where the source and destination registers are the same, the results may be different when executed on different PDP-11 processors. The use of these forms should be avoided!
Format: \((ER)\+)

Examples:

<table>
<thead>
<tr>
<th>CLR  ((R0)+)</th>
<th>$CLEAR WORDS AT ADDRESSES</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR  ((R0+3)+)</td>
<td>$CONTAINED IN REGISTERS 0, 3, AND 2 AND</td>
</tr>
<tr>
<td>CLR  ((2)+)</td>
<td>$INCREMENT REGISTER CONTENTS</td>
</tr>
<tr>
<td></td>
<td>$BY TWO.</td>
</tr>
</tbody>
</table>

2.6.4 Deferred Autoincrement Mode

The register contains the pointer to the address of the operand. The contents of the register are incremented after being used.

Format: \((ER)\+)

Example

<table>
<thead>
<tr>
<th>CLR  ((3)+)</th>
<th>$CONTENTS OF REGISTER 3 POINT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$TO ADDRESS OF WORD TO BE CLEARED</td>
</tr>
<tr>
<td></td>
<td>$BEFORE BEING INCREMENTED BY TWO</td>
</tr>
</tbody>
</table>

2.6.5 Autodecrement Mode

The contents of the register are decremented before being used as the address of the operand.1

Format: \(-(ER)\)

Examples:

<table>
<thead>
<tr>
<th>CLR  (-(R0))</th>
<th>$DECREMENT CONTENTS OF REGISTERS 0, 3, AND 2 BEFORE USING</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLR  (-(R0+3))</td>
<td>$AS ADDRESSES OF WORDS TO BE CLEARED</td>
</tr>
</tbody>
</table>

2.6.6 Deferred Autodecrement Mode

The contents of the register are decremented before being used as the pointer to the address of the operand.

Format: \(\neg-(ER)\)

Example:

<table>
<thead>
<tr>
<th>CLR  (\neg-(2))</th>
<th>$DECREMENT CONTENTS OF REG. 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$BEFORE USING AS POINTER TO ADDRESS</td>
</tr>
<tr>
<td></td>
<td>$OF WORD TO BE CLEARED</td>
</tr>
</tbody>
</table>

2.6.7 Index Mode

Format: \(E(ER)\)

The value of an expression E is stored as the second or third word of the instruction. The effective address is calculated as the value of E plus the contents of register ER. The value E is called the base.

---

1See previous footnote.
Examples:
CLR X+2(R1)  ; EFFECTIVE ADDRESS IS X+2 PLUS
             ; THE CONTENTS OF REGISTER 1
CLR -2(3)    ; EFFECTIVE ADDRESS IS -2 PLUS
             ; THE CONTENTS OF REGISTER 3

2.6.8 Deferred Index Mode
An expression plus the contents of a register gives the pointer to the
address of the operand.
Format:       @E(ER)
Example:
CLR @14(4)   ; IF REGISTER 4 HOLDS 100, AND LOCALLY:
             ; TATION 114 HOLDS 2000, LOC. 2000 IS
             ; CLEARED

2.6.9 Immediate Mode and Deferred Immediate (Absolute) Mode
The immediate mode allows the operand itself to be stored as the
second or third word of the instruction. It is assembled as an
autoincrement of register 7, the PC.
Format:       #E
Examples:
MOV  $100, R0 ; MOVE AN OCTAL 100 TO REGISTER 0
MOV  $X, R0  ; MOVE THE VALUE OF SYMBOL X TO
             ; REGISTER 0

The operation of this mode is explained as follows:
The statement MOV $100,R3 assembles as two words. These are:

0 1 2 7 0 3
0 0 0 1 0 0

Just before this instruction is fetched and executed, the PC points to
the first word of the instruction. The processor fetches the first
word and increments the PC by two. The source operand mode is 27
(autoincrement the PC). Thus, the PC is used as a pointer to fetch
the operand (the second word of the instruction) before being
incremented by two, to point to the next instruction.

If the #E is preceded by &, E specifies an absolute address.
2.6.10 Relative and Deferred Relative Modes

Relative Mode is the normal mode for memory references.

Format

Examples:

CLR 100  ;CLEAR LOCATION 100
MDV X,Y  ;MOVE CONTENTS OF LOCATION X TO LOCATION Y

This mode is assembled as Index Mode, using 7, the PC, as the register. The base of the address calculation, which is stored in the second or third word of the instruction, is not the address of the operand. Rather, it is the number which, when added to the PC, becomes the address of the operand. Thus, the base is X - PC. The operation is explained as follows.

If the statement MOV 100,R3 is assembled at location 20, then the assembled code is:

Location 20:  0 1 6 7 0 3
Location 22:  0 0 0 0 5 4

The processor fetches the MOV instruction and adds two to the PC so that it points to location 22. The source operand mode is 67; that is, indexed by the PC. To pick up the base, the processor fetches the word pointed to by the PC and adds two to the PC. The PC now points to location 24. To calculate the address of the source operand, the base is added to the designated register. That is, Base + PC = 54 + 24 = 100, the operand address.

Since the Assembler considers . as the address of the first word of the instruction, an equivalent statement would be

MOV 100-.4(PC),R3

This mode is called relative because the operand address is calculated relative to the current PC. The base is the distance (in bytes) between the operand and the current PC. If the operator and its operand are moved in memory so that the distance between the operator and data remains constant, the instruction will operate correctly.

If E is preceded by @, the expression's value is the pointer to the address of the operand.

2.6.11 Table of Mode Forms and Codes (6-bit (A) format only - see Section 3.7)

Each instruction takes at least one word. Operands of the first six forms listed below do not increase the length of an instruction. Each operand in one of the other forms however, increases the instruction length by one word.

<table>
<thead>
<tr>
<th>Form</th>
<th>Mode</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>None of these</td>
<td></td>
<td>Register</td>
</tr>
<tr>
<td>forms increase</td>
<td></td>
<td>Register n deferred</td>
</tr>
<tr>
<td>the instruction length.</td>
<td></td>
<td>Autoincrement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Autoincrement deferred</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Autodecrement</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Autodecrement deferred</td>
</tr>
</tbody>
</table>

2-13
Any of these forms adds a word to the instruction length:

\[
\begin{align*}
E(ER) & \quad 6n & \text{Index} \\
@E(ER) & \quad 7n & \text{Index deferred} \\
&E & \quad 27 & \text{Immediate} \\
@E & \quad 37 & \text{Absolute memory reference} \\
&E & \quad 67 & \text{Relative} \\
@E & \quad 77 & \text{Relative deferred reference}
\end{align*}
\]

Notes:

1. An alternate form for @R is (ER). However, the form @(ER) is equivalent to @0(ER).

2. The form @#E differs from the form E in that the second or third word of the instruction contains the absolute address of the operand rather than the relative distance between the operand and the PC. Thus, the statement CLR @#100 will clear location 100 even if the instruction is moved from the point at which it was assembled.

2.7 INSTRUCTION FORMS

The instruction mnemonics are given in Appendix B. This section defines the number and nature of the operand fields for these instructions.

In the table that follows, let R, E, and ER represent expressions as defined in Section 3.4, and let A be a 6-bit address specification of the forms:

\[
\begin{align*}
E & @E \\
R & @R \text{ or } (R) \\
(ER)+ & @,(ER)+ \\
-(ER) & @-(ER) \\
E(ER) & @E(ER) \\
@#E & \quad @#E
\end{align*}
\]

Table 2-1
Instruction Operand Fields

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Form</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Double Operand</td>
<td>Op A,A</td>
<td>MOV (R6)+,@Y</td>
</tr>
<tr>
<td>Single Operand</td>
<td>Op A</td>
<td>CLR -(R2)</td>
</tr>
<tr>
<td>Operate</td>
<td>Op</td>
<td>HALT</td>
</tr>
<tr>
<td>Branch</td>
<td>Op E</td>
<td>BR X+2, BLO -.4</td>
</tr>
<tr>
<td></td>
<td>\text{where } -128 \leq (E,-2)/2 \leq 127</td>
<td></td>
</tr>
<tr>
<td>Subroutine Call</td>
<td>JSR ER,A</td>
<td>JSR PC,SUBR</td>
</tr>
<tr>
<td>Subroutine Return</td>
<td>RTS ER</td>
<td>RTS PC</td>
</tr>
<tr>
<td>EMT/TRAP</td>
<td>Op or Op E</td>
<td>EMT</td>
</tr>
<tr>
<td></td>
<td>\text{where } 0 \leq E \leq 377\text{(octal)}</td>
<td>EMT 31</td>
</tr>
</tbody>
</table>
WRITING PAL-11A ASSEMBLY LANGUAGE PROGRAMS

The branch instructions are one word instructions. The high byte contains the op code and the low byte contains an 8-bit signed offset (7 bits plus sign) which specifies the branch address relative to the PC. The hardware calculates the branch address as follows:

a) Extend the sign of the offset through bits 8-15.

b) Multiply the result by 2. This creates a word offset rather than a byte offset.

c) Add the result to the PC to form the final branch address.

The Assembler performs the reverse operation to form the byte offset from the specified address. Remember that when the offset is added to the PC, the PC is pointing to the word following the branch instruction; hence the factor -2 in the calculation.

\[
\text{Byte offset} = (E-\text{PC})/2 \text{ truncated to eight bits.}
\]

Since PC = .+2, we have

\[
\text{Byte offset} = (E-.2)/2 \text{ truncated to eight bits.}
\]

The EMT and TRAP instructions do not use the low-order byte of the word. This allows information to be transferred to the trap handlers in the low-order byte. If EMT or TRAP is followed by an expression, the value is put into the low-order byte of the word. However, if the expression is too big (>377{octal}) it is truncated to eight bits and a Truncation (T) error occurs.

2.8 ASSEMBLER DIRECTIVES

Assembler directives (sometimes called pseudo-ops) direct the assembly process and may generate data. They may be preceded by a label and followed by a comment. The assembler directive occupies the operand field. Only one directive may be placed in any one statement. One or more operands may occupy the operand field or it may be void -- allowable operands vary from directive to directive.

2.8.1 .EOT

The .EOT directive indicates the physical End-of-Tape though not the logical end of the program. If the .EOT is followed by a single line feed or form feed, the Assembler will still read to the end of the tape, but will not process anything past the .EOT directive. If .EOT is followed by at least two line feeds or form feeds, the Assembler will stop before the end of the tape. Either case is proper, but it should be understood that even though it appears as if the Assembler has read too far, it actually hasn't.

If a .EOT is embedded in a tape, and more information to be assembled follows it, .EOT must be immediately followed by at least two line feeds or form feeds. Otherwise, the first line following the .EOT will be lost.

Any operands following a .EOT directive will be ignored. The .EOT directive allows several physically separate tapes to be assembled as one program. The last tape is normally terminated by a .END directive (see Section 3.8.3) but may be terminated with .EOT (see .END emulation in Section 3.9.4).
2.8.2 .EVEN

The .EVEN directive ensures that the assembly location counter is even by adding one if it is odd. Any operands following a .EVEN directive will be ignored.

2.8.3 .END

The .END directive indicates the logical and physical end of the source program. The .END directive may be followed by only one operand, an expression indicating the program's entry point.

At load time, the object tape will be loaded and program execution will begin at the entry point indicated by the .END directive. If the entry point is not specified, the Loader will halt after reading in the object tape.

2.8.4 .WORD

The .WORD assembler directive may have one or more operands, separated by commas. Each operand is stored in a word of the object program. If there is more than one operand, they are stored in successive words. The operands may be any legally formed expressions. For example,

```
   .=1420
   SAL=0
   .WORD 177535, +4, SAL ; STORED IN WORDS 1420, 1422, AND
   ; 1424 WILL BE 177535, 1426, AND 0.
```

Values exceeding 16 bits will be truncated from the left, to word length.

A .WORD directive followed by one or more void operands separated by commas will store zeros for the void operands. For example,

```
   .=1430
   .WORD ,5, ; ZERO, FIVE, AND ZERO ARE STORED
   ; IN WORDS 1430, 1432, AND 1434.
```

An operator field left blank will be interpreted as the .WORD directive if the operand field contains one or more expressions. The first term of the first expression in the operand field must not be an instruction or assembler directive unless preceded by a +, -, or one of the logical operators ! or &. For example,

```
   .=440
   LABEL: +MOV, LABEL ; THE OP-CODE FOR MOV, WHICH IS 010000,
   ; IS STORED IN LOCATION 440. 440 IS
   ; STORED IN LOCATION 442.
```

Note that the default .WORD will occur whenever there is a leading arithmetic or logical operator, or whenever a leading symbol is encountered which is not recognized as an instruction mnemonic or assembler directive. Therefore, if an instruction mnemonic or assembler directive is misspelled, the .WORD directive is assumed and errors will result. Assume that MOV is spelled incorrectly as MOR:

```
MOR A, B
```

Two error codes can result: a Q will occur because an expression operator is missing between MOR and A, and a U will occur if MOR is undefined. Two words will be generated; one for MOR A and one for B.
2.8.5 .BYTE

The .BYTE assembler directive may have one or more operands separated by commas. Each operand is stored in a byte of the object program. If multiple operands are specified, they are stored in successive bytes. The operands may be any legally formed expression with a result of 8 bits or less. For example,

```
SAM=5
.=410
.BYTE 48,.SAM

;STORED IN LOCATION 410 WILL BE
;060 (THE OCTAL EQUIVALENT OF 48).
;IN 411 WILL BE 005.
```

If the expression has a result of more than 8 bits, it will be truncated to its low-order 8 bits and will be flagged as a T error. If an operand after the .BYTE directive is left void, it will be interpreted as zero. For example,

```
.=420
.BYTE ,

;ZERO WILL BE STORED IN
;BYTES 420, 421 AND 422.
```

2.8.6 .ASCII

The .ASCII directive translates strings of ASCII characters into their 7-bit ASCII codes with the exception of null, rubout, carriage return, line feed, and form feed. The text to be translated is delimited by a character at the beginning and the end of the text. The delimiting character may be any printing ASCII character except colon and equal sign and those used in the text string. The 7-bit ASCII code generated for each character will be stored in successive bytes of the object program. For example,

```
.=500
.ASCII /YES/

;THE ASCII CODE FOR "Y" WILL BE
;STORED IN 500, THE CODE FOR "E"

.ASCII /5+3/2

;THE DELIMITING CHARACTER OCCURS
;AMONG THE OPERANDS. THE ASCII
;CODES FOR "5", "+", AND "3" ARE
;STORED IN BYTES 503, 504, AND
;505. 2/ IS NOT ASSEMBLED.
```

The ASCII directive must be terminated by a space or a tab.

2.9 OPERATING PROCEDURES

2.9.1 Introduction

The Assembler enables you to assemble an ASCII tape containing PAL-11A statements into an absolute binary tape. To do this, two or three passes are necessary. On the first pass the Assembler creates a table of user-defined symbols and their associated values, and a list of undefined symbols is printed on the teleprinter. On the second pass the Assembler assembles the program and punches out an absolute binary tape and/or outputs an assembly listing. During the third pass (this pass is optional) the Assembler punches an absolute binary tape or outputs an assembly listing. The symbol table (and/or a list of errors) may be output on any of these passes. The input and output devices as well as various options are specified during the initial dialogue (see Section 3.3.9). The Assembler initiates the dialogue immediately after being loaded and after the last pass of an assembly.
2.9.2 Loading PAL-11A

PAL-11A is loaded by the Absolute Loader (see Chapter 6 for operating procedures). Note that the start address of the Absolute Loader must be in the Switch Register when loading the Assembler. This is because the Assembler tape has an initial portion which clears all of Core up to the address specified in the Switch Register, and jumps to that address to start loading the Assembler.

2.9.3 Initial Dialogue

After being loaded, the Assembler initiates dialogue by printing on the teleprinter:

*S

meaning "What is the Source symbolic input device?" The response may be:

H  meaning High-speed reader
L  meaning Low-speed reader
T  meaning Teletype keyboard

If the response is T, the source program must be typed at the terminal once for each pass of the assembly and it must be identical each time it is typed.

The device specification is terminated, as is all user response, by typing the RETURN key.

If an error is made in typing at any time, typing the RUBOUT key will erase the immediately preceding character if it is on the current line. Typing CTRL/U will erase the whole line on which it occurs.

After the *S question and response, the Assembler prints:

*B

meaning "What is the Binary output device?" The responses to *B are similar to those for *S:

H  meaning High-speed punch
L  meaning Low-speed punch
\  meaning do not output binary tape
   (\ denotes typing the RETURN key)

In addition to I/O device specification, various options may be chosen. The binary output will occur on the second pass unless /3 (indicating the third pass) is typed following the H or L. Errors will be listed on the same pass if /E is typed. If /E is typed in response to more than one inquiry, only the last occurrence will be honored. It is strongly suggested that the errors be listed on the same pass as the binary output, since errors may vary from pass to pass. If both /3 and /E are typed, /3 must precede /E. The response is terminated by typing the RETURN key. Examples:
WRITING PAL-11A ASSEMBLY LANGUAGE PROGRAMS

*B L/E Binary output on the low-speed punch and the errors on the teleprinter, both during the second pass.

*B H/3/E Binary output on the high-speed punch and the errors on the teleprinter, both during the third pass.

Typing just the RETURN key will cause the Assembler to omit binary output.

After the *B question and response, the Assembler prints:

*L meaning "What is the assembly Listing output device?" The response to *L may be:

L meaning Low-speed punch (outputs a tab as a tab-rubout)
H meaning High-speed punch
T meaning Teleprinter (outputs a tab as multiple spaces)
P meaning line Printer (8K version only)
← meaning do not output listing (← denotes typing the RETURN key)

After the I/O device specification, pass and error list options similar to those for *B may be chosen. The assembly listing will be output to the third pass unless /2 (indicating the second pass) is typed following H, L, T, or P. Errors will be listed on the teleprinter during the same pass if /E is typed. If both /2 and /E are typed, /2 must precede /E. The response is terminated by typing the RETURN key. Examples:

*L L/2/E Listing on low-speed punch and errors on teleprinter during second pass.

*L H Listing on high-speed punch during third pass.

*L ← The RETURN key alone will cause the Assembler to omit listing output.

After the *L question and response, the final question is printed on the teleprinter:

*T meaning "What is the symbol Table output device?" The device specification is the same as for the *L question. The symbol table will be output at the end of the first pass unless /2 or /3 is typed in response to *T. The first tape to be assembled should be placed in the reader before typing the RETURN key because assembly will begin upon typing the RETURN key in response to the *T question. The /E option is not a meaningful response to *T. Example:

*T T/3 Symbol table output on teleprinter at end of third pass.

*T ← Typing just the RETURN key will cause the Assembler to omit the symbol table output.
WRITING PAL-11A ASSEMBLY LANGUAGE PROGRAMS

The symbol table is printed alphabetically, four symbols per line. Each symbol printed is followed by its identifying characters and by its value. If the symbol is undefined, six asterisks replace its value. The identifying characters indicate the class of the symbol; that is, whether it is a label, direct-assignment, register symbol, etc. The following examples show the various forms:

- ABCDEF = 001244 (Defined label)
- R3 = %000003 (Register symbol)
- DIRASM = 177777 (Direct assignment)
- XYZ = ***** (Undefined direct assignment)
- R6 = %***** (Undefined register symbol)
- LABEL = ***** (Undefined label)

Generally, undefined symbols (including labels) will be listed as undefined direct assignments.

Multiply-defined symbols are not flagged in the symbol table printout but they are flagged wherever they are used in the program.

It is possible to output both the binary tape and the assembly listing on the same pass, thereby reducing the assembly process to two passes (see Example 1 below). This will happen automatically unless the binary device and the listing device are conflicting devices or the same device (see Example 2 below). The only conflicting devices are the teleprinter and the low-speed punch. Even though the Assembler deduces that three passes are unnecessary, the binary and listing can be forced on pass 2 by including /2 in the responses to *B and *L (see Example 3 below).

Example 1. Runs 2 passes:

- *S H High-speed reader
- *B H High-speed punch
- *L P Line Printer
- *T T Teleprinter

Example 2. Runs 3 passes:

- *S H High-speed reader
- *B H High-speed punch
- *L H High-speed punch
- *T T Teleprinter

Example 3. Runs 2 passes:

- *S H High-speed reader
- *B H/2 High-speed punch on pass 2
- *L H/2 High-speed punch on pass 2
- *T T Teleprinter
WRITING PAL-11A ASSEMBLY LANGUAGE PROGRAMS

Note that there are several cases where the binary output can be intermixed with ASCII output:

a. \*B H/2 \*L H/2 Binary and listing to punch on pass 2
b. \*B L/E Binary to low-speed punch and error listing to teleprinter (and low-speed punch)
c. \*B L/2/E \*L T/2 Binary, error listing, and listing to low-speed punch.

The binary so generated is loadable by the Absolute Loader as long as there are no CTRL/A characters in the source program. The start of every block on the binary tape is indicated by a 001 and the Absolute Loader ignores all information until a 001 is detected. Thus, all source and/or error messages will be ignored if they do not contain any CTRL/A characters (octal 001).

If a character other than those mentioned is typed in response to a question, the Assembler will ignore it and print the question again. Example:

\*S H High-speed reader
\*B Q Q is not a valid response
\*B The question is repeated

If at any time you wish to restart the Assembler, type CTRL/P.

When no passes are omitted or error options specified, the Assembler performs as follows:

PASS 1: Assembler creates a table of user-defined symbols and their associated values to be used in assembling the source to object program. Undefined symbols are listed on the teleprinter at the end of the pass. The symbol table is also listed at this time. If an illegal location statement of the form .=expression is encountered, the line and error code will be printed out on the teleprinter before the assembly proceeds. An error in a location statement is usually a fatal error in the program and should be corrected.

PASS 2: Assembler punches the object tape, and prints the pass error count and undefined location statements on the teleprinter.

PASS 3: Assembler prints or punches the assembly program listing, undefined location statements, and the pass error count on the teleprinter.

The functions of passes 2 and 3 will occur simultaneously on pass 2 if the binary and listing devices are different, and do not conflict with each other (low-speed punch and Teleprinter conflict).
WRITING PAL-11A ASSEMBLY LANGUAGE PROGRAMS

The following table summarizes the initial dialogue questions:

<table>
<thead>
<tr>
<th>Printout</th>
<th>Inquiry</th>
</tr>
</thead>
<tbody>
<tr>
<td>*S</td>
<td>What is the input device of the Source symbolic tape?</td>
</tr>
<tr>
<td>*B</td>
<td>What is the output device of the Binary object tape?</td>
</tr>
<tr>
<td>*L</td>
<td>What is the output device of the assembly Listing?</td>
</tr>
<tr>
<td>*T</td>
<td>What is the output device of the symbol Table?</td>
</tr>
</tbody>
</table>

The following table summarizes the legal responses:

<table>
<thead>
<tr>
<th>Character</th>
<th>Response Indicated</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Teletype keyboard or printer</td>
</tr>
<tr>
<td>L</td>
<td>Low-speed reader or punch</td>
</tr>
<tr>
<td>H</td>
<td>High-speed reader or punch</td>
</tr>
<tr>
<td>P</td>
<td>Line Printer (8K version only)</td>
</tr>
<tr>
<td>/1</td>
<td>Pass 1</td>
</tr>
<tr>
<td>/2</td>
<td>Pass 2</td>
</tr>
<tr>
<td>/3</td>
<td>Pass 3</td>
</tr>
<tr>
<td>/E</td>
<td>Errors listed on same pass (not meaningful in response to *S or *T)</td>
</tr>
<tr>
<td>/O</td>
<td>Omit function</td>
</tr>
</tbody>
</table>

Typical examples of complete initial dialogues:

For minimal PDP-11 configuration:

- *S L Source input on low-speed reader
- *B L/E Binary output on low-speed punch
  Errors during same (second) pass
- *L T Listing on teleprinter during pass 3
- *T T Symbol table on teleprinter at end of pass 1

For a PDP-11 with high-speed I/O devices:

- *S H Source input on high-speed reader
- *B H/E Binary output on high-speed punch,
  Errors during same (second) pass
- *L No listing
- *T T/2 Symbol table on teleprinter at end of pass 2

2-22
2.9.4 Assembly Dialogue

During assembly, the Assembler will pause to print on the teleprinter various messages to indicate that you must respond in some way before the assembly process can continue. You may also type CTLR/P, at any time, if you wish to stop the assembly process and restart the initial dialogue, as mentioned in the previous section.

When a .EOT assembler directive is read on the tape, the assembler prints:

EOF ??

and pauses. During this pause, the next tape is placed in the reader, and RETURN is typed to continue the assembly.

If the specified assembly listing output device is the high-speed punch and if it is out of tape, or if the device is the Line Printer and is out of paper, the Assembler prints on the teleprinter:

EOM ??

and waits for tape or paper to be placed in the device. Type the RETURN key when the tape or paper has been replenished; assembly will continue.

Conditions causing the EOM? message for an assembly listing device are:

<table>
<thead>
<tr>
<th>HSP</th>
<th>LPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>No power</td>
<td>No power</td>
</tr>
<tr>
<td>No tape</td>
<td>Printer drum gate open</td>
</tr>
<tr>
<td></td>
<td>Too hot</td>
</tr>
<tr>
<td></td>
<td>No paper</td>
</tr>
</tbody>
</table>

There is no EOM if the line printer is switched off-line, although characters may be lost for this condition as well as for an EOM. If the binary output device is the high-speed punch and if it is out of tape, the Assembler prints:

EOM ??

The assembly process is aborted and the initial dialogue is begun again.

When a .END assembler directive is read on the tape, the Assembler prints:

END ??

and pauses. During the pause the first tape is placed in the reader, and the RETURN key is typed to begin the next pass. On the last pass, the .END directive causes the Assembler to begin the initial dialogue for the next assembly.

If you are starting the binary pass and the binary is to be punched on the low-speed punch, turn the punch on before typing the RETURN key for starting the pass. The carriage return and line feed characters will be punched onto the binary tape, but the Absolute Loader will ignore them.
If the last tape ends with a .EOT, the Assembler may be told to emulate a .END assembler directive by responding with E followed by the RETURN key. The Assembler will then print:

    END ?

and wait for another RETURN before starting the next pass. Example:

    EOF ?   E
    END ?

**NOTE**

When a .END directive is emulated with an E response to the EOF? message, the error counter is incremented.

To avoid incrementing the error counter, place a paper tape containing only the line .END in the reader and press the RETURN key instead of using the E response.

2.9.5 Assembly Listing

PAL-11A produces a side-by-side assembly listing of symbolic source statements, their octal equivalents, assigned absolute addresses, and error codes as follows:

```
    EELLLLLL 000000 SSS....S
       000000
       000000
```

The E's represent the error field. The L's represent the absolute address. The O's represent the object data in octal. The S's represent the source statement. While the Assembler accepts 72(decimal) characters per line on input, the listing is reduced by the 16 characters to the left of the source statement.

The above represents a three-word statement. The second and third words of the statement are listed under the command word. No addresses precede the second and third word since the address order is sequential.

The third line is omitted for a two-word statement; both second and third lines are omitted for a one-word statement.

For a .BYTE directive, the object data field is three octal digits.

For a direct assignment statement, the value of the defining expression is given in the object code field although it is not actually part of the code of the object program.

Each page of the listing is headed by a page number.
2.10 ERROR CODES

The error codes printed beside the octal and symbolic code in the assembly listing have the following meanings:

<table>
<thead>
<tr>
<th>Error Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Addressing error. An address within the instruciton is incorrect.</td>
</tr>
<tr>
<td>B</td>
<td>Bounding error. Instructions or word data are being assembled at an odd address in memory. The location counter is updated by +1.</td>
</tr>
<tr>
<td>D</td>
<td>Doubly-defined symbol referenced. Reference was made to a symbol which is defined more then once.</td>
</tr>
<tr>
<td>I</td>
<td>Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing.</td>
</tr>
<tr>
<td>L</td>
<td>Line buffer overflow. Extra characters on a line (more than 72(decimal)) are ignored.</td>
</tr>
<tr>
<td>M</td>
<td>Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a previously encountered label.</td>
</tr>
<tr>
<td>N</td>
<td>Number containing 8 or 9 has no decimal point.</td>
</tr>
<tr>
<td>P</td>
<td>Phase error. A label's definition or value varies from one pass to another.</td>
</tr>
<tr>
<td>Q</td>
<td>Questionable syntax. There are missing arguments or the instruction scan was not completed or a carriage return was not immediately followed by a line feed or form feed.</td>
</tr>
<tr>
<td>R</td>
<td>Register-type error. An invalid use of or reference to a register has been made.</td>
</tr>
<tr>
<td>S</td>
<td>Symbol table overflow. When the quantity of user-defined symbols exceeds the allocated space available in the user's symbol table, the assembler outputs the current source line with the S error code, then returns to the initial dialogue.</td>
</tr>
<tr>
<td>T</td>
<td>Truncation error. A number generated more than 16 bits of significance or an expression generated more than 8 bits of significance during the use of the .BYTE directive.</td>
</tr>
<tr>
<td>U</td>
<td>Undefined symbol. An undefined symbol was encountered during the evaluation of an expression. Relative to the expression, the undefined symbol is assigned a value of zero.</td>
</tr>
</tbody>
</table>
2.11 SOFTWARE ERROR HALTS

PAL-11A loads all unused trap vectors with the code

```
.WORD .+2,HALT
```

so that if the trap does occur, the processor will halt in the second word of the vector. The address of the halt, displayed in the console address register, therefore indicates the cause of the halt. In addition to the halts which may occur in the vectors, the standard IOX error halt at location 40 may occur (see Chapter 7).

<table>
<thead>
<tr>
<th>Address of Halt</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Reserved instruction executed</td>
</tr>
<tr>
<td>16</td>
<td>Trace trap occurred</td>
</tr>
<tr>
<td>26</td>
<td>Power fail trap</td>
</tr>
<tr>
<td>32</td>
<td>EMT executed</td>
</tr>
<tr>
<td>40</td>
<td>IOX detected error</td>
</tr>
</tbody>
</table>

See Appendix B for summaries of PAL-11A features.
CHAPTER 3
LINK-11S LINKER

3.1 INTRODUCTION

3.1.1 General Description

LINK-11S (stand alone) is a PDP-11 system program designed to link and relocate programs previously assembled by PAL-11S. The user can separately assemble the main program and each of its various subroutines without assigning an absolute load address at assembly time. The binary output of assembly (called an object module) is processed by LINK-11S to:

1. Relocate each object module and assign absolute addresses.
2. Link the modules by correlating global symbols defined in one module and referenced in other modules.
3. Print a load map which displays the assigned absolute addresses.
4. Punch a load module which can subsequently be loaded (by the Absolute Loader) and executed.

Some of the advantages of using PAL-11S and LINK-11S are:

1. The program is divided into segments (usually subroutines) which are assembled separately. If an error is discovered in one segment, only that segment needs to be reassembled. The new object module is then linked with the other object modules.
2. Absolute addresses need not be assigned at assembly time. The Linker automatically assigns absolute addresses. This keeps programs from overlaying each other. This also allows subroutines to change size without influencing the placement of other routines.
3. Separate assemblies allow the total number of symbols to exceed the number allowed in a single assembly.
4. Internal symbols (symbols which are not global) need not be unique among object modules. Thus, naming rules are required only for global symbols when separate programmers prepare separate subroutines of a single program.
5. Subroutines may be provided for general use in object module form to be linked into the user's program.
LINK-11S LINKER

LINK-11S is designed to run on an 8K PDP-11 with an ASR-33. A PCll (high speed paper tape reader and punch) and an LPll (line printer) may be used if available. The PCll significantly speeds up the linking process. An LPll provides a fast device for the load map listing.

3.1.2 Absolute and Relocatable Program Sections

A program assembled by PAL-11S may consist of an absolute program section, declared by the .ASECT assembler directive, and a relocatable program section, declared by the .CSECT assembler directive. (If a program has neither an .ASECT or .CSECT directive, the assembler implicitly assumes a .CSECT directive.) The program and data in the absolute section are assigned absolute addresses as specified by the location counter setting statements (.=x). The Linker assigns absolute addresses to the program and data in the relocatable section. Addresses are normally assigned such that the relocatable section is at the high end of memory. The assignment of addresses may be influenced by command string options (see Section 3.3.2).

The Linker appropriately modifies all instructions and/or data as necessary to account for the relocation of the control section.

LINK-11S can handle object modules containing named control (relocatable) sections as generated by PAL-11R. However, PAL-11S can create only the unnamed control section (which has the special default name of 6 blanks) and the absolute section (with the special name .ABS.). The unnamed control section is internal to each object module. That is, every object module may have an unnamed control section (each with the name 6 blanks) but the Linker treats them independently. Each is assigned an absolute address such that they occupy mutually exclusive areas of memory. Named control sections, on the other hand, are treated globally. That is, if different object modules each have control sections with the same name, they are all assigned the same absolute load address and the size of the area reserved or loading of the section is the maximum of the sizes of each section. Thus, named control sections allow the sharing of data and/or programs among object modules. This is very similar to the handling and function of labelled COMMON in FORTRAN IV. A restriction of LINK-11S is that the name of a control section must not be the same as the name of a global entry symbol, as this results in multiple definition errors.

3.1.3 Global Symbols

Global symbols provide the links for communication between object modules (or assemblies). Global symbols are created with the .GLOBL assembler directive. Symbols which are not global are called internal symbols. If the global symbol is defined (as a label or direct assignment) in an object module it is called an entry symbol, and other object modules may reference it. If the global symbol is not defined in the object module it is an external symbol. It is assumed to be defined (as an entry symbol) in some other object module.

As the Linker reads the object modules it records all the global symbol definitions and references. It then modifies the instructions and/or data that reference the global symbols.

3-2
LINK-11S LINKER

3.2 INPUT AND OUTPUT

3.2.1 Object Module

Input to LINK-11S is the object module. This is the output of PAL-11S (or any other program which can create an object module). The Linker reads each object module twice; that is, it is a two-pass processor.

On pass 1, the Linker reads each object module to gather enough information to assign absolute addresses to all relocatable sections and absolute values to all globals. This information appears in the global symbol directory (GSD) of the object module.

On pass 2, the Linker reads all of each object module and produces the load module (see Section 3.2.2). The data gathered on pass 1 guides the relocation and linking process on pass 2.

3.2.2 Load Modules

The normal output of the Linker is a load module which may be loaded and run.

A load module consists of formatted binary blocks holding absolute load addresses and object data as specified for the Paper Tape System Absolute Loader and the PDF-11 Disk Monitor. The first few words of data are the communications directory (COMD) and have an absolute load address equal to the lowest relocated address of the program. The absolute loader loads the COMD at the specified address but the program subsequently overlays it. The disk monitor loader expects the COMD and loads it where the monitor wants it. The end of the load module is indicated by a TRA block; that is, a block containing only a load address. The byte count in the formatted binary block is 6 on this block; on all other blocks the byte count is larger than 6. The TRA (transfer address) is selected by the Linker to be the first even transfer address seen. Thus, if four object modules are linked together and if the first and second had a .END statement, the third had a .END A and the fourth had a .END B, the transfer address would be A of module three.

The overlaying of the COMD by the relocated program is a trick to allow the Absolute Loader to handle load modules with a COMD. However, a problem arises if a load module is to be loaded by the absolute loader and either of the following conditions exists:

a. The object modules used to construct the load module contained no relocatable code; or

b. The total size of the relocatable code is less than 20 (decimal) bytes (the size of the COMD).

In either case, there is not enough relocatable code to overlay the COMD which means the COMD will load into parts of memory not intended to be altered by the user. The COMD's load address, selected by the Linker in the above cases, is such that it will be up against the current top of memory (see *T option in section 3.3.1). If the top happens to be very low, the Linker does not allow the COMD to be loaded below address 0; it loads it at 0.
3.2.3 Load Map

The load map provides several types of information concerning the load module's make-up. The map begins with an indication of the low and high limits of the relocatable code and the transfer address. Then there is a section of the map for each object module included in the linking process. Each of these sections begins with the module name followed by a list of the control sections and the entry points for each control section. For each control section, the base of the section (its low address) and its size (in bytes) is printed to the right of the section name (enclosed in angle brackets). Following each section name is a list of entry points and their addresses. After all information has been printed for each object module, any undefined symbols are listed. Note that modules are loaded such that if modules A, B and C are linked together, A is lowest and C is highest in memory.

The format is quite self-explanatory as can be seen from the following example:

LOAD MAP

TRANSFER ADDRESS: 037434
LOW LIMIT: 037406
HIGH LIMIT: 037460

******

MODULE MOD1
SECTION ENTRY ADDRESS SIZE
< . ABS. > 000000 000000
< > 037406 000044
  X3 037452
  X4 037440
  X5 037450
  X7 037430

******

MODULE MOD2
SECTION ENTRY ADDRESS SIZE
< > 037452 000006
  X1 037452
  X2 037452

******

UNDEFINED REFERENCES
X6

PASS 2

*
3.3 OPERATING PROCEDURES

3.3.1 Loading and Command String

The Linker is loaded by the Absolute Loader and is self-starting. It uses a simple command dialogue which allows the object module, load module and load map devices to be specified. During pass 1 and pass 2, the Linker asks for each object module individually.

Operation begins by the linker typing its name and version. This is followed by the input option printed as *IΔ. The responses are:

- Read object module from HSR.
- Read object module from LSR.

The input option is followed by the output option *OΔ. The responses are:

- Punch load module on HSP.
- Punch load module on LSP.

LINK-11S asks if a load map is desired by typing *MΔ. The legal responses are / for no map, TΔ or HΔ or PΔ for a map on the teleprinter, high-speed punch, or line printer, respectively.

The next two options concern the placement of the relocated object program in memory. The standard version of the Linker assumes it is linking for an 8K machine. It relocates the program such that it is as high as possible in 8K but leaves room for the Absolute and Boot Loaders. These assumed values may be changed by altering parameters HGHMEM (highest legal memory address +1) and ALODSZ (number of bytes allocated for Absolute Loader and Boot Loader) and reassembling the Linker. The user may control where a program is relocated with the *T and *B options. After the option *TΔ has been typed, the user may respond as follows:

- Relocate so that program is up against the current top of memory. If the top has not been changed, then the top is the assembled-in top (HGHMEM-ALODSZ). The standard assumption is 16272 decimal (16384-112) or 37460 octal.

- NΔ

N is an octal number (unsigned) which defines a new top address.

If a new top is specified, the *B option is suppressed.

After the option *BΔ has been printed the user may respond as follows:

- Use current top of memory.

- NΔ

N is an unsigned octal number which defines the bottom address of the program. That is, a new top of memory is calculated so that the bottom of the program corresponds with N.

Once a top of memory has been calculated (by *T or *B), that value is used until it is changed.

3-5
LINK-11S LINKER

LINK-11S indicates the start of pass one by typing PASS 1. The input is requested by the Linker, one tape at a time, by typing ₤. The legal responses are:

Read a tape and request more input.

List all undefined globals on the teleprinter and request more input.

End of input. If there are undefined globals, list them on the teleprinter and request more input. Otherwise print the load map, if requested, and enter pass 2.

End of input. Assign 0 to any undefined globals, print the load map (if requested), and enter pass 2.

The Linker indicates the start of pass 2 by typing PASS 2. It then requests each input tape as in pass 1.

A carriage return is the only useful response to ₤ on pass 2. The modules must be read on pass 2 in the same order as pass 1. When the last module has been read the Linker automatically finishes the load module and restarts itself.

Leader and trailer are punched on the load module.

If the LSP is being used for the load module output, it should be turned on before pass 2 begins. Thus, turn it on before typing ₤ or ₤. The echo of these characters (and the load map, if printed on the TTY) is punched on the load module but may be easily removed since leader is punched on the load module. In any case, ASCII information in a load module is ignored by the Absolute and Disk Monitor loaders. However, the LSP can be turned on while leader is being punched (after the linker has typed PASS 2) to keep the load map, etc., from being punched onto the tape.

Note:

On all command string options, except for ₤T and ₤B, the linker examines only the last character typed preceding the carriage return. Thus,

ABCDEFGH

is equivalent to H

3.3.1.1 Operational Cautions - The Linker does not give a warning if a program is linked so low in memory that it goes below address 0. However, this case is easily seen by examining the low and high limits which are always printed (on the load map or on the teleprinter).

The Linker reads object modules until an end of medium is detected. Object modules from the DEC Program Library contain a special checksum at the end of the tape which must be removed before they are linked. Failure to remove this checksum can result in fatal Linker errors.
3.3.2 Error Procedure and Messages

3.3.2.1 Restarting - CTRL/P (symbolized as ^P) is used for two purposes by LINK-11. If a ^P is typed while a load map is being printed, the load map is aborted and the Linker continues. A ^P typed at any other time causes the Linker to restart itself.

3.3.2.2 Non-Fatal Errors -

1. Non-unique object module name - this error is detected during pass 1; an error message is issued and the module is rejected. The message is:

   ?MODULE NAME XXXXXX NOT UNIQUE

   The Linker then asks for more input.

2. Load map device EOM - this error allows the user to fix the device and continue or abort the map listing. The Linker prints:

   ?MAP DEVICE EOM,
   TYPE <CR> TO CONTINUE

   Any response, terminated by / or + causes the Linker to continue. A + P causes the map to be aborted.

3. A byte relocation error - the Linker tries to relocate and link byte quantities. However, relocation usually fails and linking may fail. Failure is defined as the high byte of the relocated value (or the linked value) not being all zero. In such a case, the value is truncated to 8 bits and the following message is printed:

   ?BYTE RELOC ERROR AT ABS ADDRESS XXXXXX.

   The Linker automatically continues.

4. If the object modules are not read in the same order on pass 2 as pass 1, the Linker indicates which module should be loaded next by typing:

   ?LOAD XXXXXX NEXT!

   The linker then asks for more input.

5. Multiply-Defined Globals - this results in the following error message during pass 1:

   ?XXXXXXX MULTIPLY DEFINED BY MODULE XXXXXX.

   The second definition is ignored and the Linker continues.

3.3.2.3 Fatal Errors - Each of the following errors causes the indicated error message to be printed and the Linker to be restarted.

   1. Symbol Table overflow - the message is:

      ?SYMBOL TABLE OVERFLOW - MODULE XXXXXX, SYMBOL XXXXXX
LINK-11S LINKER

2. System Errors - this class of errors prints:

?SYSTEM ERROR xx

where xx is an identifying number as follows:

<table>
<thead>
<tr>
<th>Number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Unrecognized symbol table entry was found.</td>
</tr>
<tr>
<td>02</td>
<td>A relocation directory references a global name which cannot be found in the symbol table.</td>
</tr>
<tr>
<td>03</td>
<td>A relocation directory contains a location counter modification command which is not last.</td>
</tr>
<tr>
<td>04</td>
<td>Object module does not start with a GSD.</td>
</tr>
<tr>
<td>05</td>
<td>The first entry in the GSD is not the module name.</td>
</tr>
<tr>
<td>06</td>
<td>An RLD references a section name which cannot be found.</td>
</tr>
<tr>
<td>07</td>
<td>The TRA specification references a non-existent module name.</td>
</tr>
<tr>
<td>08</td>
<td>The TRA specification references a non-existent section name.</td>
</tr>
<tr>
<td>09</td>
<td>An internal jump table index is out of range.</td>
</tr>
<tr>
<td>10</td>
<td>A checksum error occurred on the object module.</td>
</tr>
<tr>
<td>11</td>
<td>An object module binary block is too big (more than 64 decimal words of data).</td>
</tr>
<tr>
<td>12</td>
<td>A device error occurred on the load module output device.</td>
</tr>
</tbody>
</table>

All system errors except for numbers 10 and 12 indicate a program failure either in the Linker or the program which generated the object module. Error 05 can occur if a tape is read which is not an object module.

3.3.2.4 **Error HALTs** - LINK-11S loads all of its unused trap vectors with the code:

```
.WORD .+2, HALT
```

so that if the trap occurs, the processor halts in the second word of the vector. The address of the halt, displayed in the console lights, therefore indicates the cause of the halt.
3.3.2 Error Procedure and Messages

3.3.2.1 Restarting - CTRL/P (symbolized as ^P) is used for two purposes by LINK-11. If a ^P is typed while a load map is being printed, the load map is aborted and the Linker continues. A ^P typed at any other time causes the Linker to restart itself.

3.3.2.2 Non-Fatal Errors -

1. Non-unique object module name - this error is detected during pass 1; an error message is issued and the module is rejected. The message is:

   ?MODULE NAME XXXXXXX NOT UNIQUE

The Linker then asks for more input.

2. Load map device EOM - this error allows the user to fix the device and continue or abort the map listing. The Linker prints:

   ?MAP DEVICE EOM.
   TYPE <CR> TO CONTINUE

Any response, terminated by \ or + causes the Linker to continue. A + P causes the map to be aborted.

3. A byte relocation error - the Linker tries to relocate and link byte quantities. However, relocation usually fails and linking may fail. Failure is defined as the high byte of the relocated value (or the linked value) not being all zero. In such a case, the value is truncated to 8 bits and the following message is printed:

   ?BYTE RELOC ERROR AT ABS ADDRESS XXXXXXX.

The Linker automatically continues.

4. If the object modules are not read in the same order on pass 2 as pass 1, the Linker indicates which module should be loaded next by typing:

   ?LOAD XXXXXX NEXT!

The linker then asks for more input.

5. Multiply-Defined Globals - this results in the following error message during pass 1:

   ?XXXXXXX MULTIPLY DEFINED BY MODULE XXXXXXX.

The second definition is ignored and the Linker continues.

3.3.2.3 Fatal Errors - Each of the following errors causes the indicated error message to be printed and the Linker to be restarted.

1. Symbol Table overflow - the message is:

   ?SYMBOL TABLE OVERFLOW - MODULE XXXXXXX, SYMBOL XXXXXXX
2. System Errors - this class of errors prints:

\texttt{?SYSTEM\ ERROR \ xx}

where \texttt{xx} is an identifying number as follows:

<table>
<thead>
<tr>
<th>Number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>Unrecognized symbol table entry was found.</td>
</tr>
<tr>
<td>02</td>
<td>A relocation directory references a global name which cannot be found in the symbol table.</td>
</tr>
<tr>
<td>03</td>
<td>A relocation directory contains a location counter modification command which is not last.</td>
</tr>
<tr>
<td>04</td>
<td>Object module does not start with a GSD.</td>
</tr>
<tr>
<td>05</td>
<td>The first entry in the GSD is not the module name.</td>
</tr>
<tr>
<td>06</td>
<td>An RLD references a section name which cannot be found.</td>
</tr>
<tr>
<td>07</td>
<td>The TRA specification references a non-existent module name.</td>
</tr>
<tr>
<td>08</td>
<td>The TRA specification references a non-existent section name.</td>
</tr>
<tr>
<td>09</td>
<td>An internal jump table index is out of range.</td>
</tr>
<tr>
<td>10</td>
<td>A checksum error occurred on the object module.</td>
</tr>
<tr>
<td>11</td>
<td>An object module binary block is too big (more than 64 decimal words of data).</td>
</tr>
<tr>
<td>12</td>
<td>A device error occurred on the load module output device.</td>
</tr>
</tbody>
</table>

All system errors except for numbers 10 and 12 indicate a program failure either in the Linker or the program which generated the object module. Error 05 can occur if a tape is read which is not an object module.

3.3.2.4 Error HALTS - LINK-11S loads all of its unused trap vectors with the code:

```
.WORD .+2, HALT
```

so that if the trap occurs, the processor halts in the second word of the vector. The address of the halt, displayed in the console lights, therefore indicates the cause of the halt.
## LINK-11S LINKER

<table>
<thead>
<tr>
<th>Address of HALT (octal)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Reserved instruction executed.</td>
</tr>
<tr>
<td>16</td>
<td>Trace trap occurred.</td>
</tr>
<tr>
<td>26</td>
<td>Power fail trap.</td>
</tr>
<tr>
<td>32</td>
<td>EMT executed.</td>
</tr>
</tbody>
</table>

A halt at address 40 indicates an IOXLPT detected error. R0 (displayed in the console lights) contains an identifying code:

<table>
<thead>
<tr>
<th>Code in R0</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Illegal memory reference, SP overflow or illegal instruction.</td>
</tr>
<tr>
<td>1</td>
<td>Illegal IOX command.</td>
</tr>
<tr>
<td>2</td>
<td>Slot number out of range.</td>
</tr>
<tr>
<td>3</td>
<td>Device number illegal.</td>
</tr>
<tr>
<td>4</td>
<td>Referenced slot not INIT ed.</td>
</tr>
<tr>
<td>5</td>
<td>Illegal data mode.</td>
</tr>
</tbody>
</table>

IOXLPT also sets R1 as follows:

If the error code is 0, R1 contains the PC at the time of the error.

If the error code is 1-5, R1 points to some element in the IOT argument list or to the instruction following the argument list, depending on whether IOXLPT has finished decoding all the arguments when it detects the error.
CHAPTER 4

EDITING THE SOURCE PROGRAM

The PDP-ll Text Editor program (ED-ll) enables you to display your source program (or any text) on the teleprinter, make corrections or additions to it, and punch all or any portion of the program on paper tape. This is accomplished by typing simple one-character commands on the keyboard.

The Editor commands can be grouped according to function:

1. input/output;
2. searching for strings of characters;
3. positioning the current character location pointer;
4. inserting, deleting, and exchanging text portions.

All input/output functions are handled by IOX, the PDP-ll Input/Output Executive (see Chapter 7).

4.1 COMMAND MODE AND TEXT MODE

Whenever ED-ll prints an * on the teleprinter, you may type a command to it. (Only one command per line is acceptable.) The Editor is then said to be in Command Mode. While most commands operate exclusively in this mode, there are five ED-ll commands that require additional information in order for the commands to be carried out. The Editor goes into Text Mode to receive this text.

Should a nonexistent command be typed or a command appear in incorrect format, ED-ll prints a ?. This is followed by an * at the beginning of a new line indicating that the Editor is in Command Mode.

Editor processing begins in Command Mode. When you type a command, no action occurs until you follow it by typing the RETURN key (symbolized as ↵). If the command is not a text-type command, typing the RETURN key initiates the execution of the command and ED-ll remains in Command Mode. However, if the command is a text-type command (Insert, eXchange, Change, Get, or wHole), typing the RETURN key causes the Editor to go into Text Mode. At this time you should type the text to be operated on by the command. This can include the non-printing characters discussed below, as well as spaces and tabs (up to eight spaces generated by the CTRL/TAB keys).

Note that typing the RETURN key always causes the physical return of the Teletype print element to the beginning of the line, and automatically generates a line feed, thereby advancing the carriage to a new line. In Text Mode, the RETURN key not only serves these
mechanical functions, allowing you to continue typing at the beginning of a new line, but at the same time it enters a carriage return and line feed character into the text. (A carriage return not followed by a line feed cannot, therefore, be entered from the keyboard.)

RETURN and LINE FEED are both counted as characters and can be edited along with the printing characters (as can the form feed, discussed in Section 4.2.5). When you wish to terminate Text Mode and reenter Command Mode, you must type the LINE FEED key symbolized as \textbackslash D. A typed LINE FEED is not considered to be part of the text unless it is the first character entered in Text Mode.

4.2 COMMAND DELIMITERS

4.2.1 Arguments

Some ED-11 commands require an argument to specify the particular portion of text to be affected by the command or how many times to perform the command. In other commands this specification is implicit and arguments are not allowed.

The ED-11 command arguments are described as follows:

1. \textit{n} stands for any number from 1 through 32767 (decimal) and may, except where noted, be preceded by a + or -. If no sign precedes \textit{n}, \textit{n} is assumed to be a positive number.

   where an argument is acceptable, its absence implies an argument of 1 (or -1 if a - is present).

   The role of \textit{n} varies according to the command with which it is associated.

2. \textit{0} refers to the beginning of the current line.

3. \textit{@} refers to a marked (designated) character location (see Section 4.2.3).

4. \textit{/} refers to the end of text in the Page Buffer.

The roles of all arguments are explained further with the corresponding commands which qualify them.

4.2.2 The Character Location Pointer (Dot)

Almost all ED-11 commands function with respect to a movable reference point, Dot. This character pointer is normally located between the most recent character operated upon and the next character and, at any given time, can be thought of as "where the Editor is" in your text. There are commands which move Dot anywhere in the text, thereby redefining the "current location" and allowing greater facility in the use of the other commands.
EDITING THE SOURCE PROGRAM

4.2.3 Mark

In addition to Dot, a secondary character pointer known as Mark also exists in ED-11. This less agile pointer is used with great effect to mark or "remember" a location by moving to Dot and conditionally remaining there while Dot moves on to some other place in the text. Thus, it is possible to think of Dot as "here" and Mark as "there". Positioning of Mark, which is referenced by means of the argument @, is discussed below in several commands.

4.2.4 Line-Oriented Command Properties

ED-11 recognizes a line as a unit by detecting a line terminator in the text. This means that ends of lines (line feed or form feed characters) are counted in line-oriented commands. This is important to know, particularly if Dot, which is a character location pointer, is not pointing at the first character of a line.

In such a case, an argument n does not affect the same number of lines (forward) as its negative (backward). For example, the argument -l applies to the character string beginning with the first character following the second previous end-of-line character and ending at Dot; argument +l applies to the character string beginning at Dot and ending at the first end-of-line character. If Dot is located, say, in the center of a line, notice that this affects 1-1/2 lines back or 1/2 line forward, respectively:

Example of List Commands -lL and +lL:

<table>
<thead>
<tr>
<th>Text</th>
<th>Command</th>
<th>Printout</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPB</td>
<td>ICHAR,#033</td>
<td>*-lL</td>
</tr>
<tr>
<td>BEQ</td>
<td>$ALT</td>
<td>BEQ $ALT</td>
</tr>
<tr>
<td>CMPB</td>
<td>ICHAR,#175</td>
<td>*+lL</td>
</tr>
<tr>
<td>BNE</td>
<td>PLACE</td>
<td>CMPB 1</td>
</tr>
<tr>
<td>Dot is here</td>
<td></td>
<td>CHAR,#175</td>
</tr>
</tbody>
</table>

4.2.5 The Page Buffer

The Page Buffer holds the text being edited. The unit of source data that is read into the Page Buffer from a paper tape, is the page. Normally a page is terminated, and therefore defined, by a form feed (CTRL/FORM) in the source text wherever a page is desired. (A form feed is an acceptable Text Mode character.) Overflow, no-tape, or reader-off conditions can also end a page of input (as described in Section 4.3.1.2). Since more than one page of text can be in the buffer at the same time, it should be noted that the entire contents of the Page Buffer are available for editing.
4.3 COMMANDS

4.3.1 Input and Output Commands

Three commands are available for reading in a page of text. The Read command (Section 4.3.1.2) is a specialized input command; the Next command (Section 4.3.1.4) reads in a page after punching out the previous page; and the Whole command (Section 4.3.3.2) reads in and punches out pages of text as part of a search for a specified character string.

Output commands either list text or punch it on paper tape. The List command causes specified lines of text to be printed at the terminal so that they may be examined. Paper tape commands (Next and Whole also perform input) provide for the output of specified pages, lines, form feeds (for changing the amount of data that constitutes a given page), and blank tape. Note that the process of outputting text does not cause Dot to move.

4.3.1.1 Open - The Open command (O) should be typed whenever a new tape is put in the reader. This is used when the text file being edited is on more than one paper tape.

Note also that if the reader is off at the time an input command is given, turning the reader on must be followed by the Open command.

4.3.1.2 Read - One way of getting a page of text into the Page Buffer so that it can be edited is the Read (R) command. The R command causes a page of text to be read from either the low-speed reader or high-speed reader (as specified in the starting dialogue, Section 4.4.2), and appended to the contents (if any) of the Page Buffer.

Text is read in until either:

1. A form feed character is encountered;

2. The page buffer is 128 characters from being filled, or a line feed is encountered after the buffer has become 500 characters from being filled;

3. The reader is turned off, or runs out of paper tape (see Open command, Section 4.3.1.1).

Following execution of an R command, Dot and Mark are located at the beginning of the Page Buffer.

A 4K system can accommodate about 4000 characters of text. Each additional 4K of memory provides space for about 8000 characters.

NOTE

An attempt to overflow the storage area causes the command (in this case, R) to stop executing. A ? is then printed, followed by an * on the next line indicating that a command may be typed. No data is lost.
4.3.1.3 List and Punch - Output commands List (L) and Punch (P) can be described together, as they differ only in that the device addressed by the former is the terminal, and the device addressed by the latter is the paper tape punch. Dot is not moved by these commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>nL</td>
<td>Lists body of the nth line ending with end-of-line</td>
</tr>
<tr>
<td>nP</td>
<td>Punches body of the nth line</td>
</tr>
<tr>
<td>-nL</td>
<td>Lists body of the first character following end-of-line before Dot</td>
</tr>
<tr>
<td>-nP</td>
<td>Punches body of the first character before Dot</td>
</tr>
<tr>
<td>@L</td>
<td>Lists body of the characters between Dot and the marked location</td>
</tr>
<tr>
<td>@P</td>
<td>Punches body of the characters before Marked location</td>
</tr>
<tr>
<td>/L</td>
<td>Lists body of the last character in the Page Buffer ending with a newline</td>
</tr>
<tr>
<td>/P</td>
<td>Punches body of the last character in the Page Buffer</td>
</tr>
</tbody>
</table>

In addition to the above List commands, there are three special List commands that accept no arguments. The current line is defined as the line containing Dot, i.e., from the line feed (or form feed) preceding Dot to the line feed (or form feed) following Dot.

- **V**
  - Lists the entire line containing Dot

- **<**
  - Same as -nL. If Dot is located at the beginning of a line, this simply lists the line preceding the current line

- **>**
  - Lists the line following the current line

**Examples:**

<table>
<thead>
<tr>
<th>TEXT</th>
<th>COMMANDS</th>
<th>PRINTOUT</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMPB ICHAR,#033</td>
<td>V</td>
<td>CMPB ICHAR,#175</td>
</tr>
<tr>
<td>BEQ $ALT</td>
<td>&lt;</td>
<td>BEQ $ALT</td>
</tr>
<tr>
<td>CMPB ICHAR,#175</td>
<td>&gt;</td>
<td>CMPB IPLACE</td>
</tr>
<tr>
<td>BNE IPLACE</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Dot is here. Dot remains here.

4.3.1.4 Next - Typing nN punches out the entire contents of the Page Buffer (followed by a trailer of blank tape if a form feed is the last character in the buffer), deletes the contents of the buffer, and reads the Next page into the buffer. It performs this sequence n times. If there are fewer than the n pages specified, the command is executed for the number of pages actually available, and a ? is printed out. Following execution of a Next, Dot and Mark are located at the beginning of the Page Buffer.
EDITING THE SOURCE PROGRAM

4.3.1.5 Form Feed and Trailer -

F  Punches out a Form feed character and four inches of blank tape

nT  Punches out four inches of Trailer (blank) tape n times

4.3.1.6 Procedure with Low-Speed Punch - If the low speed punch is the specified output device (see Section 4.4.2), the Editor pauses before executing any tape command just typed (Punch, Form feed, Trailer, Next, whole). The punch must be turned on at this time, after which typing the SPACE bar initiates the execution of the command. Following completion of the operation, the Editor pauses again to let you turn the punch off. When the punch has been turned off, typing the SPACE bar returns ED-11 to Command Mode.

4.3.2 Commands to Move Dot and Mark

4.3.2.1 Beginning and End -

B  Moves Dot to the Beginning of the Page Buffer

E  Moves Dot to the End of the Page Buffer (see also /J and /A below)

4.3.2.2 Jump and Advance -

nJ  Jumps Dot forward past n characters

-nJ  Moves Dot backward past n characters

nA  Advances Dot forward past n ends-of-lines to the beginning of the succeeding line

-nA  Moves Dot backwards across n ends-of-lines and positions Dot immediately after n+1 ends-of-lines, i.e., at the beginning of the -n line.

0J or 0A  Moves Dot to the beginning of the current line

@J or @A  Moves Dot to the Marked location

/J or /A  Moves Dot to the end of the Page Buffer (see also E above)

Notice that while n moves Dot n characters in the Jump command, its role becomes that of a line counter in the Advance command. However, because 0, @, and / are absolute, their use with these commands overrides line/character distinctions. That is, Jump and Advance perform identical functions if both have either 0, @ or / for an argument.
EDITING THE SOURCE PROGRAM

4.3.2.3 Mark - The M command marks ("remembers") the current position of Dot for later reference in a command using the argument @. Note that only one position at a time can be in a marked state. Mark is also affected by the execution of those commands which alter the contents of the Page Buffer:

C D H I K N R X

4.3.3 Search Commands

4.3.3.1 Get - The basic search command nG starts at Dot and gets the nth occurrence of the specified text in the Page Buffer. If no argument is present, it is assumed to be 1. When you type the command, followed by the RETURN key, ED-11 goes into Text Mode. The character string to be searched for must now be typed. (ED-11 will accept a search object of up to 42 characters.) Typing the LINE FEED key terminates Text Mode and initiates the search.

This command sets Dot to the position immediately following the found character string, and a 0L listing is performed by ED-11. If a carriage return, line feed, or form feed is specified as part of the search object, the automatic 0L displays only a portion of text -- the part defined as the last line. Where any of these characters is the last character of the search object, the 0L of course yields no printout at all.

If the search is unsuccessful, Dot is at the end of the Page Buffer and a ? is printed out. The Editor then returns to Command Mode.

Examples:

1. Text Command Printout

   MOV @RMAX,&R5
   ADD #6,(R5)+
   CLR $CK3
   TST R2
   BEQ CKCR

   Dot was here.

   Dot is now here.

2. CMPB ICHAR,#RUBOUT G
   BEQ SITE TE
   BR PUT BR+

   Dot

   Dot

4.3.3.2 Wh0le - A second search command, H, starts at Dot and looks through the Wh0le text file for the next occurrence of the character string you have specified in Text Mode. It combines a Get and a Next such that if the search is not successful in the Page Buffer, the contents of the buffer are punched on tape, the buffer contents are deleted, and a new page is read in, where the search is continued. This continues until the search object is found or until the complete source text has been searched. In either case, Mark is at the beginning of the Page Buffer.
If the search object is found, Dot is located immediately following it, and a 0L is performed by ED-11. As in the Get command, if the search is not successful Dot is at the end of the buffer and a ? appears on the teleprinter. Upon completion of the command, the Editor will be in Command Mode. No argument is allowed. Note that an H command specifying a nonexistent search object can be used to close out an edit, i.e., copy all remaining text from the input tape to the output tape.

4.3.4 Commands to Modify the Text

4.3.4.1 Insert - The Insert command (I) allows text to be inserted at Dot. After I is typed (followed by the typing of the RETURN key), the Editor goes into Text Mode to receive text to be inserted. Up to 80 characters per line are acceptable. Execution of the command occurs when the LINE FEED key (which does not Insert a line feed character unless it is the first key typed in Text Mode) is typed terminating Text Mode. At this point, Dot is located in the position immediately following the last inserted text character. If the Marked location was anywhere after the text to be Inserted, Dot becomes the new Marked location.

During an insert, it sometimes happens that the user accidentally types CTRL/P rather than SHIFT/P (for @), thus deleting the entire insert (see Section 4.4.1). To minimize the effect of such a mistake, the insert may be terminated every few lines and then continued with a new Insert command.

As with the Read command, an attempt to overflow the Page Buffer causes a ? to be printed out followed by an * on the next line indicating that a command may be typed. All or part of the last line typed may be lost. All previously typed lines are inserted. Examples:

<table>
<thead>
<tr>
<th>Text</th>
<th>Command</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>MOV #8.,EKOT</td>
<td>I/CN</td>
<td>MOV #8.,EKOCNT</td>
</tr>
<tr>
<td>Dot</td>
<td></td>
<td>Dot</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dot</td>
</tr>
<tr>
<td>Inserting a carriage return (and automatic line feed):</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CLR R1CLR R2</td>
<td>I/</td>
<td>CLR R1</td>
</tr>
<tr>
<td>Dot</td>
<td>+</td>
<td>CLR R2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dot</td>
</tr>
<tr>
<td>Inserting a single line feed:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>LOOK WHAT HAPPENS HERE</td>
<td>I/</td>
<td>LOOK WHAT</td>
</tr>
<tr>
<td>Dot</td>
<td>+</td>
<td>HAPPENS HERE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dot</td>
</tr>
</tbody>
</table>
EDITING THE SOURCE PROGRAM

4.3.4.2 Delete and Kill - These commands are closely related to each other; they both erase specified text from the Page Buffer. The Delete command (D) differs from the Kill command (K) only in that the former accepts an argument, n, that counts characters to be removed, while the latter accepts an argument, n, that counts lines to be removed. 0, @, and / are also allowed as arguments. After execution of these commands, Dot becomes the Marked location.

nD Deletes the following n characters
-nD Deletes the previous n characters
nK Kills the character string beginning at Dot and ending at the nth end-of-line
-nK Kills the character string beginning with the first character following the (n+1)th previous end-of-line and ending at Dot
0D or 0K Removes the current line up to Dot
@D or @K Removes the character string bounded by Dot and Mark
/D or /K Removes the character string beginning at Dot and ending with the last character in the Page Buffer

<table>
<thead>
<tr>
<th>Text</th>
<th>Command</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>;CHECK THE MOZXDE</td>
<td>-2D</td>
<td>;CHECK THE MODE</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Dot</td>
</tr>
<tr>
<td>;IS IT A TAB OR</td>
<td>2K</td>
<td>;IS IT A TAB</td>
</tr>
<tr>
<td>;IS IT A CR</td>
<td></td>
<td>Dot</td>
</tr>
</tbody>
</table>

4.3.4.3 Change and exchange - The Change (C) and eXchange (X) commands can be thought of as two-phase commands combining, respectively, an Insert followed by a Delete, and an Insert followed by a Kill. After the Change or eXchange command is typed, ED-11 goes into Text Mode to receive the text to be inserted. If n is used as the argument, it is then interpreted as in the Delete (character-oriented) or Kill (line-oriented), and accordingly removes the indicated text. 0, @, and / are also allowed as arguments.

nC Changes the following n characters
xxx Changes the previous n characters
-nC Changes the previous n characters
xxx eXchanges the character
xxxx eXchanges the character
xxxx string beginning at Dot and ending at the nth end-of-line
-nX eXchanges the character
xxx string beginning with the first character following the (n+1)th previous end-of-line and ending at Dot
EDITING THE SOURCE PROGRAM

QC or QX  Replaces the current line up to Dot
xxxx xxxx
xxxx xxxx

@C or @X  Replaces the character string bounded by Dot
xxx xxx  and the Marked location
xxx xxx

/C or /X  Replaces the character string beginning at Dot
xxx xxx  and ending with the last character in the Page
Buffer.

Again, the use of absolute arguments 0, and @, and / overrides the
line/character distinctions that n and -n produce in these commands.

If the Insert portion of a Change or eXchange is terminated because of
attempting to overflow the Page Buffer, data from the latest line may
have been lost, and text removal does not occur. Such buffer overflow
might be avoided by separately executing a Delete or Kill followed by
an Insert, rather than a Change or eXchange, which does an Insert
followed by a Delete or Kill. Examples:

<table>
<thead>
<tr>
<th>Text</th>
<th>Command</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>;A LINE FEED IS HERE</td>
<td>-9C /</td>
<td>; A TAB IS HERE</td>
</tr>
<tr>
<td>TAB:</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;THIS ;IS ON Dot PAPER ;THIS ;IS ON PAPER</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;FOUR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>;LINES</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dot</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

4.4 OPERATING PROCEDURES

4.4.1 Error Corrections

During the course of editing a page of the program, it may become
necessary to correct mistakes in the commands themselves. There are
four special commands which do this:

1. Typing the RUBOUT key removes the preceding typed character,
   if it is on the current line. Successive RUBOUTs remove
   preceding characters on the line (including the SPACE), one
   character for each RUBOUT typed.

2. The CTRL/U combination (holding down the CTRL key and typing
   U) removes all the characters in the current line.

3. CTRL/P cancels the current command in its entirety. This
   includes all the current command text just typed, if ED-11
   was in Text Mode. Do not use another CTRL/P before typing a
   line terminator as this will cause an ED-11 restart (see 4.
   below). If CTRL/P is typed while a found search object of a
   Get or whole is being printed out, the normal position of Dot
   (just after the specified search object) is not affected.

   CTRL/P should not be used while a punch operation is in
   progress as it is not possible to know exactly how much data
   will be output.
EDITING THE SOURCE PROGRAM

4. Two CTRL/P's not interrupted by a typed line terminator restart ED-11, initiating the dialogue described in Section 4.4.2.

After removing the incorrect command data, the user can directly type in the desired input.

4.4.2 Starting

The Editor is loaded by the Absolute Loader (see Chapter 6, Section 6.2.2) and starts automatically. Once the Editor has been loaded, the following sequence occurs:

<table>
<thead>
<tr>
<th>ED-11 Prints</th>
<th>User Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>*I</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td>H</td>
</tr>
<tr>
<td>*O</td>
<td>L</td>
</tr>
<tr>
<td></td>
<td>H</td>
</tr>
</tbody>
</table>

If all text is to be entered from the keyboard (i.e., via the Insert command), either L or H may be specified for Input.

If the output device is the high-speed punch (HSP), the Editor enters Command Mode to accept input. Otherwise, the sequence continues with:

LSP OFF ↳ (when low-speed Punch (LSP) is off)

Upon input of ↳ from the keyboard, the Editor enters Command Mode and is ready to accept input.

4.4.3 Restarting

To restart ED-11, type CTRL/P twice. This initiates the normal starting dialogue described in Section 4.4.2. If the Low-speed Reader (LSR) is in operation it must first be turned off. The text to be edited should be loaded (or reloaded) at this time.

4.4.4 Creating a Paper Tape

Input commands assume that text is to be read from a paper tape by the low-speed reader or high-speed reader. However, the five commands that go into Text Mode enable the user to input from the keyboard. The Insert command, in particular (Section 4.3.4.1) can be useful for entering large quantities of text not on paper tape. The Page Buffer can thus be filled from the keyboard, and a paper tape actually created by using a command to punch out the buffer contents.
EDITING THE SOURCE PROGRAM

4.4.5 Editing Example

The following example consists of three parts:

1. The marked up source program listing indicating the desired changes.

2. The ED-11 commands to implement those changes (with comments on the editing procedure).

NOTE

Typing the RETURN key terminates Command Mode in all cases. In commands which then go into Text Mode, typing the LINE FEED key (symbolized as + ) produces the terminator.

3. The edited text.

Part I  Original Source for Edit

$COMMON INPUT ROUTINE FOR USE BY NON FILE DEVICES

$INPUT: ADC ICHAR,(R5)+ ;UPDATE CKSUM
CLR -(LS) ;CLEAR DONE
MOV (R5)+,RMAX ;GET ADR MAX
MOV (R5)+,MODADR ;GET ADR MODE
;R5 NOW POINTS TO POINTER

$CKMODE:BITB @MODADB,$ASCII ;IS THIS ASCII
BNE CKBIN ;NO----TRY BINARY

$CKNUL: TSTB ICHAR ;ASCII---IS CHAR A NULL
BEQ CK ;YES--NO GO

$CKPAR: BITB @MODADR,$PARBIT ;LOOK AT MODE TO SEE IF
BNE PAROK ;SUPPOSED TO CHECK PARITY?

PAROK: CLR OCHAR
BIC $177200,ICCHAR ;STRIP PARITY
CMPB @10(RADD),$KBD ;IS THIS KBD INPUT
BNE OKO ;NO
TSTB EKDCNT ;YES----DONE KDO OF LAST?
BEQ OK ;YES
CLR ICHAR ;NO----DROP NEW CHAR

$JP2CK: JMP CK

;WHAT IS THE CHAR

$OK: CMPB ICHAR,CTRLC ;IS IT A ~C
BNE OKO ;NO
MOV #UPC,OCHAR ;YES--ECHO ~C
INC RDUN
MOV #$ABRTAD,20(R6) ;DIDDLE RETURN ADR
BB PLUS1

4-12
EDITING THE SOURCE PROGRAM

CKUPU: CMPB ICHAR, #HTAB ; IS IT A TAB
BNE CKCR ; NO
MOVB $BLNKS, OCHAR ; YES --- ECHO BLANKS
MOV TABCNT, EKOCNT ; SET UP COUNTER
BB PUT ;

CKCR: CMPB ICHAR, #CR ; IS IT A CR?
BNE $CK3 ; NO
MOV $CRLF, OCHAR ; YES --- ECHO CRLF
INC RDUN ;

$CK3: CMPB ICHAR, #033
BEQ $ALT
CMPB ICHAR, #175
BEQ $ALT
CMPB ICHAR, #176
BNE CK

$ALT: MOV $BOL, OCHAR
MOV #175, ICHAR
INC RDUN
BB PUT

ALT

4-13
EDITING THE SOURCE PROGRAM

```
CKLF:  CMPB  ICHAR, #LF
        BNE  CKFF
        INC  RDUN
        BB   PUT

CKFF:  MOV  ICHAR, OCHAR
        CMPB  ICHAR, #FF
        BNE  PUT
        MOV  $B, ,EKOCNT
        MOV  $LFLF, OCHAR
        BB   PUT
```

Part II: Editing Session

Assume that ED-11 has been started, is in Command Mode, and the tape is in the reader. Underlined matter indicates ED-11 output.

* R ; Reads in a page of text
* H ; Searches entire program for 2CK: -
  2CK;↑ ; when found ED-11 performs a 0L
  $JP2CK:

  * G ; Searches current page for next CK -
  CK ; when found ED-11 performs a 0L
  $JP2CK  JMP  CK

* I ; Inserts DUN following CK
  DUN;↑

  * G ; Searches for next CKUPP -
  CKUPP;↑ ; when found ED-11 performs a 0L

  * -5C ; OK0 replaces last 5 characters (CKUPP)
  OK0

  * 6A ; Dot is moved 6 lines ahead (including a blank line)

  * 9K ; 9 lines are killed starting with CKUPP:

  * L ; Next line is listed - Dot is not moved
  ; THIS IS NOT KBD INPUT

  * I ; Blank line is inserted

  ↑

  * A ; Dot is moved 1 line ahead to point to character 0 of OK0:

  * 4X ; Following comments replace the next 4 lines
  ; FORMATTED AND UNFORMATTED
  ; ASCII ARE HANDLED THE SAME+

  * G ; Searches for next CKINP: -
  CKINP;↑

```
EDITING THE SOURCE PROGRAM

*0J ;Dot is moved to the beginning of the current line.
*/K ;The rest of the page is killed (3 lines)
*N ;Current page is punched out on paper tape - a new page is read in
*L TST 2(R5) ;BC=0?

;The next line is listed - Dot is not moved
*15K ;15 lines are killed starting with TST
*2L ;1 blank line and 1 line of text are listed - Dot is not moved

CKTAB: CMPB ICHAR,#HTAB ;IS IT A TAB

*2G $CK3+ ;Searches for 2nd occurrence of $CK3 - 0L printout verifies it is found
$CK3

*-C ;ALT replaces preceding character
ALT+

*V $CKALT: CMPB ICHAR,#033 ;Lists entire current line to verify the above-C result

*G $033+ ;Searches for the 033 to position Dot ;for next command -- 0L occurs
$CKALT: CMPB ICHAR,#033

*I ;The following text is inserted in the comment field ;IS CHAR AN ALTMODE?

*G CKLF+ ;Searches for next CKLF -- 0L occurs

BNE CKLF

*-2C ;EX replaces the preceding two characters ;(LF)
EX

*2J ;Jumps Dot past the carriage return and ;line feed characters

*K ;Kills next line (starting with $ALT:)

*I $ALT:+ ;Inserts $ALT: at beginning of the following line

*A ;Advances Dot past 1 line feed to the ;beginning of the next line

*M ;Marks the position of Dot

*B ;Moves Dot to the beginning of the current page

4-15
EDITING THE SOURCE PROGRAM

*OP
; Punches out the lines from Dot to the
; position just marked - Dot not moved

*A
; Moves Dot from the beginning of the
; page to the marked position

*K
; Kills the next 2 lines

PART III Edited Source

; COMMON INPUT ROUTINE FOR USE BY NON FILE DEVICES

INPUT: ADD ICHAR,(R5)+ ; UPDATE CKSUM
CLR -(LS) ; CLEAR DONE
MOV (R5)+,RMAX ; GET ADDR MAX
MOV (R5)+,MODADR ; GET ADDR MODE
; R5 NOW POINTS TO POINTER

CHKMODE: BITB @MODADR,#ASCII ; IS THIS ASCII
BNE CKBIN ; NO----TRY BINARY

CKNUL: TSTB ICHAR ; ASCII---IS CHAR A NULL
BEQ CK ; YES---NO GO

CKPAR: BITB @MODADR,#PARBIT ; LOOK AT MODE TO SEE IF
BNE PAROK ; SUPPOSED TO CHECK PARITY?
MOV ICHAR,OCR ; YES---CK IT
JSR R7,PARGEN
SUB ICHAR,OCR ;
BEQ PAROK ; OK?
BIS $PARERR,@MODADR ; NO----SET ERR BIT
PAROK: CLR OCR
BIC $177200,ICHAR ; STRIP PARITY
CMPB @10(RADD),#KBD ; IS THIS KBD INPUT
BNE OK0 ; NO
TSTB EKOCNT ; YES---DONE EKO OF LAST?
BEQ OK ; YES
CLR ICHAR ; NO----DROP NEW CHAR

JP2CK: JMP CKDUN

OK: CMPB ICHAR,#CTRLC ; WHAT IS THE CHAR
BNE OK0 ; IS IT A 'C
MOV #UPC,OCR ; YES---ECHO 'C
INC RDUN
MOV #ABRTAD,20(R6) ; DIDDLE RETURN ADR

BB PLUS1
BEQ CK ; THIS IS NOT KBD INPUT
BEQ CK ; FORMATTED AND UNFORMATTED
BEQ CK ; ASCII ARE HANDLED THE SAME

CMPP ICHAR,#RUBOUT ; IS THIS A RUBOUT
BEQ CK ; YES----IGNORE IT
BB PUT ; NO------

CKTAB: CMPB ICHAR,#HTAB ; IS IT A TAB
BNE CKCR ; NO
MOV #BLNKS,OCR ; YES----ECHO BLANKS
MOVP TABCNT,EKOCNT ; SET UP COUNTER
BB PUT ;
4.5 SOFTWARE ERROR HALTS

ED-11 loads all unused trap vectors with the code

```
.WORD .+2,HALT
```

so that if the trap does occur, the processor halts in the second word of the vector. The address of the halt, displayed in the console address register, therefore indicates the cause of the halt. In addition to the halts which may occur in the vectors, the standard IOX error halt at location 40 may occur (see Chapter 7).

<table>
<thead>
<tr>
<th>Address of HALT</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Reserved instruction executed</td>
</tr>
<tr>
<td>16</td>
<td>Trace trap occurred</td>
</tr>
<tr>
<td>26</td>
<td>Power fail trap</td>
</tr>
<tr>
<td>32</td>
<td>EMT executed</td>
</tr>
<tr>
<td>36</td>
<td>TRAP executed</td>
</tr>
<tr>
<td>40</td>
<td>IOX detected error</td>
</tr>
</tbody>
</table>
CHAPTER 5
DEBUGGING OBJECT PROGRAMS ON-LINE

5.1 INTRODUCTION

ODT-11 (On-line Debugging Technique for the PDP-11) is a system program which aids in debugging assembled object programs. From the Teletype keyboard you interact with ODT and the object program to:

- print the contents of any location for examination or alteration
- run all or part of an object program using the breakpoint feature
- search the object program for specific bit patterns
- search the object program for words which reference a specific word
- calculate offsets for relative addresses

During a debugging session you should have at the terminal the assembly listing of the program to be debugged. Minor corrections to the program may be made on-line during the debugging session. The program may then be run under control of ODT to verify any change made. Major corrections, however, such as a missing subroutine, should be noted on the assembly listing and incorporated in a subsequent updated program assembly.

A binary tape of the debugged program can be obtained by use of the DUMPAB program (see Chapter 6, section 6.3).

5.1.1 ODT-11 and ODT-11X

There are two versions of ODT included in the PDP-11 Paper Tape Software System: a standard version, ODT-11, and an extended version, ODT-11X.¹ Both versions are independent, self-contained programs. ODT-11X has all the features of ODT-11, plus some additional features. Each version is supplied on two separate paper tapes: a source tape and an absolute binary tape. The purpose of the tapes, and loading and starting procedures are explained in a later section of this chapter.

ODT-11 is completely described in section 5.2, and the additional features of ODT-11X are covered in section 5.3. In all sections of this chapter, except where specifically stated, reference to ODT

¹ Only ODT-11X is available for the LSI-11 or the PDP-11/03.
applies to both versions. Concluding sections discuss ODT's internal operations -- how it effects breakpoints, how it uses the "trace trap" and the T-bit, and other useful data.

The following discussion assumes that the reader is familiar with the PDP-11 introduction formats and the PAL-11A Assembly Language as described in Chapter 3.

5.1.2 ODT's Command Syntax

ODT's commands are composed of the following characters and symbols. They are often used in combination with the address upon which the operation is to occur, and are offered here for familiarization prior to their thorough coverage which follows. Unless indicated otherwise, n below represents an octal address.

n/ open the word at location n
/
reopen last opened location
n\ (SHIFT/L) open the byte at location n (ODT-11X only)
\ reopen the last opened byte (ODT-11X only)
⁺ (LINE FEED key) open next sequential location
⁺⁺ open previous location

RETURN close open location and accept the next command
⁻⁻ take contents of opened location, index by contents of PC, and open that location
etical take contents of opened location as absolute address and open that location (ODT-11X only)
> take contents of opened location as relative branch instruction and open referenced location (ODT-11X only)
< return to sequence prior to last @, >, or — command and open succeeding location (ODT-11X only)

$n/$ open general register n (0-7)
; separates commands from command arguments (used with alphabetic commands below)
;B remove Breakpoint(s) (see description of each ODT version for particulars)

n;B set Breakpoint at location n

¹The circumflex appears on some keyboards and printers in place of the up-arrow.
²The underline appears on some keyboards and printers in place of the back-arrow.
DEVELOPING OBJECT PROGRAMS ON-LINE

n;rB set Breakpoint r at location n (ODT-11X only)
;rB remove r(th) Breakpoint (ODT-11X only)
;n;E search for instructions that reference Effective address n
;n;W search for Words with bit patterns which match n
;nS enable Single-instruction mode (n can have any value and is not significant); disable breakpoints
;S disable Single-instruction mode
;n;G Go to location n and start program run
;p Proceed with program execution from breakpoint; stop when next breakpoint is encountered or at end of program

In Single-instruction mode only (ODT-11X), Proceed to execute next instruction only

n;P Proceed with program execution from breakpoint; stop after encountering the breakpoint n times.

In single-instruction mode only (ODT-11X), Proceed to execute next n instructions.

n/(word);n;O calculate Offset from location n to location m
$B/ open Breakpoint status word (ODT-11)
open BREAKPOINT 0 STATUS WORD (ODT-11X)
$M/ open search Mask
$S/ open location containing user program's Status register
$P/ open location containing ODT's Priority level

With ODT-11, location references must be to even numbered 16-bit words. With ODT-11X, location references may be to 16-bit words or 8-bit bytes.

The semicolon in the above commands is ignored by ODT-11, but is used for the sake of consistency, since similar commands to ODT-11X require it.

5.2 COMMANDS AND FUNCTIONS

When ODT is started as explained in section 5.6, it indicates its readiness to accept commands by printing an asterisk (*) on the left margin of the terminal paper. In response to the asterisk, you can issue most commands; for example, you can examine and, if desired, change a word, run the object program in its entirety or in segments, or even search core for certain words or references to certain words. The discussion below first explains some elementary features, and then covers the more sophisticated features.

All commands to ODT are issued using the characters and symbols shown above in Section 5.1.2.
5.2.1 Opening, Changing, and Closing Locations

An open location is one whose contents ODT has printed for examination, and whose contents are available for change. A closed location is one whose contents are no longer available for change. Any even-numbered location may be opened using ODT-ll.

The contents of an open location can be changed by typing the new contents followed by a single character command, which requires no argument (i.e. + RETURN + @ > <). Any command typed to open a location when another location is already open causes the currently open location to be closed.

5.2.1.1 The Slash (/) - One way to open a location is to type its address followed by a slash:

*1000/012746

Location 1000 is open for examination and is available for change. Note that in all examples ODT's printout is underlined; your typed input is not.

Should you not wish to change the contents of an open location, merely type the RETURN key and the location will be closed; ODT prints another asterisk and waits for another command. However, should you wish to change the word, simply type the new contents before giving a command to close the location.

*1000/012746 012345

In the example above, location 1000 now contains 012345 and is closed since the RETURN key was typed after entering the new contents, as indicated by ODT's second asterisk.

Used alone, the slash reopens the last location opened:

*1000/012345 2340

*/002340

As shown in the example above, an open location can be closed by typing the RETURN key. In this case, ODT changed the contents of location 1000 to 002340 and then closed the location before printing the *. A single slash then directed ODT to reopen the last location opened. This allowed us to verify that the word 002340 was correctly stored in location 1000. (ODT supplies the leading zeroes if not given.)

Note again that opening a location while another is currently open automatically closes the currently open location before opening the new location.

5.2.1.2 The LINE FEED Key - If the LINE FEED key is typed when a location is open, ODT closes the open location and opens the next sequential location:

*1000/002340 +
001002/012740

( + denotes typing the LINE FEED key)
DEBUGGING OBJECT PROGRAMS ON-LINE

In this example, the LINE FEED key instructed ODT to print the address of the next location along with its contents and to wait for further instructions. After the above operation, location 1000 is closed and 1002 is open. The open location may be modified by typing the new contents.

5.2.1.3 The Up-Arrow(↑) - The up-arrow (or circumflex) symbol is effected by typing the SHOPT and N key combination. If the up-arrow is typed when a location is open, ODT closes the open location and opens the previous location (as shown by continuing from the example above):

001002/012740  ↑ (↑ is printed by typing SHOPT and N)
001000/002340

Now location 1002 is closed and 1000 is open. The open location may be modified by typing the new contents.

5.2.1.4 The Back-Arrow(←) - The back-arrow (or underline) symbol is effected by typing the SHIFT and 0 key combination. If the back-arrow is typed to an open location, ODT interprets the contents of the currently open location as an address indexed by the Program Counter (PC) and opens the location so addressed:

*1006/000006  ← (← is printed by typing SHIFT and 0)
001016/100405

Notice in this example that the open location (1006) was indexed by the PC as if it were the operand of an instruction with address mode 67 as explained in Chapter 3.

A modification to the opened location can be made before a ↑, ↑, or ← is typed. Also, the new contents of the location will be used for address calculations using the ← command. Example:

*100/000222  4↑
000102/000111  6↑
000100/000004  100←
000202/(contents)  (modify to 4 and open next location)
(modify to 6 and open previous location)
(change to 100 and open location indexed by PC)

5.2.1.5 Accessing General Registers 0-7 - The program's general registers 0-7 can be opened using the following command format:

*$n/

where n is the integer representing the desired register (in the range 0 through 7). When opened, these registers can be examined or changed by typing in new data as with any addressable location. For example:

*$0/000033

(R0 was examined and closed)

and

*$4/000474 464

(R4 was opened, changed, and closed)
DEBUGGING OBJECT PROGRAMS ON-LINE

The example above can be verified by typing a slash in response to ODT's asterisk:

*/000464

The +, +, or @ commands may be used when a register is open (the @ is an ODT-11X command).

5.2.1.6 Accessing Internal Registers - The program's Status Register contains the condition codes of the most recent operational results and the interrupt priority level of the object program. It is opened using the following command:

*$$/000311

where $$ represents the address of the Status Register. In response to $$/ in the example above, ODT printed the 16-bit (of which only the low-order 8 bits are meaningful): Bits 0-3 indicate whether a carry, overflow, zero, or negative (in that order) has resulted, and bits 5-7 indicate the interrupt priority level (in the range 0-7) of the object program.

The $ is used to open certain other internal locations:

$B
 internal breakpoint status word (see section 5.2.2.2)

$M
 mask location for specifying which bits are to be examined during a bit pattern search (see section 5.2.4)

$P
 location defining the operating priority of ODT (see section 5.2.6)

$S
 location containing the condition codes (bits 0-3) and interrupt priority level (bits 5-7)

5.2.2 Breakpoints

The breakpoint feature facilitates monitoring the progress of program execution. A breakpoint may be set at any instruction which is not referenced by the program for data. When a breakpoint is set, ODT replaces the contents of the breakpoint location with a trap instruction. Thus, when the program is executed and the breakpoint is encountered, program execution is suspended, the original contents of the breakpoint location are restored, and ODT regains control.

5.2.2.1 Setting the Breakpoint(n;B) - ODT-11 provides only one breakpoint; ODT-11X provides eight. Breakpoint(s) may be changed at any time. A breakpoint is set by typing the address of the desired location of the breakpoint followed by ;B. For example:

*1020;B
#

sets a breakpoint at location 1020. The breakpoint above is changed to location 1120 as shown below.

*1020;B
#1120;B
#
DEBUGGING OBJECT PROGRAMS ON-LINE

Breakpoints should not be set at locations referenced by the program for data, nor at an IOT, EMT, or TRAP instruction. This restriction is explained in section 5.5.2.

The breakpoint is removed by typing ;B without an argument, as shown below.

\*1120;B
\*;B
\* (sets breakpoint at location 1120)
(removes breakpoint)

5.2.2.2 Locating the Breakpoint($B) – The command $B/ causes ODT-11 to print the address of the breakpoint (see also section 5.3.3 on $B in ODT-11X):

\*$B/001120

The breakpoint was set at location 1120. $B represents the address containing ODT-11's breakpoint location. Typing the RETURN key in the example above leaves the breakpoint at location 1120 and returns control to ODT-11. The breakpoint could be changed to a different location:

\*+$B/001120 1114
\*+$B/001114
\*

The breakpoint was found in location 1120, changed to location 1114, and the change was verified.

If no breakpoint is set, $B contains an address internal to ODT-11.

5.2.3 Running the Program(n;G and n;P)

Program execution is under control of ODT. There are two commands for running the program: n;G and n;P. The n;G command is used to start execution (GO) and n;P to continue (Proceed) execution after halting at a breakpoint. For example:

\*1000;G

starts execution at location 1000. The program runs until it encounters a breakpoint or until program completion. If the program enters an infinite loop, it must be restarted or reentered as explained in section 5.6.2.

When a breakpoint is encountered, execution stops and ODT-11 prints B; followed by the address of the breakpoint. Desired locations can then be examined for expected data. For example:

\*1010;B
\*1000;G
B;001010
\* (breakpoint is set at location 1010)
(execution started at location 1000)
(execution stopped at location 1010)

To continue program execution from the breakpoint, type ;P in response to ODT-11's last *.

When a breakpoint is set in a loop, it may be desirable to allow the program to execute a certain number of times through the loop before
recognizing the breakpoint. This may be done by typing the \texttt{n;P} command and specifying the number of times the breakpoint is to be encountered before program execution is suspended (on the \texttt{n(th)} encounter). (See section 5.3.3 for ODT-1lx interpretation of this command when more than one breakpoint is set in a loop.)

Example:

\begin{verbatim}
B:001010  (execution halted at breakpoint)
*1250;b  (set breakpoint at location 1250)
*4;P     (continue execution. loop through
          breakpoint 3 times and halt on the
          4(th) occurrence of the breakpoint)
\end{verbatim}

The breakpoint repeat count can be inspected by typing \texttt{$B/} followed by \texttt{LINE FEED}. The repeat count is then printed. This also provides an alternative way of specifying the count. Since the location is open, its contents can be modified in the usual manner by typing new contents followed by the \texttt{RETURN} key.

\begin{verbatim}
*B/001114+  (address of breakpoint is 1114)
nnnnn/000003  6 (repeat count was 3, changed to 6)
\end{verbatim}

Breakpoints are inserted when performing an \texttt{n;G} or \texttt{n;P} command. Upon execution of the \texttt{n;G} or \texttt{n;P} command, the general registers 0-6 are set to the values in the locations specified as \texttt{S0-S6} and the processor status register is set to the value in the location specified as \texttt{S}.

### 5.2.4 Searches

With ODT you can search all or any specified portion of core memory for any specific bit pattern or for references to a specific location.

The location represented by \texttt{$M} is used to specify the mask of the search. The next two sequential locations contain the lower and upper limits of the search. Bits set to 1 in the mask are examined during the search; other bits are ignored. For example,

\begin{verbatim}
*$M/000000 177400+  (\dagger \text{ denotes typing LINE FEED})
nnnnn/000000 1000+  (starting address of search)
nnnnn/000000 1040 (last address in search)
\end{verbatim}

where \texttt{nnnnnn} represents some location in ODT. This location varies and is meaningful only for reference purposes. Note that in the first line above, the slash was used to open \texttt{$M} which now contains 177400, and that the LINE FEEDs opened the next two sequential locations which now contain the lower and upper limits of the search.

#### 5.2.4.1 Word Search(n;W)

Before initiating a word search, the mask and search limits must be specified as explained above. Then the search object and the initiating command are given using the \texttt{n;W} command where \texttt{n} is the search object. When a match is found, the address of the unmasked matching word is printed. For example:

\begin{verbatim}
*$M/000000 177400+  (test high order eight bits)
nnnnn/000000 1000+  
nnnnn/000000 1040
*400;W
001010/000770
001034/000404
\end{verbatim}

\(5-8\)
In the search process, the word currently being examined and the search object are exclusive ORed (XORed), and the result is ANDed to the mask. If this result is zero, a match has been found, and is reported at the terminal. Note that if the mask is zero, all locations within the limits are printed.

5.2.4.2 Effective Address Search(\(n;E\)) - ODT enables you to search for words which address a specified location. After specifying the search limits (section 5.2.4), type \(n;E\) (where \(n\) is the effective address) to initiate the search.

Words which are either an absolute address (argument \(n\) itself), a relative address offset, or a relative branch to the effective address are printed after their addresses. For example:

\[
\begin{align*}
&*\text{SM/177400} + \\
n\text{nnnnn/001000} & 1010 + \\
n\text{nnnnn/001040} & 1060 \\
&*\text{1034;E} \\
&001016/001006 & (\text{relative branch}) \\
&001054/002767 & (\text{relative branch}) \\
&*\text{1020;E} \\
&001022/177774 & (\text{relative address offset}) \\
&001030/001020 & (\text{absolute address}) \\
&*
\end{align*}
\]

Particular attention should be given to the reported references to the effective address because a word may have the specified bit pattern of an effective address without actually being so used. ODT will report these as well.

5.2.5 Calculating Offsets(\(n;O\))

Relative addressing and branching use an offset - the number of words or bytes forward or backward from the current location of the effective address. During the debugging session it may be necessary to change a relative address or branch reference by replacing one instruction offset with another. ODT calculates the offsets in response to the \(n;O\) command.

The command \(n;O\) causes ODT to print the 16-bit and 8-bit offsets from the currently open location to address \(n\). In ODT-11, the 8-bit offset is printed as a 16-bit word. For example:

\[
\begin{align*}
&*\text{346/000034} 414;O \ 000044 \ 000022 \ 22 \\
&*/000022 \\
&*\text{20/000046} 200;O \ 000156 \ 000067 \ 67 \\
&*/20/000067
\end{align*}
\]

In the first example, location 346 is opened and the offsets from that location to location 414 are calculated and printed. The contents of location 346 are then changed to 22 and verified on the next line. The 16-bit offset is printed followed by the 8-bit offset. In the example above, 000156 is the 16-bit offset and 000067 is the 8-bit offset.

The 8-bit offset is printed only if the 16-bit offset is even, as in the case above. With ODT-11 only, the user must determine whether the 8-bit offset is out of the range 177600 to 000177 (-128 decimal to 127 decimal). The offset of a relative branch is calculated and modified as follows:
Note that the modified low-order byte 377 must be combined with the unmodified high-order byte. Location 1034 was still open after the calculation, thus typing 103777 changed its contents; the location was then closed.

5.2.6 ODT'S Priority Level($P$)

$P$ represents a location in ODT that contains the priority level at which ODT operates. If $P$ contains the value 377, ODT operates at the priority level of the processor at the time ODT is entered. Otherwise $P$ may contain a value between 0 and 7 corresponding to the fixed priority at which ODT operates.

To set ODT to the desired priority level, open $P$. ODT prints the present contents, which may then be changed:

```
*SP/000006 377
```

If $P$ is not specified, its value is seven.

Breakpoints may be set in routines at different priority levels. For example, a program running at a low priority level may use a device service routine operating at a higher priority level. If a breakpoint occurs from a low priority routine, if ODT operates at a low priority, and if an interrupt does occur from a high priority routine, then the breakpoints in the high priority routine will not be executed since they have been removed.

5.3 ODT-11X

ODT-11X has all the commands and features of ODT-11 as explained in section 5.2, plus the following.

5.3.1 Opening, Changing and Closing Locations

In addition to operating on words, ODT-11X operates on bytes.

One way to open a byte is to type the address of the byte followed by a backslash:

```
*1001/025
```

A backslash typed alone reopen the last open byte. If a word was previously open, the backslash reopens its even byte.

```
*1002/000004\004
```

The LINE FEED and up-arrow (or circumflex) keys operate on bytes if a byte is open when the command is given. For example:

```
*1001\025 +
001002\004 +
001001\025
```
5.3.1.1 Open the Addressed Location(\(\@\)) - The symbol \(\@\) optionally modifies, closes an open word, and uses its contents as the address of the location to open next.

\[
\begin{align*}
*1006/001024 & \quad \@ & \quad (\text{open location 1024 next}) \\
001024/000500 & \\
*1006/001024 & \quad 2100 & \quad (\text{modify to 2100 and open location 2100}) \\
002100/177774 & 
\end{align*}
\]

5.3.1.2 Relative Branch Offset(\(>\)) - The right angle bracket, \(>\), optionally modifies, closes an open word, and uses its even byte as a relative branch offset to the next word opened.

\[
\begin{align*}
*1032/000407 & \quad 301 & \quad (\text{modify to 301 and interpret as a relative branch}) \\
000636/000010 & 
\end{align*}
\]

Note that 301 is a negative offset (-77). The offset is doubled before it is added to the PC; therefore, 1034 + -176 = 636.

5.3.1.3 Return to Previous Sequence(\(<\)) - The left angle bracket, \(<\), optionally modifies, closes an open location, and opens the next location of the previous sequence interrupted by a +, \(\@\), or \(>\) command. Note that \(<\), \(\@\), \(>\) cause a sequence change to the word opened. If a sequence change has not occurred, \(<\) simply opens the next location as a LINE FEED does. The command operates on both words and bytes.

\[
\begin{align*}
*1032/000407 & \quad 301 & \quad (>\text{causes a sequence change}) \\
000636/000010 & \quad (<\text{causes a return to original sequence}) \\
001034/001040 & \quad @ & \quad (@\text{causes a sequence change}) \\
001040/000405/005 & \quad (<\text{now operates on byte}) \\
001035\ 002 & \quad (<\text{acts like +}) \\
001036\ 004 & 
\end{align*}
\]

5.3.2 Calculating Offsets(n;O)

The command n;O causes ODT to print the 16-bit and 8-bit offsets from the currently open location to address n. The following examples, repeated from the ODT-11 section describing this command (see section 5.2.5), show a difference only in printout format:

\[
\begin{align*}
*346/000034 & \quad 414;0 & \quad 000044\ 022 & \quad 22 \\
& /000022 & \\
*1034/103421 & \quad 1034;0 & \quad 177776\ 377\ \021 & \quad 377 \\
& /103777 & 
\end{align*}
\]

Note that the modified low-order byte 377 must be combined with the unmodified high-order byte.
5.3.3 Breakpoints

With ODT-11X you can set up to eight breakpoints concurrently, numbered 0 through 7. The n;B command used in ODT-11 to set the breakpoint at address n sets the next available breakpoint in ODT-11X. Specific breakpoints may be set or changed by the n;mB command where m is the number of the breakpoint. For example:

*1020;B  (sets breakpoint 0)
*1030;B  (sets breakpoint 1)
*1040;B  (sets breakpoint 2)
*1032;1B  (resets breakpoint 1)

The ;B command used in ODT-11 to remove the only breakpoint removes all breakpoints in ODT-11X. To remove only one of the breakpoints, use the ;nB command, where n is the number of the breakpoint. For example:

*1.2B  (removes the second breakpoint)

The $B/ command opens the location containing the address of breakpoint 0. The next seven locations contain the addresses of the other breakpoints in order, and thus can be opened using the LINE FEED key. (The next location is for single-instruction mode, explained in the next section.) Example:

*$B/001020 +
nnnnn/001032 +
nnnnn/(address internal to ODT)

In this example, breakpoint 2 is not set. The contents are an address internal to ODT. After the table of breakpoints is the table of Proceed command repeat counts for each breakpoint and for the single-instruction mode (see Section 5.3.4).

nnnnnn/001036 +  (address of breakpoint 7)
nnnnn/nnnnnn +  (single-instruction address)
nnnnnnn/000000 +  (count for breakpoint 0)
nnnnnnn/000000 +  (count for breakpoint 1)

It should be noted that a repeat count in a Proceed command refers only to the most recent breakpoint. Execution of other breakpoints encountered is determined by their own repeat counts.

5.3.4 Single-Instruction Mode

With this mode you can specify the number of instructions you wish executed before suspension of the program run. The Proceed command, instead of specifying a repeat count for a breakpoint encounter, specifies the number of succeeding instructions to be executed. Note that breakpoints are disabled when single-instruction mode is operative. Commands for single-instruction mode follow:

;nS  Enables single-instruction mode (n can have any value and serves only to distinguish this form from the form ;S); breakpoints are disabled.
n;P Proceeds with program run for next n instructions before reentering ODT (if n is missing, it is assumed to be 1). (Trap instructions and associated handlers can affect the Proceed repeat count. See section 5.5.2.)

;S Disables single-instruction mode

When the repeat count for single-instruction mode is exhausted and the program suspends execution, ODT prints:

```
B8;n
```

where n is the address of the next instruction to be executed. The $B breakpoint table contains this address following that of breakpoint 7. However, unlike the table entries for breakpoints 0-7, the B8 entry is not affected by direct modification.

Similarly, the repeat count for single-instruction mode follows the repeat count for breakpoint 7. This table entry, however, may be directly modified, and thus is an alternative way of setting the single-instruction mode repeat count. In such a case, ;P implies the argument set in the $B repeat count table rather than the argument 1.

5.4 ERROR DETECTION

ODT-11 and ODT-11X inform you of two types of errors: illegal or unrecognizable command and bad breakpoint entry.

Neither ODT-11 nor ODT-11X checks for the legality of an address when commanded to open a location for examination or modification.

Thus, the command

```
177774/
```

references nonexistent memory, and causes a trap through the vector at location 4. If this vector has not been properly initialized (by IOX, or the user program if IOX is not used), unpredictable results occur.

Similarly, a command such as

```
$20/
```

which references an address eight times the value represented by $2, may cause an illegal (nonexistent) memory reference.

Typing other than a legal command causes ODT to ignore the command, print

```
?
```

and wait for another command. Therefore, to cause ODT to ignore a command just typed, type an illegal character (such as 9 or RUBOUT) and the command will be treated as an error, i.e., ignored.

ODT suspends program execution whenever it encounters a breakpoint, i.e., a trap to its breakpoint routine. If the breakpoint routine is entered and no known breakpoint caused the entry, ODT prints:

```
BE001542
```

5-13
DEBUGGING OBJECT PROGRAMS ON-LINE

and waits for another command. In the example above, BE001542 denotes Bad Entry from location 001542. A bad entry may be caused by an illegal trace trap instruction, setting the T-bit in the status register, or by a jump to the middle of ODT.

5.5 PROGRAMMING CONSIDERATIONS

Information in this section is not necessary for the efficient use of ODT. However, its content does provide a better understanding of how ODT performs some of its functions.

5.5.1 Functional Organization

The internal organization of ODT is almost totally modularized into independent subroutines. The internal structure consists of three major functions: command decoding, command execution, and various utility routines.

The command decoder interprets the individual commands, checks for command errors, saves input parameters for use in command execution, and send control to the appropriate command execution routine.

The command execution routines take parameters saved by the command decoder and use the utility routines to execute the specified command. Command execution routines exit either to the object program or back to the command decoder.

The utility routines are common routines such as SAVE-RESTORE and I/O. They are used by both the command decoder and the command executors.

Communication and data flow are illustrated in Figure 5-1.

5.5.2 Breakpoints

The function of a breakpoint is to pass control to ODT whenever the user program tries to execute the instruction at the selected address. Upon encountering a breakpoint, the user can utilize all of the ODT commands to examine and modify his program.

When a breakpoint is executed, ODT removes the breakpoint instruction(s) from the user's code so that the locations may be examined and/or altered. ODT then types a message to the user, in the form Bn(Bm;n for ODT-11X), where n is the breakpoint address (and m is the breakpoint number). The breakpoints are automatically restored when execution is resumed.

A major restriction in the use of breakpoints is that the word
Figure 5-1 Communication and Data Flow
where a breakpoint has been set must not be referenced by the program in any way since ODT has altered the word. Also, no breakpoint should be set at the location of any instruction that clears the T-bit. For example:

```
MOV $240,177776 ;SET PRIORITY TO LEVEL 5.
```

A breakpoint occurs when a trace trap instruction (placed in the user program by ODT) is executed. When a breakpoint occurs, ODT takes the following steps:

1. Set processor priority to seven (automatically set by trap instruction).
2. Save registers and set up stack.
3. If internal T-bit trap flag is set, go to step 13.
4. Remove breakpoint(s).
5. Reset processor priority to ODT's priority or user's priority.
6. Make sure a breakpoint or Single-instruction mode caused the interrupt.
7. If the breakpoint did not cause the interrupt, go to step 15.
8. Decrement repeat count.
9. Go to step 18 if non-zero, otherwise reset count to one.
10. Save Teletype status.
11. Type message to user about the breakpoint or Single-instruction mode interrupt.
12. Go to command decoder.
14. Jump to the "GO" processor.
15. Save Teletype status.
16. Type "BE" (Bad Entry) followed by the address.
17. Clear the T-bit, if set, in the user status and proceed to the command decoder.
18. Go to the "Proceed", bypassing the TTY restore routine.

Note that steps 1-5 inclusive take approximately 100 microseconds during which time interrupts are not permitted to occur (ODT is running at level 7).

When a proceed (;P) command is given, the following occurs:

1. The proceed is checked for legality.
2. The processor priority is set to seven.
3. The T-bit flags (internal and user status) are set.
4. The user registers, status, and Program Counter are restored.

5. Control is returned to the user.

6. When the T-bit trap occurs, steps 1, 2, 3, 13, and 14 of the breakpoint sequence are executed, breakpoints are restored, and program execution resumes normally.

When a breakpoint is placed on an IOT, EMT, TRAP, or any instruction causing a trap, the following occurs:

1. When the breakpoint occurs as described above, ODT is entered.

2. When ;P is typed, the T-bit is set and the IOT, EMT, TRAP, or other trapping instruction is executed.

3. The current PC and status (with the T-bit included) are pushed on the stack.

4. The new PC and status (no T-bit set) are obtained from the respective trap vector.

5. The whole trap service routine is executed without any breakpoints.

6. When an RTI is executed, the saved PC and PS (including the T-bit) are restored. The instruction following the trap-causing instruction is executed. If this instruction is not another trap-causing instruction, the T-bit trap occurs, causing the breakpoints to be reinserted in the user program, or the Single-instruction mode repeat count to be decremented. If the following instruction is a trap-causing instruction, this sequence is repeated, starting at step 3.

NOTE

Exit from the trap handler must be via the RTI instruction. Otherwise, the T-bit will be lost. ODT will not gain control again since the breakpoints have not been reinserted yet.

In ODT-11, the ;P command is illegal if a breakpoint has not occurred (ODT responds with ?). In ODT-11X, ;P is legal after any trace trap entry.

WARNING

Since ODT-11 ignores all semicolons, typing the ODT-11X form of breakpoint command number to ODT-11, specifying a breakpoint number n, causes the following error:

100;B (sets the breakpoint at location 100)
100;0B (sets the breakpoint at location 1000)
100;4B (sets the breakpoint at location 1004)
The internal breakpoint status words for ODT-11 have the following format:

1. The first word contains the breakpoint address. If this location points to a location within ODT, it is assumed no breakpoint is set for the cell (specifically, ODT has set a dummy breakpoint within itself).

2. The next word contains the breakpoint repeat count.

For ODT-11X (with eight breakpoints) the formats are:

1. The first eight words contain the breakpoint addresses for breakpoints 0-7. (The ninth word contains the address of the next instruction to be executed in Single-instruction mode.)

2. The next eight words contain the respective repeat counts. (The following word contains the repeat count for Single-instruction mode.)

The user may change these words at will, either by using the breakpoint commands or by direct manipulation with $B.

When program runaway occurs (that is, when the program is no longer under ODT control, perhaps executing an unexpected part of the program where a breakpoint has not been placed) ODT may be given control by pressing the HALT key to stop the machine, and restarting ODT (see Section 5.6.2). ODT prints *, indicating that it is ready to accept a command.

If the program being debugged uses the terminal for input or output, the program may interact with ODT to causes an error since ODT also uses the terminal. This interactive error does not occur when the program being debugged is run without ODT.

1. If the terminal output interrupt is enabled upon entry to the ODT break routine, and no output interrupt is pending when ODT is entered, ODT is entered, ODT generates an unexpected interrupt when returning control to the program.

2. If the interrupt of the terminal input (the keyboard) is enabled upon entry to the ODT break routine, and the program is expecting to receive an interrupt to input a character, both the expected interrupt and the character will be lost.

3. If the terminal input (keyboard) has just read a character into the reader data buffer when the ODT break routine is entered, the expected character in the input data buffer will be lost.

5.5.3 Search

The word search allows the user to search for bit patterns in specified sections of memory. Using the $M/ command, the user specifies a mask, a lower search limit ($M+2), and an upper search limit ($M+4). The search object is specified in the search command itself.

The word search compares selected bits (where ones appear in the mask) in the word and search object. If all selected bits are equal, ODT prints the unmasked word.
The search algorithm is:

1. Fetch a word at the current address.
2. XOR (exclusive OR) the word and search object.
3. AND the result of step 2 with the mask.
4. If the result of step 3 is zero, type the address of the unmasked word and its contents. Otherwise, proceed to step 5.
5. Add two to the current address. If the current address is greater than the upper limit, type * and return to the command decoder, otherwise go to step 1.

Note that if the mask is zero, ODT prints every word between the limits, since a match occurs every time (i.e., the result of step 3 is always zero).

In the effective address search, ODT interprets every word in the search range as an instruction which is interrogated for a possible direct relationship to the search object.

The algorithm for the effective address search is (where (x) denotes contents of x, and k denotes the search object):

1. Fetch a word at the current address X.
2. If (x)=k [direct reference], print contents and go to step 5.
3. If (x)+x+2=k [indexed by PC], print contents and go to step 5.
4. If (x) is a relative branch to k, print contents.
5. Add two to the current address. If the current address is greater than the upper limit, perform a carriage return/line feed and return to the command decoder; otherwise, go to step 1.

5.5.4 Teletype Interrupt
Upon entering the TTY SAVE routine, the following occurs:

1. Save the LSR status register (TKS).
2. Clear interrupt enable and maintenance bits in the TKS.
3. Clear interrupt enable and maintenance bits in the TPS.

To restore the TTY:

1. Wait for completion of any I/O from ODT.
2. Restore the TKS.
3. Restore the TPS.
DEBUGGING OBJECT PROGRAMS ON-LINE

NOTES

If the TTY printer interrupt is enabled upon entry to the ODT break routine, the following may occur:

1. If no output interrupt is pending when ODT is entered, an additional interrupt always occurs when ODT returns control to the user.

2. If an output interrupt is pending upon entry, the expected interrupt occurs when the user regains control.

If the TTY reader (keyboard) is busy or done, the expected character in the reader data buffer will be lost.

If the TTY reader (keyboard) interrupt is enabled upon entry to the ODT break routine, and a character is pending, the interrupt (as well as the character) will be lost.

5.6 OPERATING PROCEDURES

This section describes procedures for linking ODT on LSI-11 machines, and for loading ODT on other PDP-11 machines. It describes starting, restarting, error recovery, and setting the priority level of ODT.

5.6.1 Linking Procedures (LSI-11 Systems Only)

For LSI-11 systems, ODT-11X is supplied on relocatable object tapes. Binary tapes are produced by linking the ODT-11X object tape with the object tapes of the program to be debugged (using LINK-11S). The ODT-11X tape should be the first tape processed by LINK-11S; in this manner, ODT-11X is started first when the binary tape is loaded.

5.6.2 Loading Procedures (non-LSI-11 Systems Only)

For all systems other than LSI-11, ODT is supplied on source and binary tapes. Appendix N explains assembly instructions for source tapes. Binary tapes are loaded with the Absolute Loader. Since ODT is started as soon as it is loaded, the program to be debugged should be loaded prior to ODT.

When supplied on binary tape, ODT-11 loads beginning at location 13026, and occupies about 533 (decimal) words of memory. ODT-11X loads beginning at location 12054, and requires about 800 (decimal) words of memory.
5.6.3 Starting and Restarting

The Absolute Loader starts ODT automatically after loading it into core. ODT indicates its readiness to accept input by printing an *.

The starting address for ODT-11 on binary tape is 13026; the starting address for ODT-1lx on binary tape is 12054. If ODT is reassembled using PAL-11A, the starting address in indicated in the symbol table as the value of the symbol 0.ODT. If ODT is linked using LINK-11S, the starting address is indicated in the link map as the value of the global symbol 0.ODT.

When ODT is started at its start address, the SP register is set to an ODT internal stack, registers R0-R5 are left untouched, and the trace trap vector is initialized. If ODT is started after breakpoints have been set in a program, ODT ignores the breakpoints and leaves the program modified, i.e., the breakpoint instructions are left in the program.

There are two ways to restart ODT:

1. Restart at start address+2
2. Reenter at start address+4

To restart, key in the start address+2, press LOAD ADDRESS and then START. A restart saves the general registers, removes all the breakpoint instructions from the user program and then ignores all breakpoints, i.e., simulates the ;B command.

To reenter, key in the load address+4, press LOAD ADDRESS and then START. A reenter saves the general registers, removes the breakpoint instructions from the user program, and types the BE (Bad Entry) error message. ODT remembers which breakpoints were set and resets them on the next ;G command (;P is illegal after a Bad Entry).
CHAPTER 6
LOADING AND DUMPING MEMORY

This chapter describes procedures for loading programs into memory (using the Bootstrap Loader and Absolute Loader) and for dumping the contents of memory (using the DUMPAB and/or DUMPTT programs).

The Bootstrap Loader, which loads short paper tape programs (162 or fewer octal words), appears on one of three forms, depending upon the system configuration:

1. Hardware - on some CPUs, the Bootstrap Loader is present as a ROM chip.

2. Software - on some CPUs, the Bootstrap Loader must be toggled in via console switches.

3. Firmware - on LSI-lls, the Bootstrap Loader is a firmware loader, present as a programmable ROM chip.

Once familiar with the operation of the Bootstrap Loader, the user can load other programs (such as the Absolute Loader, DUMPAB, and DUMPTT).

The Absolute Loader (see section 6.2) is a system program that enables the user to load data punched on paper tape in absolute binary format into any available memory bank. It is used primarily to load the paper tape system software, binary programs assembled with PAL-llA, and binary tapes produced by LINK-llS from object tapes produced by PAL-llS.

The loader programs are loaded into the upper-most area of available core and are available for use with system and user programs. Programs should not use the locations used by the loaders without restoring their contents; otherwise, the loaders must be reloaded since they will have been altered by the object program.

Core memory dump programs (see section 6.3) print or punch the contents of specified areas of core. For example, when developing or debugging user programs it is often necessary to get a copy of the program or portions of core. There are two dump programs supplied in the paper tape software system: DUMPTT, which prints or punches the octal representation of specified portions of core, and DUMPAB, which punches specified portions of core in absolute binary format suitable for loading with the Absolute Loader.

6-1
LOADING AND DUMPING MEMORY

6.1 PAPER TAPE BOOTSTRAPS

 Procedures for operating the various PDP-11 paper tape bootstraps are described below:

6.1.1 BM792-YA Paper Tape Bootstrap ROM

1. Set the console ENABLE/HALT switch to HALT.

2. Place the bootstrap tape in the desired paper tape reader with the special bootstrap leader code over the reader sensors (under the reader station).

3. If the low-speed reader (ASR-33) is to be used, and a high-speed reader is present on the system, turn the high-speed reader OFF. If the high-speed reader is to be used, turn it ON.

4. Set the console ENABLE/HALT switch to ENABLE.

5. Set the console switch register to 773000.

6. Press the console START switch. The contents of the bootstrap tape will be loaded into the highest locations of memory.

7. The bootstrap transfers control to the program just loaded. Typically, this program halts.

6.1.2 BM873-YA Bootstrap Loader ROM

1. Set the console ENABLE/HALT switch to HALT.

2. Place the bootstrap tape in the desired paper tape reader with the special bootstrap leader code over the reader sensors (under the reader station).

3. If the low-speed reader (ASR-33) is to be used, and a high-speed reader is present on the system, turn the high-speed reader OFF. If the high-speed reader is to be used, turn it ON.

4. Set the console ENABLE/HALT switch to ENABLE.

5a. If the low-speed reader is to be used, set the console switch register to 773210.

5b. If the high-speed reader is to be used, set the console switch register to 773312.

6. Press the console START switch. The contents of the bootstrap tape will be loaded into the highest locations of memory.

7. The bootstrap transfers control to the program just loaded. Typically, this program halts.
LOADING AND DUMPING MEMORY

6.1.3 LSI-11 Firmware Paper Tape Bootstrap

1. Press the front panel BOOT/INIT switch. This enables the micro-ODT; an @ prints at the terminal.

2. Place the bootstrap tape in the desired paper tape reader with the special bootstrap leader code over the reader sensors (under the reader station).

3. If the low-speed reader (ASR-33) is to be used, and a high-speed reader is present on the system, turn the high-speed reader OFF. If the high-speed reader is to be used, turn it ON.

4. Type the command/status register address of the input device followed by L to load the tape.

   For example, when loading from the console terminal reader, type:

   @ 177560L

   After reading the contents of the tape, the LSI-11 microprocessor starts the program, which typically halts. In this case, the micro-ODT automatically restarts and prints @ followed by the address of the instruction after the HALT instruction. For example, after loading the Absolute Loader on an 8K system, the micro-ODT prints:

   @375000
   @

   The starting address of the Absolute Loader in this case is 375000.

6.1.4 M9301-YB Bootstrap Loader

1a. If the system does not have a switch register, press the front panel BOOT/INIT switch.

1b. If the system does not have a BOOT/INIT switch, set the console switch register to 773000; press LOAD/ADDR; then press START.

2. Four numbers are printed at the terminal, followed by a $. These numbers are the contents of the general registers R0, R4, R6, and R5, respectively. For CPUs without switch registers (such as the 11/04), R5 contains the contents of the program counter (PC) at the time BOOT/INIT was pressed.

   For example:

   007740 012450 00546 004054
   $

3. Place the bootstrap tape in the desired paper tape reader with the special bootstrap leader code over the reader sensors (under the reader station).
LOADING AND DUMPING MEMORY

4. Type the device code (PR for high-speed reader, TT for terminal reader), and type RETURN, as follows:

§PR↑ or §TT↑

After reading the contents of the tape, the Bootstrap Loader transfers control to the program just loaded. Typically, this program halts.

6.1.5 M9301-YA Bootstrap Loader

If a console terminal is available, boot instructions for the M9301-YA Bootstrap Loader are the same as for the M9301-YB Bootstrap Loader (Section 6.1.4).

If no console terminal is available, the auto-boot feature of the M9301-YA must be used. See the M9301 Maintenance Manual for instructions on placing the appropriate paper tape bootstrap in the M9301 module micro-switch. Then follow the procedure below:

1. Place the bootstrap tape in the desired paper tape reader with the special bootstrap leader code over the reader sensors (under the reader station).

2. Set the console HALT/CONT switch to CONT.

3. Press the console BOOT/INIT switch. After reading the contents of the tape, the Bootstrap Loader transfers control to the program just loaded. Typically, this program halts.

6.1.6 Other Bootstrap Loaders

This section is for users without any of the bootstrap aids listed above.

The Bootstrap Loader should be loaded (toggled) into the highest core memory bank. The locations and corresponding instructions of the Bootstrap Loader are listed and explained below.

<table>
<thead>
<tr>
<th>Location</th>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx7744</td>
<td>016701</td>
</tr>
<tr>
<td>xx7746</td>
<td>000026</td>
</tr>
<tr>
<td>xx7750</td>
<td>012702</td>
</tr>
<tr>
<td>xx7752</td>
<td>000352</td>
</tr>
<tr>
<td>xx7754</td>
<td>005211</td>
</tr>
<tr>
<td>xx7756</td>
<td>105711</td>
</tr>
<tr>
<td>xx7760</td>
<td>100376</td>
</tr>
<tr>
<td>xx7762</td>
<td>116162</td>
</tr>
<tr>
<td>xx7764</td>
<td>000002</td>
</tr>
<tr>
<td>xx7766</td>
<td>xx7400</td>
</tr>
<tr>
<td>xx7770</td>
<td>005267</td>
</tr>
<tr>
<td>xx7772</td>
<td>177756</td>
</tr>
<tr>
<td>xx7774</td>
<td>000765</td>
</tr>
<tr>
<td>xx7776</td>
<td>YYYYYY</td>
</tr>
</tbody>
</table>

Figure 6-1 Bootstrap Loader Instructions
LOADING AND DUMPING MEMORY

In Figure 6-1, xx represents the highest available memory bank. For example, the first location of the Loader would be one of the following, depending on memory size, and xx in all subsequent locations would be the same as the first.

<table>
<thead>
<tr>
<th>Location</th>
<th>Memory Bank</th>
<th>Memory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>017744</td>
<td>0</td>
<td>4K</td>
</tr>
<tr>
<td>037744</td>
<td>1</td>
<td>8K</td>
</tr>
<tr>
<td>057744</td>
<td>2</td>
<td>12K</td>
</tr>
<tr>
<td>077744</td>
<td>3</td>
<td>16K</td>
</tr>
<tr>
<td>117744</td>
<td>4</td>
<td>20K</td>
</tr>
<tr>
<td>137744</td>
<td>5</td>
<td>24K</td>
</tr>
<tr>
<td>157744</td>
<td>6</td>
<td>28K</td>
</tr>
</tbody>
</table>

Note also in Figure 6-1 that the contents of location xx7766 should reflect the appropriate memory bank in the same manner as the location.

The contents of location xx7776 (yyyyyy in the Instruction column of Figure 6-1) should contain the device status register address of the paper tape reader to be used when loading the bootstrap formatted tapes. Either paper tape reader may be used, specified as follows:

- Teletype Paper Tape Reader -- 177560
- High-Speed Paper Tape Reader -- 177550

6.1.6.1 Loading the Loader Into Core - Toggle in the Bootstrap Loader as explained below.

1. Set xx7744 in the Switch Register (SR) and press LOAD ADDRESS (xx7744 is displayed in the ADDRESS REGISTER).

2. Set the first instruction, 016701, in the SR and lift DEposit (016701 is displayed in the DATA register).

NOTE

When DEPositing data into consecutive words, the DEposit automatically increments the ADDRESS REGISTER to the next word.

3. Set the next instruction, 000026, in the SR and lift DEposit (000026 is displayed in the DATA register).

4. Set the next instruction in the SR, press DEPosit, and continue depositing subsequent instructions (ensure that location xx7766 reflects the proper memory bank) until after 000765 has been deposited in location xx7774.

5. Deposit the desired device status register address in location xx7776, the last location of the Bootstrap Loader.

It is good programming practice to verify that all instructions are stored correctly. This is done by proceeding at step 6 below.

6. Set xx7744 in the SR and press LOAD ADDRESS.
LOADING AND DUMPING MEMORY

7. Press EXAMine (the octal instruction in location xx7744 is displayed in the DATA register so that it can be compared to the correct instruction, 016701. If the instruction is correct, proceed to step 8; otherwise go to step 10.

8. Press EXAMine (the instruction of the location displayed in the ADDRESS REGISTER is displayed in the DATA register; compare the DATA register contents to the instruction for the displayed location.

9. Repeat step 8 until all instructions have been verified or go to step 10 whenever the correct instruction is not displayed.

When an incorrect instruction is displayed, it can be corrected by performing steps 10 and 11.

10. With the desired location displayed in the ADDRESS REGISTER, set the correct instruction in the SR and lift DEPosit (the contents of the SR are deposited in the displayed location).

11. Press EXAMine to ensure that the instruction was correctly stored (it is displayed in the DATA register).

12. Proceed at step 9 until all instructions have been verified.

The Bootstrap Loader is now loaded into core. The procedures above are illustrated in the flowchart of Figure 6-2.

6.1.6.2 Loading Bootstrap Tapes — Any paper tape punched in bootstrap format is referred to as a bootstrap tape (see Section 6.1.3) and is loaded into core using the Bootstrap Loader. Bootstrap tapes begin with about two feet of special bootstrap leader code (ASCII code 351, not blank leader tape as required by the Absolute Loader).

With the Bootstrap Loader in core, the bootstrap tape is loaded into core starting anywhere between location xx7400 and location xx7743, i.e., 162 (octal) words. The paper tape input device used is that which is specified in location xx7776 (see section 6.1.6.1).

Bootstrap tapes are loaded into core as explained below.

1. Set the ENABLE/HALT switch to HALT.

2. Place the bootstrap tape in the specified reader with the special bootstrap leader code over the reader sensors (under the reader station).

3. Set the console switch register to xx7744 (the starting address of the Bootstrap Loader) and press LOAD ADDResS.

4. Set the ENABLE/HALT switch to ENABLE.

5. Press START. The bootstrap tape passes through the reader as data is being loaded into core.

6. The bootstrap tape stops after the last frame of data (see Figure 6-5) has been read into core. The program on the bootstrap is now in core.

The procedures above are illustrated in the flowchart of Figure 6-3.
LOADING AND DUMPING MEMORY

Figure 6-2 Loading and Verifying the Bootstrap Loader
LOADING AND DUMPING MEMORY

Figure 6-3 Loading Bootstrap Tapes Into Core

Should the bootstrap tape not read in immediately after depressing the START switch, one of the following conditions may exist:

1. Bootstrap Loader not correctly loaded.
2. Wrong input device used.
3. Code 351 not directly over the reader sensors.

6.1.6.3 Bootstrap Loader Operation - The Bootstrap Loader source program is shown below. The starting address in the example denotes that the Loader is to be loaded into memory bank zero (a 4K system).
LOADING AND DUMPING MEMORY

```
1 000000  .ASECT
2 000001 R1 = %1 ; POINTER TO DEVICE ADDRESS
3 000002 R2 = %2 ; LOAD ADDRESS DISPLACEMENT
4 000007 PC = %7 ; PROGRAM COUNTER
5 017400 LOAD = 17400 ; DATA CANNOT BE LOADED BELOW
6 THIS ADDRESS
7
8 017744 . = LOAD+344 ; STARTING ADDRESS
9
10 17744 016701 START: MOV DEVICE,R1 ; COPY DEVICE ADDRESS
10 000026
11 17750 012702 LOOP: MOV (PC)+,R2 ; COPY ADDRESS DISPLACEMENT
12 17752 000352 DSPMNT: +,-LOAD ; INITIALLY OFFSET TO THIS LOC
13 ; NOTE THAT THIS LOC IS PART OF
14 ; PREVIOUS INSTRUCTION
15 17754 005211 INC @R1 ; START THE PAPER TAPE READER
16 17756 105711 WAIT: TSTB @R1 ; FRAME READY?
17 17760 106376 BPL WAIT ; BR IF NOT
18 17762 116162 MOVB 2(R1),LOAD(R2) ; STORE FRAME READ IN MEMORY
19 17770 005267 INC DSPMNT ; INCREMENT DISPLACEMENT TO NEXT
20 17775
21 17774 000765 BRNCH: BR LOOP ; READ NEXT BYTE
22 17776 177560 DEVICE: 177560 ; ADDRESS OF INPUT DEVICE. MAY BE
23 177550 IF HIGH SPEED READER
24
25 000001' .END
```

Figure 6-4 The Bootstrap Loader Program

The program above is a brief example of the PAL-11A Assembly Language which is explained in Chapter 2.

Bootstrap tapes are coded in the following format.

```
351
. Special bootstrap leader code (at least two feet in length)
351
XXX Load offset (see text below)
AAA
BBB
CCC Program to be loaded (up to 162 words or 344 frames)
.
.
2zz
301
035
026
000
302
025
373
yyy Jump offset (see text below)
```

Figure 6-5 Bootstrap Tape Format
LOADING AND DUMPING MEMORY

The Bootstrap Loader starts by loading the device status register address into R1 and 352g into R2. The next instruction indicates a read operation in the device and the next two instructions form a loop to wait for the read operation to be completed. When data is encountered it is transferred to a location determined by the sum of the index word (xx7400) and the contents of R2.

Because R2 is initially 352g, the first word is moved to location xx7752, and it becomes the immediate data to set R2 in the next execution of the loop. This immediate data is then incremented by one and the program branches to the beginning of the loop.

The leader code, plus the increment, is equal in value to the data placed in R2 during the initialization; therefore, leader code has no effect on the loader program. Each time leader code is read the processor executes the same loop and the program remains unmodified. The first code other than leader code, however, replaces the data to be loaded into R2 with some other value which acts as a pointer to the program starting location (loading address). Subsequent bytes are read not into the location of the immediate data but into consecutive core locations. The program will thus be read in byte by byte. The INC instruction which operates on the data for R2 puts data bytes in sequential locations, and requires that the value of the leader code and the offset be one less than the value desired in R2.

The boot overlay code overlays the first two instructions of the Loader, because the last data byte is placed in the core location immediately preceding the Loader. The first instruction is unchanged by the overlay, but the second instruction is changed to place the next byte read, jump offset, into the lower byte of the branch instruction. By changing the offset in this branch instruction, the Loader can branch to the start of the loaded program or to any point within the program.

The Bootstrap Loader is self-modifying, and the program loaded by the Loader restores the Loader to its original condition by restoring the contents of locations xx7752 and xx7774 to 000352 and 000765 respectively.

6.2 THE ABSOLUTE LOADER

The Absolute Loader is a system program that enables the user to load data punched on paper tape in absolute binary format into any available memory bank. It is used primarily to load the paper tape system software, binary programs assembled with PAL-I1A, and binary tapes produced by LINK-I1S from object tapes produced by PAL-I1S. The major features of the Absolute Loader include:

1. Testing of the checksum on the input tape to assure complete, accurate loads.

2. Starting the loaded program upon completion of loading without additional user action, as specified by the .END in the program just loaded.

3. Specifying the load bias of position independent programs at load-time rather than at assembly time, by using the desired Loader switch register option.
LOADING AND DUMPING MEMORY

6.2.1 Loading the Loader Into Core

The Absolute Loader is supplied on punched paper tape in bootstrap format. Therefore, a Bootstrap Loader is used to load the Absolute Loader into core. It occupies locations xx7474 through xx7743, and its starting address is xx7500. The Absolute Loader program is 72 words long, and is loaded adjacent to the Bootstrap Loader as explained in section 6.1.6.2.

6.2.2 Using the Absolute Loader

Paper tapes punched in absolute binary format are also called absolute tapes, binary tapes, or .LDA tapes. These are the tapes loaded by the Absolute Loader.

In the following discussion, reference is made to a "switch register." For systems without switch registers (such as the LSI-11 and PDP-11/04), this term refers to a software switch register, which is a memory location internal to the Absolute Loader for systems without hardware switch registers. The location within the Absolute Loader is xxx516, where xxx reflects memory size as follows:

<table>
<thead>
<tr>
<th>Memory</th>
<th>xxx</th>
</tr>
</thead>
<tbody>
<tr>
<td>4K</td>
<td>017</td>
</tr>
<tr>
<td>8K</td>
<td>037</td>
</tr>
<tr>
<td>12K</td>
<td>057</td>
</tr>
<tr>
<td>16K</td>
<td>077</td>
</tr>
<tr>
<td>20K</td>
<td>117</td>
</tr>
<tr>
<td>24K</td>
<td>137</td>
</tr>
<tr>
<td>28K</td>
<td>157</td>
</tr>
</tbody>
</table>

When text indicates that a value be placed in a switch register, users without hardware switch registers must use either the M9301 console emulator or the LSI-11 micro-ODT, as appropriate, to store the switch register value in location xxx516. Once this value has been stored, the user starts the Absolute Loader at location xxx500. Once the Absolute Loader is loaded, it initializes the value of location xxx516 to 0. This value changes only when modified by the user.

A normal load occurs when data is loaded into memory according to the load addresses on the binary tape. The user must set bit 0 of the switch register to 0 immediately before starting the load.

There are two types of relocated loads:

1. Loading to continue from where the loader left off after the previous load -

   This is used, for example, when the object program being loaded is contained on more than one tape. It is specified by setting the switch register to 000001 immediately before starting the load.

2. Loading into a specific area of core -

   This is normally used when loading position independent programs. A position independent program is one which may be loaded and run anywhere in available core. The program is written using the position independent instruction format (see Chapter 9). This type of load is specified by setting the switch register to the load bias and adding 1 to it
LOADING AND DUMPING MEMORY

(i.e., setting bit 0 to 1). The effect of this is to add the value in the switch register to the start address on the tape.

Optional switch register settings for the three types of loads are listed below.

<table>
<thead>
<tr>
<th>Type of Load</th>
<th>Switch Register Bits 1-14</th>
<th>Bit 0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>(ignored)</td>
<td>0</td>
</tr>
<tr>
<td>Relocated - continue loading where left off</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Relocated - load in specified area of core</td>
<td>nnnnn (specified address)</td>
<td>1</td>
</tr>
</tbody>
</table>

The absolute tape may be loaded using either of the paper tape readers. The desired reader is specified in the last word of available core memory (xx7776), the input device status word, as explained in section 6.1.6. The input device status word may be changed at any time prior to loading the absolute tape.

With the Absolute Loader in core as explained in section 6.1.6.2, absolute tapes are loaded as explained below.

1. Set the ENABLE/HALT switch to HALT.

   To use an input device different from that used when loading the Absolute Loader, change the address of the device status word (in location xx7776) to reflect the desired device, i.e., 177560 for the Teletype reader or 177550 for the high-speed reader.

2. Set the switch register to xx7500 and press LOAD ADDR.

3. Set the switch register to reflect the desired type of load (Figure E-3 in Appendix E).

4. Place the absolute tape in the proper reader with blank leader tape directly over the reader sensors.

5. Set ENABLE/HALT to ENABLE.

6. Press START. The absolute tape begins passing through the reader station as data is being loaded into core.

If the absolute tape does not begin passing through the reader station, the Absolute Loader is not in core correctly. Reload the Loader and start over at step 1 above. If it halts in the middle of the tape, a checksum error occurred in the last block of data read in.

Normally, the absolute tape stops passing through the reader station when it encounters the transfer address as generated by the statement, .END, denoting the end of a program. If the system halts after loading, check that the low byte of the DATA register is zero. If so, the tape is correctly loaded. If not zero, a checksum error (explained later) has occurred in the block of data just loaded, indicating that some data was not correctly loaded. Reload the tape starting at step 1 above.
LOADING AND DUMPING MEMORY

When loading a continuous relocated load, subsequent blocks of data are loaded by placing the next tape in the appropriate reader and pressing the CONTinue switch.

The Absolute Loader may be restarted at any time by starting at step 1 above.

6.2.3 Absolute Loader Operation

The Loader uses the eight general registers (R0-R7) and does not preserve or restore their previous contents. Therefore, caution should be taken to restore or load these registers when necessary after using the Loader.

A block of data punched on paper tape in absolute binary format has the following format.

<table>
<thead>
<tr>
<th>FRAME</th>
<th>Data</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>001</td>
</tr>
<tr>
<td>2</td>
<td>000</td>
</tr>
<tr>
<td>3</td>
<td>xxx</td>
</tr>
<tr>
<td>4</td>
<td>xxx</td>
</tr>
<tr>
<td>5</td>
<td>yyy</td>
</tr>
<tr>
<td>6</td>
<td>yyy</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>zzz</td>
<td></td>
</tr>
</tbody>
</table>

A program on paper tape may consist of one or more blocks of data. Each block with a byte count (frames 3 and 4) greater than six causes subsequent data to be loaded into core (starting at the address specified in frames 5 and 6 for a normal load). The byte count is a positive integer denoting the total number of bytes in the block, excluding the checksum. When the byte count of a block is six, the specified load address is checked to see whether the address is to an even or to an odd location. If even, the Loaded transfers control to the address specified. Thus the loaded program runs upon completion of loading. If odd, the loader halts.

The transfer address (TRA) may be explicitly specified in the source program by placing the desired address in the operand field following the .END statement. For example,

.END ALPHA

specifies the symbolic location ALPHA as the TRA, and

.END

causes the Loader to halt. With

.END nnnnnn

the Loader also halts if the address (nnnnnn) is odd.

The checksum is displayed in the low byte of the DATA register of the computer console. Upon completion of a load, the low byte of the DATA register should be all zeros (unlit). Otherwise, a checksum error has occurred, indicating that the load was not correct. The checksum is the low-order byte of the negation of the sum of all the previous bytes in the block. When all bytes of a block including the checksum are added together, the low-order byte of the result should be zero.
LOADING AND DUMPING MEMORY

If not, some data was lost during the load or erroneous data was picked up; the load was incorrect. When a checksum error is displayed, the entire program should be reloaded, as explained in the previous section. The loaders occupy core memory as illustrated below.

<table>
<thead>
<tr>
<th>xx7776</th>
<th>I/O DEVICE WORD</th>
</tr>
</thead>
<tbody>
<tr>
<td>xx7744</td>
<td>BOOTSTRAP LOADER</td>
</tr>
<tr>
<td>xx7500</td>
<td>ABSOLUTE LOADER</td>
</tr>
<tr>
<td>xx7474</td>
<td>LOADER STACK</td>
</tr>
</tbody>
</table>

6.3 CORE MEMORY DUMPS

A core memory dump program is a system program which enables the user to dump (print or punch) the contents of any specified portion of core memory onto the Teletype printer and/or punch, line printer or high-speed punch. There are two dump programs available in the Paper Tape Software System:

1. DUMPTT\(^1\), which dumps the octal representation of the contents of specified portions of core onto the teleprinter, low-speed punch, high-speed punch, or line printer.

2. DUMPAB, which dumps the absolute binary code of the contents of specified portions of core onto the low-speed punch or high-speed punch.

Both dump programs are supplied on punched paper tape in bootstrap and absolute binary formats. The bootstrap tapes are loaded over the Absolute Loader as explained in section 6.1.6.3, and are used when it would be undesirable to alter the contents of user storage (below the Absolute Loader). The absolute binary tapes are position independent and may be loaded and run anywhere in core as explained in section 6.2.2.

DUMPTT and DUMPAB are similar in function, and differ primarily in the type of output they produce.

6.3.1 Operating Procedures

Neither dump program punches leader or trailer tape, but DUMPAB always punches ten blank frames of tape at the start of each block of data dumped.

\(^1\)DUMPTT is not available for systems without switch registers.
LOADING AND DUMPING MEMORY

6.3.1.1 Using DUMPAB on Systems Without Switch Registers - Operating procedures for DUMPAB on systems without switch registers are as follows:

1. Select either the absolute binary or the bootstrap version of DUMPAB and place it in the reader specified by location xx7776 (see section 6.1).

2a. If using a bootstrap tape, load the tape using the procedure outlined in section 6.1. When the computer halts, go to step 3.

2b. If using an absolute binary tape, load the tape using the procedure outlined in section 6.2.2, relocating as follows:

   a. Select the address to which the program is to be relocated. The relocation offset is then equal to the loading address. For example, if the desired relocation address is 000400, the relocation offset is 000401.

   b. Deposit the relocation offset with bit 0 set in the Absolute Loader's software switch register. Using the example from the previous step, the user would deposit 000401 into location xxx516.

Start the Absolute Loader.

3. When the program halts, find the address in the program counter. For LSI-11 machines, the value is printed at the console terminal by the micro-ODT. For UNIBUS PDP-11 machines, the user must press the BOOT/INIT switch to obtain register values at the console terminal (see section 6.1.4). The last of the four values displayed is the PC contents.

Add 2 to the value of the PC. (For example, the PC contents for the bootstrap version of DUMPAB are xxx516; adding 2 to this value gives xxx520.) This new value is the address of the first of these succeeding parameters, described in subsequent steps.

4. Deposit the address of the first byte to be dumped into the first parameter (whose address was determined in the previous step).

5. Deposit the address of the last byte to be dumped into the second parameter (next sequential location).

6. The third parameter contains the value 177564 (a default specifying the ASR-33 punch). If this is the first time this step is executed and the high-speed reader is the desired output device, change the value of the third parameter to 177554.

7a. If using the LSI-11, type P to proceed.

7b. If using a UNIBUS PDP-11, restart the program (at xxx510 if bootstrap tape); press CONT when the program halts.

8. DUMPAB dumps the specified segment of memory and halts.

9. Repeat steps 4 through 8 until all desired memory segments have been dumped.
LOADING AND DUMPING MEMORY

10. A transfer block for DUMPAB must be generated to terminate the dump. This value must be deposited in the first parameter (step 4) to terminate DUMPAB. If the tape is not to be self-starting, use 000001 as the transfer address. Under no conditions can 000000 be used as the transfer address.

11. Deposit 000000 in the second parameter (as in step 5).

12. Repeat step 7a or 7b, as appropriate, to punch the transfer block.

6.3.1.2 Using DUMPAB and DUMPTT on Systems with Switch Registers

1. Select the dump program desired and place it in the reader specified by location xx7776 (see Section 6.1).

2. If a bootstrap tape is selected, load it using the Bootstrap Loader, section 6.1.6.2. When the computer halts go to step 4.

3. If an absolute binary tape is selected, load it using the Absolute Loader (section 6.2.2), relocating as desired.

Place the proper start address in the switch register, press LOAD ADDRESS and START. (The start addresses are shown in section 6.3.3).

4. When the computer halts, enter the address of the desired output device status register in the switch register and press CONTINUE (low-speed punch and teleprinter = 177564; high-speed punch = 177554; line printer = 177514).

5. When the computer halts, enter in the switch register the address of the first byte to be dumped and press CONTINUE. This address must be even when using DUMPTT.

6. When the computer halts again enter in the switch register the address of the last byte to be dumped and press CONTINUE. When using the low-speed punch, set the punch to ON before pressing CONTINUE.

7. Dumping proceeds on the selected output device.

8. When dumping is complete, the computer halts.

If further dumping is desired, proceed to step 5. It is not necessary to respecify the output device address except when changing to another output device. In such a case, proceed to the second paragraph of step 3 to restart.

If DUMPAB is being used, a transfer block must be generated as described below. If a tape read by the Absolute Loader does not have a transfer block, the loader will wait in an input loop. In such a case, the program may be manually initiated. However, this practice is not recommended, as there is no guarantee that load errors will not occur when the end of the tape is read.

The transfer block is generated by performing step 5 with the transfer address in the Switch Register, and step 6 with the transfer address minus 1 in the Switch Register. If the tape is not to be self-starting, an odd-numbered address must be specified in step 5 (000001, for example).
LOADING AND DUMPING MEMORY

The dump programs use all eight general registers and do not restore their original contents. Therefore, after a dump the general registers should be loaded as necessary prior to their use by subsequent programs.

6.3.2 Output Formats

The output from DUMPTT is in the following format:

xxxxxx>yyyyyy yyyyy yyyyy yyyyy yyyyy yyyyy yyyyy

where xxxxxx is the octal address of the first location printed or punched, and yyyyy are words of data, the first of which starts at location xxxxxx. This is the format for every line of output. There will be no more than eight words of data per line, but there will be as many lines as are needed to complete the dump.

The output from DUMPAB is in absolute binary, as explained in section 6.2.3.

6.3.3 Storage Maps

The DUMPTT program is 87 words long. When used in absolute format the storage map is:

```
xx7776
xx7744
xx7500
xx7474

xxxxxx+256

xxxxxx
```

When used in bootstrap format the storage map is:
LOADING AND DUMPING MEMORY

xx7776

BOOTSTRAP LOADER

xx7744

DUMPTT

start address = xx7440
xx7434

TWO-WORD STACK SPACE

The DUMPAB program (for systems with a switch register) is 65(10) words long. When used in absolute format the storage map is:

xx7776

BOOTSTRAP LOADER

xx7744

ABSOLUTE LOADER

xx7500
xx7474

LOADER STACK SPACE

xxxxxxxx+244

DUMPAB

xxxxxxxx

THREE-WORD STACK SPACE

xxxxxxxx = desired load address = start address

When used in bootstrap format the storage map is:
LOADING AND DUMPING MEMORY

\( \text{BOOTSTRAP LOADER} \)

\( \text{DUMPAB} \)

\( \text{THREE-WORD STACK SPACE} \)

\( \text{start address} = xx7510 \\
\text{xx7500} \)

The DUMPAB program (for systems without a switch register) is 82(10) words long. When used in absolute format the storage map is:

\( \text{BOOTSTRAP LOADER} \)

\( \text{ABSOLUTE LOADER} \)

\( \text{LOADER STACK SPACE} \)

\( \text{DUMPAB} \)

\( \text{TWO-WORD STACK SPACE} \)

\( xxxxxx = \text{desired load address} = \text{start address} \)
LOADING AND DUMPING MEMORY

When used in bootstrap format the storage map is:

```
x
xx7776
xx7744
start address = xx7500
xx7474
```

<table>
<thead>
<tr>
<th>BOOTSTRAP LOADER</th>
</tr>
</thead>
<tbody>
<tr>
<td>DUMPAB</td>
</tr>
<tr>
<td>TWO-WORD STACK SPACE</td>
</tr>
</tbody>
</table>

6-20
CHAPTER 7
INPUT/OUTPUT PROGRAMMING

7.1 INTRODUCTION

The PDP-11 Input/Output eXecutive (IOX), frees the user from dealing directly with I/O devices. It provides programming formats that allow programs written for the paper tape software system to be used later in a monitor environment with only minor coding changes.

IOX provides asynchronous I/O service for the following non-file-oriented devices:

1. Teletype keyboard, printer, and tape reader and punch
2. High-speed paper tape reader and punch

For line printer handling, in addition to all IOX facilities, IOXLPT is available.

Simple I/O requests can be made, specifying devices and data forms for interrupt-controlled data transfers, that can occur concurrently with the execution of a user program. Multiple I/O devices can run single- or double-buffered I/O processing simultaneously.

Real-time capability is provided by allowing user programs to be executed at device priority levels upon completion of a device action or data transfer.

Communication with IOX is accomplished by IOT (Input/Output Trap) instructions in the user's program. Each IOT is followed by two or three words consisting of one of the IOX commands and its operands. The IOX commands can be divided into two categories:

1. those concerned with establishing necessary conditions for performing input and output (mainly initializations), and
2. those concerned directly with the transfer of data.

When transfer of data is occurring, IOX is operating at the priority level of the device. The calling program runs at its priority level, either concurrent with the data transfer, or sequentially.

Programming format for commands is:

```
IOT
.WORD (an address)
.BYTE (a command code), (a slot number)
```

Before using the data transfer commands, two preparatory tasks must be performed:
INPUT/OUTPUT PROGRAMMING

1. Since device specifications are made by referring to "slots" in IOX's Device Assignment Table (DAT) rather than devices themselves, the slots specified in the code must have devices assigned to them.

2. The buffer, whose address is specified in the code, must be set up with information about the data.

In those non-data-transfer commands where an address or slot number does not apply, a 0 must be used. Addresses or codes indicated can, of course, be specified symbolically.

The following program segment illustrates a simple input-process-output sequence. It includes:

1. The setting up of a single buffer
2. All necessary initializations
3. A formatted ASCII read into the buffer
4. A wait for completion of the read
5. Processing of data just read
6. A write command from the buffer.

```
RESET=2  ;ASSIGN IOX COMMAND CODES
READ=11
WAITR=4
WRITE=12

IOT    ;IOX RESET TO DO NECESSARY
  .WORD 0
  .BYTE RESET, 0
  ;INITIALIZATIONS INCLUDING
  ;INITING SLOT 0 FOR KBD, AND 1 FOR TTY

IOT    ;TRAP TO IOX
  .WORD BUFFER
  .BYTE READ, 0
  ;SPECIFY BUFFER
  ;READ FROM KBD (SLOT 0) TILL
  ;LINE FEED OR FORM FEED

WAIT:  ;TRAP TO IOX
  IOT    ;BUSY RETURN ADDRESS WHILE WAITING
  .WORD WAIT
  .BYTE WAITR, 0
  ;FOR KBD TO FINISH
  ;WAIT FOR KBD (SLOT 0) TO FINISH
  (process BUFFER)

IOT    ;TRAP TO IOX
  .WORD BUFFER
  .BYTE WRITE, 1
  ;WRITE TO TELEPRINTER (SLOT 1)

BUFFER: 100  ;BUFFER SIZE IN BYTES
0  ;CODE FOR FORMATTED ASCII MODE
0  ;IOX WILL SET HERE THE NUMBER OF BYTES READ
,=,+100  ;STORAGE RESERVED FOR 100 BYTES
```

In more complex programming it is likely that more than one buffer will be set up for the transfer of data, so that data processing can occur concurrently rather than sequentially, as here. Note too, that there are five IOX commands not used in this example that will help meet the requirements of I/O problems not as straightforward as this.

7-2
7.1.1 Using IOX With The LSI-11 Processor

IOX (IOXLPT) is supplied on source and relocatable object tapes. It is thus unnecessary to assemble IOX unless the program is to be modified. User object tapes can be linked with the IOX object tape (using LINK-11S) to produce an absolute binary tape. Appendix J describes assembly procedures for source tapes.

IOX requires approximately 633 (decimal) words of core; IOXLPT requires approximately 724 (decimal) words.

7.1.2 Using IOX with Unibus PDP-11 Processors

IOX (IOXLPT) is supplied on source and binary tapes. Appendix J describes assembly procedures for source tapes. Binary tapes are loaded prior to user programs by the Absolute Loader. After IOX is loaded, the Absolute Loader halts.

IOXLPT is used instead of IOX if the program uses a line printer.

IOX is supplied on an absolute binary tape with a loading address of 15100; the load address for IOXLPT is 34600. If the user desires different load addresses, the programs must be reassembled as described in Appendix J.

IOX requires approximately 634 (decimal) words of core; IOXLPT requires approximately 725 (decimal) words.

7.1.3 IOX Interrupt and Trap Vectors

IOX (IOXLPT) loads the following interrupt and trap vectors:

- Console terminal
- High speed reader and punch
- Timeout and other errors
- IOT
- Line printer (IOXLPT only)

7.2 THE DEVICE ASSIGNMENT TABLE

The Device Assignment Table (DAT) makes programs device-independent by allowing the user to refer to a slot to which a device has been assigned, rather than a specific device itself. Thus, changing the input or output device becomes a simple matter of reassigning a different device to the slot indicated in the program.

The DAT is created by means of the Reset and/or Init commands. The IOX codes for devices (listed in the description of the Init command below) are assigned to the slots.
INPUT/OUTPUT PROGRAMMING

7.2.1 Reset

IOT
.WORD 0
.BYTE 2, 0

This command must be the first IOX command issued by a user program. It clears the DAZ, initializes IOX, resets all devices to their state at power-up, enables keyboard interrupts, and initializes DAT slots 0 and 1 for the keyboard and teleprinter, respectively.

7.2.2 Initialization

IOT
.WORD (address of device code)
.BYTE 1, (slot number)

The device whose code (stored as a byte) is found at the specified address is associated with the specified slot (numbered in the range 0-7). The device interrupt is turned off when necessary. (The keyboard interrupt always remains enabled.) There is no restriction on the number of slots that can be initialized to the same device.

<table>
<thead>
<tr>
<th>DEVICE</th>
<th>DEVICE CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Teletype Keyboard (KBD)</td>
<td>1</td>
</tr>
<tr>
<td>Teletype printer (TTY)</td>
<td>2</td>
</tr>
<tr>
<td>Low-Speed Reader (LSR)</td>
<td>3</td>
</tr>
<tr>
<td>Low-Speed Punch (LSP)</td>
<td>4</td>
</tr>
<tr>
<td>High-Speed Reader (HSR)</td>
<td>5</td>
</tr>
<tr>
<td>High-Speed Punch (HSP)</td>
<td>6</td>
</tr>
<tr>
<td>Line Printer (IOXLPT only) (LPT)</td>
<td>10</td>
</tr>
</tbody>
</table>

Note that a device code is used only in the Initialization (INIT) command. All other commands that refer to a device do so by means of a slot. Example:

```
INIT=1
IOT
.WORD HSRCOD
.BYTE INIT=3
...$FOR HSR

HSRCOD: .BYTE 5 $HSR CODE
```

7.3 BUFFER ARRANGEMENT IN DATA TRANSFER COMMANDS

Use of data-transfer commands (Read, Write, Real-time Read, Real-time Write) requires the creation of at least one buffer. This buffer is used not only to store data for processing, but to hold information regarding the quantity, form, and status of the data. The non-data
portion of the buffer is called the buffer header, and precedes the
data portion. In data transfer commands, it is the address of the
first word of the buffer header that is specified in the word
following the IOT of the command.

NOTE

IOX uses the buffer header while
transferring data. The user's program
must not change or reference it.

The buffer format is:

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer</td>
<td>Maximum number of data bytes (unsigned integer)</td>
</tr>
<tr>
<td>Buffer+2</td>
<td>Mode of data (byte)</td>
</tr>
<tr>
<td>Buffer+3</td>
<td>Status of data (byte)</td>
</tr>
<tr>
<td>Buffer+4</td>
<td>Number of data bytes involved in transfer (unsigned integer)</td>
</tr>
<tr>
<td>Buffer+6</td>
<td>Actual data begins here</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BUFFER SIZE (IN BYTES)</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATUS</td>
</tr>
<tr>
<td>-------</td>
</tr>
<tr>
<td>BYTE COUNT</td>
</tr>
<tr>
<td>DATA</td>
</tr>
<tr>
<td>.</td>
</tr>
<tr>
<td>.</td>
</tr>
</tbody>
</table>

7.3.1 Buffer Size
The first word of the buffer contains the size (in bytes) of the data
portion of the buffer as specified by the user. IOX will not store
more than this many data bytes on input. Buffer size has no meaning
on output.

7.3.2 Mode Byte
The low-order byte of the second word holds information concerning the
mode or transfer. A choice of four modes exists:

Coded as

1. Formatted ASCII    0  (or 200 to suppress echo)
2. Formatted Binary   1
3. Unformatted ASCII   2  (or 202 to suppress echo)
4. Unformatted Binary  3

7-5
The term echo applies only to the KBD. Data transfers from other devices never involve an echo.

### MODE BYTE

<table>
<thead>
<tr>
<th>Bits</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>=1</td>
</tr>
<tr>
<td>NO ECHO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>=0</td>
</tr>
<tr>
<td>ECHO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### 7.3.3 Status Byte

The high-order byte of the second word of the buffer header contains information set by IOX on the status of the data transfer:

- Bits 0-4 contain the non-fatal error codes (coded octally)
- Bit 5: 1 = End-Of-File has occurred (attempt at reading data after an End-Of-Medium)
- Bit 6: 1 = End-of-Medium has occurred (see Section 7.3.3.3)
- Bit 7: 1 = Done (Data Transfer complete)

### STATUS BYTE

<table>
<thead>
<tr>
<th>Bits</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>DONE</td>
</tr>
<tr>
<td>1=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EOM</td>
</tr>
<tr>
<td>1=</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>EOF</td>
</tr>
<tr>
<td>SEE CODES</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- NON-FATAL ERRORS

#### 7.3.3.1 Non-Fatal Error Codes (Octal)

- 2 = checksum error
- 3 = truncation of a long line
- 4 = an improper mode

1. A checksum error can occur only on a Formatted Binary read (see Section 7.4.3).
INPUT/OUTPUT PROGRAMMING

2. Truncation of a long line can occur on either a Formatted Binary or Formatted ASCII read (Section 7.4.1). This error occurs when the binary block or ASCII line is bigger than the buffer size specified in the buffer header. In both cases, IOX continues reading characters into the last byte in the buffer until the end of the binary block or ASCII line is encountered.

3. An improper mode can occur only on a Formatted Binary read. Such occurrence means that the first non-null character encountered was not the proper starting character for a Formatted Binary block (see Section 7.4.3).

7.3.3.2 Done Bit - When the data transfer to or from the buffer is complete, the Done Bit is set by IOX.

7.3.3.3 End-Of-Medium Bit - The following conditions cause the EOM bit to be set in the buffer Status byte associated with a data transfer command. An EOM occurrence also sets the Done Bit.

<table>
<thead>
<tr>
<th>HSR</th>
<th>HSP</th>
<th>LSR</th>
<th>LPT</th>
</tr>
</thead>
<tbody>
<tr>
<td>No tape</td>
<td>No tape</td>
<td>Timeout detected</td>
<td>No paper</td>
</tr>
<tr>
<td>Off line</td>
<td>No power</td>
<td></td>
<td>No power Printer drum gate open</td>
</tr>
<tr>
<td>No power</td>
<td></td>
<td></td>
<td>Overtemperature condition</td>
</tr>
</tbody>
</table>

An End-Of-Medium condition on an output device is cleared by a manual operation such as putting a tape in the high-speed punch. IOX does not retain any record of an EOM on an output device. However, an EOM on an input device is recorded by IOX so that succeeding attempts to read from that device will cause an End-Of-File (see Section 7.3.3.4). To reenable input the device must be manually readied and a Seek command (Section 7.6) executed on the proper slot. The INIT and RESET commands will also clear the EOM condition for the device.

See Section 7.5.3 for information on detection of conditions causing LSR timeouts.

When an End-Of-Medium has occurred on a Read, there may be data in the buffer. If an EOM has occurred on a Write, there is no way of knowing how much of the buffer was written.

7.3.3.4 End-Of-File Bit - An EOF condition appears in the Status byte if an attempt to read is made after an EOM has occurred. EOF cannot occur on output. When an EOF has occurred, no data is available in the buffer.
7.3.4 Byte Count

The third word contains the Byte Count:

Input: In unformatted data modes, IOX reads as many data bytes as the user has specified. In formatted modes, IOX inserts here the number of data bytes available in the buffer. In all modes, if an EOM occurs, IOX will set the Byte Count equal to the number of bytes actually read. If an EOF occurs, Byte Count will be set to 0.

Output: Byte Count determines the number of bytes output, for all modes. An HSP end-of-tape or LPT out-of-paper condition will also terminate output, and EOM will be set in the Status byte. IOX does not modify the Byte Count on output.

7.4 MODES

7.4.1 Formatted ASCII

A Formatted ASCII read transfers 7-bit characters (bit 8 will be zero) until a line feed or form feed is read. IOX sets the Byte Count word in the buffer header to indicate the number of characters in the buffer. If the line is too long, characters are read and overlaid into the last byte of the buffer until an end-of-line (a line feed or form feed) or EOM is detected. Thus, if there is no error, the buffer will always contain a line feed or form feed.

A Formatted ASCII write transfers the number of 7-bit characters specified by the buffer Byte Count. Bit 8 will always be output as zero.

Device-Dependent Functions

Keyboard

Seven-bit characters read from the keyboard are entered in the buffer and are echoed on the teleprinter except as follows:

Null - Ignored. This character is not echoed or transferred to the buffer.

Tab (CTRL/TAB keys) - Echoes as spaces up to the next tab stop. "Stops" are located at every 8th carriage position.

RUBOUT - Deletes the previous character on the current line and echoes as a backslash (\). If there are no characters to delete, RUBOUT is ignored.

CTRL/U - Deletes the current line and echoes as ^U.

Carriage Return (RETURN key) - Echoes as a carriage return followed by a line feed. Both characters enter the buffer.

CTRL/P - Echoes as ^P and causes a jump to the restart address, if non-zero (see 7.6.2).

The echo may be suppressed by setting bit 7 of the buffer header Mode byte.
If the buffer overflows, only the characters which fit into the buffer are echoed. Of course, characters which are deleted by RUBOUT or CTRL/U do not read into the buffer even though they are echoed. If a carriage return causes an overflow, or is typed after an overflow has occurred, a carriage return and line feed will be echoed but only the line feed will enter the buffer.

In the following Formatted ASCII examples:

1. assume there is room for five characters

2. \ indicates:
   \ in left column, the RETURN key
   \ in center column, the execution of a carriage return
   \ in right column, the ASCII code for carriage return

3. + indicates:
   + in center column, the execution of a line feed
   + in right column, the ASCII code for line feed

4. RUB OUT indicates the RUBOUT key

5. CTRL indicates the CTRL and U keys.
   U

<table>
<thead>
<tr>
<th>Typed</th>
<th>Echoed</th>
<th>Entered Buffer</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABC\</td>
<td>ABC\</td>
<td>ABC\</td>
</tr>
<tr>
<td>ABCD\</td>
<td>ABCD\</td>
<td>ABCD\</td>
</tr>
<tr>
<td>ABCDEF\</td>
<td>ABCD\</td>
<td>ABCD\</td>
</tr>
<tr>
<td>ABCDEF RUB| ABCD\</td>
<td>ABCD\</td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CTRL RUB U</td>
<td>+U\</td>
<td>U\</td>
</tr>
<tr>
<td>OUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCDEF RUB</td>
<td>ABCD\</td>
<td>A\</td>
</tr>
<tr>
<td>RUB OUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ABCDEF RUB RUB</td>
<td>ABCD \</td>
<td>A\</td>
</tr>
<tr>
<td>OUT OUT OUT</td>
<td>\X\</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Low-Speed Reader and High-Speed Reader

All characters are transferred to the buffer except that nulls and rubouts are ignored.

Teleprinter

Characters are printed from the buffer as they appear except that nulls are ignored and tabs are output as spaces up to the next tab stop.

Low-Speed Punch and High-Speed Punch

Characters are punched from the buffer as they appear except that nulls are ignored and tabs are followed by a rubout.
INPUT/OUTPUT PROGRAMMING

Line Printer (IOXLPT only)

Characters are printed from the buffer as they appear except as follows:

- Nulls - Ignored
- Tab - Output as spaces up to the next tab stop.
- Carriage Return - Ignored. It is assumed that a line feed or form feed follows. These characters cause the line printer "carriage" to advance.

All characters beyond the 80th are ignored except a line feed or form feed.

7.4.2 Unformatted ASCII

Unformatted ASCII transfers the number of 7-bit characters specified by the header Byte Count.

Device-Dependent Functions

Keyboard

Characters are read and echoed except as follows;

- Tab - Echoes as spaces up to the next tab stop.
- CTRL/P - Echoes as "P and causes a jump to the restart address, if non-zero (see 7.6.2).

7.4.3 Formatted Binary

Formatted Binary is used to transfer checksummed binary data (8-bit characters) in blocks. A Formatted Binary block appears as follows:

<table>
<thead>
<tr>
<th>Byte (Octal)</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Start of block</td>
</tr>
<tr>
<td>000</td>
<td>Always null</td>
</tr>
<tr>
<td>XXX XXX</td>
<td>Block Byte Count (low-order followed by high-order). Count includes data and preceding four bytes.</td>
</tr>
<tr>
<td>DDD DDD . . .</td>
<td>Data bytes</td>
</tr>
<tr>
<td>DDD DDD</td>
<td>Checksum. Negation of the sum of all preceding bytes in the block.</td>
</tr>
</tbody>
</table>

7-10
INPUT/OUTPUT PROGRAMMING

IOX creates the block on output, from the buffer and buffer header. The Byte Count word in the buffer header specifies the number of data bytes following, which are to be output. Note that the Byte Count output is four larger than the header Byte Count. As the block is output, IOX calculates the checksum which is output following the last data byte.

On Formatted Binary reads, IOX ignores null characters until the first non-null character is read. If this character is a 001, a Formatted Binary block is assumed to follow and is read from the device under control of the Byte Count value. If the first non-null character is not 001, the read is immediately terminated and error code 4 is set in the Status byte. As the block is read a checksum is calculated and compared to the checksum following the block. If the checksum is incorrect, error code 2 is set in the Status byte of the buffer header. If the binary block is too large (Byte Count less 4, larger than the Buffer Size specified in the header), the last byte of the buffer is overlaid until the last data byte has been read; error code 3 is set in the Status byte.

Device-Dependent Functions

None. Eight-bit data characters are transferred to and from the device and buffer exactly as they appear.

7.4.4 Unformatted Binary

This mode transfers 8-bit characters with no formatting or character conversions of any kind. For both input and output, the buffer header Byte Count determines the number of characters transferred.

Device-Dependent Functions

None

7.5 DATA TRANSFERS

7.5.1 Read

IOR
.WORD (address of first word of the buffer header)
.BYTE ll, (slot number)

This command causes IOX to read from the device associated with the specified slot according to the information found in the buffer header. IOX initiates the transfer of data, clears the Status byte, and returns control to the calling program. If the device on the selected slot is busy, or a conflicting device (see Section 7.5.3) is busy, IOX retains control until the data transfer can be initiated. Upon completion of the Read, the appropriate bits in the Status byte are set by IOX and the Byte Count word indicates the number of bytes in the data buffer. Note that use of the KBD while an LSR Read is in progress will intersperse KBD characters into the buffer unpredictably.
7.5.2 Write

IOT
.WORD (address of first word of the buffer header)
.BYTE 12, (slot number)

IOX writes on the device associated with the specified slot according to the information found in the buffer header. Transfer of data occurs in the amount specified by Byte Count (Buffer+4). IOX returns control to the calling program as soon as the transfer has been initiated. If the device on the selected slot is busy, or a conflicting device is busy, IOX retains control until the transfer can be initiated. Upon completion of the Write, IOX will set the Status byte to the latest conditions. If a Write causes an EOM condition, the user has no way of determining how much of his buffer has been written (the Byte Count remains the same.)

7.5.3 Device Conflicts In Data Transfer Commands

Because there is a physical association between the devices on the ASR Teletype, certain devices cannot be in use at the same time. When a data transfer command is given, IOX simultaneously checks for two conditions before executing the command:

1. Is the device requested already in use? and,

2. Is there some other device in use that would result in an operational conflict?

IOX resolves both conflict situations by waiting until the first device is no longer busy, before allowing the requested device to start functioning. (This is an automatic Waitr command. See next section.) For example, if the LSR is in use, and either a KBD request or a second request for the LSR itself is made, IOX will wait until the current LSR read has been completed before returning control to the calling program. In the particular case of the LSR, IOX also performs a timeout check while waiting for it to become available.

When a Read command has been issued for the LSR, IOX waits about 100 milliseconds for each character to be read. If no character is detected by this time (presumably because the LSR is turned off, or out of tape), a timeout is declared and IOX sets EOM in the appropriate buffer Status byte.

The following is a table listing the devices. Corresponding to each device on the left is a list of devices (or the echo operation) which would conflict with it in operation.

<table>
<thead>
<tr>
<th>Device</th>
<th>All Possible Conflicting Devices or Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBD</td>
<td>CHO, KBD, TTY, LSR, LSP</td>
</tr>
<tr>
<td>TTY</td>
<td>Echo, KBO, TTY, LSP</td>
</tr>
<tr>
<td>LSR</td>
<td>KBD, LSR</td>
</tr>
<tr>
<td>LSP</td>
<td>Echo, KBD, TTY, LSP</td>
</tr>
<tr>
<td>HSR</td>
<td>HSR</td>
</tr>
<tr>
<td>HSP</td>
<td>HSP</td>
</tr>
<tr>
<td>LPT (IOXLPT only)</td>
<td>LPT</td>
</tr>
</tbody>
</table>

7-12
7.5.4 **Waitr (Wait, Return)**

```
IOT
.WORD (busy return address)
.BYTE 4, (slot number)
```

Waitr, like device conflict resolution, causes IOX to test the status of the device associated with the specified slot. If the device (or any possible conflicting device) is not transferring data, control is passed to the instruction following the Waitr. Otherwise, IOX transfers program control to the busy return address. If it is desired to continuously test for completion of data transfer on the device, the busy return address of the immediately preceding IOT instruction can be specified, effecting a Wait loop.

If a slot is init'd to any device other than the LSR, control is returned to the calling program about 150 microseconds after execution of a Waitr. For the LSR, however, the time is about 100 milliseconds.

Note that a not-busy return from Waitr normally means the device is available. However, in the case of a Write, this only means that the last character has been output to the device. The device is still in the process of printing or punching the character. Thus, care must be exercised when performing an IOX Reset, hardware RESET, or HALT after a Write-Waitr sequence, since these may prevent the last character from being physically output.

7.5.5 **Waitr vs. Testing the Buffer Done Bit**

Since IOX permits you to have device-independent code, it may not be known, from run to run, what devices will be assigned to the slots in your program. Waitr tests the status, not only of the device it specifies, but also of all possible conflicting devices.

This means that when Waitr indicates that the device is not busy, the data transfer on the device of interest may have been done for some time. Depending on the program and what devices are assigned to the slots for a given run, the Waitr could have been waiting an additional amount of time for a conflicting device to become free.

Where this possibility exists and buffer availability is what is of interest, testing the Done bit of the Status byte (set when buffer transfer is complete) would be preferable to Waitr; whereas Waitr would be preferable if device availability is what is of interest.

This distinction is made in order to write device-independent code. In the example below:

1. If the devices at slots 2 and 3 could be guaranteed always to be conflicting, neither Waitr nor testing the Done bit would be necessary, because IOX would automatically wait for the busy device to finish before allowing the other device to begin.

2. If these devices could be guaranteed never to be conflicting, it wouldn't matter which of these methods was used, because Waitr couldn't be waiting extra time for a conflicting device (of no interest) to become free.
Example:

<table>
<thead>
<tr>
<th>PROGRAM A</th>
<th>PROGRAM B</th>
</tr>
</thead>
<tbody>
<tr>
<td>IOT</td>
<td>IOT</td>
</tr>
<tr>
<td>.WORD BUF2</td>
<td>.WORD BUF2</td>
</tr>
<tr>
<td>.BYTE READ, SLOT2</td>
<td>.BYTE READ, SLOT2</td>
</tr>
<tr>
<td></td>
<td>IOT</td>
</tr>
<tr>
<td>.WORD BUF1</td>
<td>.WORD BUF1</td>
</tr>
<tr>
<td>.BYTE READ, SLOT2</td>
<td>.BYTE READ, SLOT2</td>
</tr>
<tr>
<td></td>
<td>IOT</td>
</tr>
<tr>
<td>.WORD BUF2</td>
<td>.WORD BUF2</td>
</tr>
<tr>
<td>.BYTE WRITE, SLOT3</td>
<td>.BYTE WRITE, SLOT3</td>
</tr>
<tr>
<td>DUNTST:</td>
<td>DUNTST:</td>
</tr>
<tr>
<td>TSTB BUF1+3</td>
<td>IOT</td>
</tr>
<tr>
<td>DEVST:</td>
<td>.WORD DEVST</td>
</tr>
<tr>
<td>BPL DUNTST</td>
<td>.BYTE WAITR, SLOT2</td>
</tr>
<tr>
<td></td>
<td>IOT</td>
</tr>
<tr>
<td>.WORD SLOT2DEV</td>
<td>.BYTE INIT, SLOT4</td>
</tr>
</tbody>
</table>

Programs A and B do two successive reads from the same device into two different buffers. Since the devices are the same, IOX waits for the first read to finish before allowing the second to begin.

In Program A, we wish to process buffer 1. To have issued a Waitr for the device associated with slot 2 could have meant waiting also for the device at slot 3 if that device were in conflict. Hence, testing the Done bit in the buffer header is the proper choice.

In Program B, we wish control of the device at slot 2, so that it can be assigned to another slot and so we must know its availability. Therefore, Waitr is appropriate.

7.5.6 Single Buffer Transfer on One Device

A: IOT                $TRAP TO IOX
   .WORD BUF1         $SPECIFY BUFFER
   .BYTE READ, SLOT3  $READ FROM DEVICE AT
                        $SLOT 3 INTO BUFFER

BUSY: IOT              $TRAP TO IOX
       .WORD BUSY       $SPECIFY BUSY RETURN ADDRESS
       .BYTE WAITR, SLOT3 $WAIT FOR DEVICE AT SLOT
                            $3 TO FINISH READING

JMP A

The program segment above includes a Waitr which goes to a Busy Return address that is its own IOT -- continuously testing the device at slot 3 for availability. In this instance, involving only a single device and a single buffer, a Done condition in the Buffer 1 Status byte can be inferred from the availability of the device at slot 3. This knowledge assures us that all data requested for Buffer 1 is available for processing.

Testing the Done Bit of Buffer 1 might have been used instead, but was not necessary with only one device operating. Moreover, a Waitr, unlike a Done Bit test, would detect a timeout on the LSR if that device happened to be associated with slot 3.
7.5.7  Double Buffering

IOT
WORD BUF1
BYTE READ, SLOT3

$TRAP TO IOX
$SPECIFY BUFFER 1
$READ FROM DEVICE AT
$SLOT 3 INTO BUFFER 1

A:
IOT
WORD BUF2
BYTE READ, SLOT3

$TRAP TO IOX
$SPECIFY BUFFER 2
READ FROM DEVICE AT SLOT
$3 INTO BUFFER 2

(BUFFER BUF1 concurrent with Read into BUF2)

B:
IOT
WORD BUF1
BYTE READ, SLOT3

$TRAP TO IOX
$SPECIFY BUFFER 1
$READ FROM DEVICE AT
$SLOT 3 INTO BUFFER 1

(BUFFER BUF2 concurrent with Read into BUF1)

JMP A

The example above illustrates a time-saving double-buffer scheme whereby data is processed in Buffer 1 at the same time as new data is being read into Buffer 2; and, sequentially, data is processed in Buffer 2 at the same time as new data is being read into Buffer 1.

Because IOX ensures that the requested device is free before initiating the command, the subsequent return of control from the IOT at A implies that the read prior to A is complete; that is, that buffer 1 is available for processing. Similarly, the return of control from the IOT at B implies that buffer 2 is available. Waitr's are not required because IOX has automatically ensured the device's availability before initiating each Read.

7.5.8  Readr (Real-time Read)

IOT
.WORD (address of first word of the buffer header)
.BYTE l3, (slot number)
.WORD (done-address)

The Readr command functions as the Read except that upon completion of the data transfer, program control goes to the specified Done-address at the priority level of the device. Readr is used when you wish to execute a segment of your program immediately upon completing the data transfer. IOX goes to the Done address by executing a JSR R7, Done-address.

The general registers, which were saved when the last character interrupt occurred, are on the SP stack in the order indicated below:

(SP)↓
Return address to IOX
R5
R4
R3
R2
R1
R0

7-15
INPUT/OUTPUT PROGRAMMING

Return to IOX is accomplished by an RTS R7 instruction. IOX will then restore all registers and return to the interrupted program. Care should be taken in initiating another data transfer if the specified device can conflict with device requests at other priority levels. Waitr cannot be used to resolve conflict situations between priority levels.

7.5.9  Writr (Real-time Write)

IOT
.WORD (address of first word of the buffer header)
.BYTE 14, (slot number of device)
.WORD (done address)

The Writr command functions as the Write except that, upon completion of the data transfer, program control goes to the specified Done-address at the priority level of the device. IOX goes to the Done-address by executing a JSR R7, Done-address. The condition of the general registers and the return to IOX are the same as for Readr. Writr is used when you wish to execute a segment of your program immediately upon completing the data transfer.

As in the Readr, care should be taken in initiating another data transfer if the specified device can conflict with device requests at the priority level of the calling program.

7.6  REENABLING THE READER AND RESTARTING

7.6.1  Seek

IOT
.WORD 0
.BYTE 5, (slot number of LSR or HSR)

The Seek command clears IOX's internal End-Of-Medium (EOM) indicator on the LSR or HSR, making possible a subsequent read on those devices. With no EOM, an EOF cannot occur. The device associated with the specified slot remains Initied.

7.6.2  Restart

IOT
.WORD (address to restart)
.BYTE 3, 0

This command designates an address at which to restart your program. After this command has been issued, typing CTRL/P on the KED will transfer program control to the restart address, providing there is no LSR read in progress. In such a case, the LSR must be turned off (causing a timeout) before typing a CTRL/P. If the Restart address is designated as 0, the CTRL/P Restart capability is disabled.

The Restart command does not cancel any I/O in progress. It is the program's responsibility in its restart routine to clean up any I/O by executing a RESET command and ensuring that the stack pointer is reset.
7.7 FATAL ERRORS

Fatal errors result in program termination and a jump to location 408 (loaded with a HALT by IOX), with R0 set to the error code and R1 set as follows:

If the fatal error was due to an illegal memory reference (code 0), R1 will contain the PC at the time of the error.

If the fatal error was due to an error coded in the range 1-5, R1 will point to some element in the IOT argument list or to the instruction following the argument list, depending on whether IOX has finished decoding the arguments when it detects the error.

<table>
<thead>
<tr>
<th>Fatal Error Code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Illegal Memory Reference, SP overflow, illegal instruction</td>
</tr>
<tr>
<td>1</td>
<td>Illegal IOX command</td>
</tr>
<tr>
<td>2</td>
<td>Slot out of range</td>
</tr>
<tr>
<td>3</td>
<td>Device out of range</td>
</tr>
<tr>
<td>4</td>
<td>Slot not init ed</td>
</tr>
<tr>
<td>5</td>
<td>Illegal data mode</td>
</tr>
</tbody>
</table>

Note that the SP stack contains the value of the registers at the time of the error, namely

(SP) → R5
     R4
     R3
     R2
     R1
     R0
     PC
     Processor Status (PS)

(See Section 7.3.3.1 for a discussion of non-fatal errors.)

7.8 EXAMPLE OF PROGRAM USING IOX

This program is used to duplicate paper tape. Note that it could be altered by changing the device code at RDEV or PDEV. For instance, the program could easily be made to list a tape.

R0=0
R1=1
R2=2
R3=3
R4=4
R5=5
KSLAT=0
TSLOT=1
RSLOT=3
PSLOT=4
RESET=2
RESTRT=3
INIT=1
INPUT/OUTPUT PROGRAMMING

WAITR=4
READ=11
WRITE=12
EDF=20000
CR=15
LF=12

\$CR ASSIGNED ASCII CODE FOR CARRIAGE RETURN
\$LF ASSIGNED ASCII CODE FOR LINE FEED

\$CANNED MESSAGE
\$FORMATTED ASCII

\$BYTE COUNT

\$BUFFER SIZE
\$FORMATTED ASCII MODE
\$BC
\$CR LF

\$DEVICE CODE FOR HSR
\$DEVICE CODE FOR HSP

\$BUFFER SIZE
\$CODE FOR UNFORMATTED BINARY
\$SPECIFIES NUMBER OF BYTES FOR TRANSFER
\$RESERVES STORAGE FOR DATA

\$BUFFER SIZE
\$CODE FOR UNFORMATTED BINARY
\$SPECIFIES NUMBER OF BYTES FOR TRANSFER
\$RESERVES STORAGE FOR DATA

\$SPECIFY ADDRESS FOR BOTTOM OF STACK

\$INITIALIZATION

\$BEGIN* SPECIFIED AS RESTART
\$ADDRESS FOR CTRL P
\$SET UP INITIAL BC ON BUF1
\$SET UP INITIAL BC ON BUF2

\$TYPE OUT DIRECTIONS

\$READ A CR\$LF

\$WAIT FOR HIM TO TYPE A CARRIAGE RETURN, LINE FEED

\$INIT READER
\$INIT PUNCH

7-18
INPUT/OUTPUT PROGRAMMING

IOT
BUF1
\BYTE READ,RSLOT

LOOP: IOT
BUF2
\BYTE READ,RSLOT

BIT   #EOF Buf1+2
BNE   BEGIN
\END OF FILE?
\YES
\NO

IOT
BUF1
\BYTE WRITE,P SLOT

C: IOT
C
\BYTE WAIT,R SLOT

IOT
BUF1
\BYTE READ,RSLOT

BIT   #EOF,Buf2+2
BNE   BEGIN
\END OF FILE?

IOT
BUF2
\BYTE WRITE,P SLOT

D: IOT
B
\BYTE WAIT,R SLOT
BR LOOP
\END BEGIN

7.9 IOX INTERNAL INFORMATION

7.9.1 Conflict Byte/Word

The IOX Conflict byte (in IOXLPT, Conflict Word) contains the status (busy or free) of all devices as well as whether or not an echo is in progress. Bit 0 is the echo bit, bits 1-6 (and 8 in IOXLPT) refer to the corresponding codes for devices:

If Bit is Set

Bit 0 = Echo in progress

Bit Device 1 = KBD busy

Bit Device 2 = TTY busy

Bit Device 3 = LSR busy

7-19
INPUT/OUTPUT PROGRAMMING

Bit Device  |  4 = LSP busy
Bit Device  |  5 = HSR busy
Bit Device  |  6 = HSP busy
Bit Device  |  8 = LPT busy

In IOXLPT, the Conflict Byte is expanded to a word in order to accommodate the line printer, there being no bit 8 to correspond with that device's code of 10 (octal) (the lowest available code for an output device — see Section 7.9.5.1).

<table>
<thead>
<tr>
<th>Device</th>
<th>All Possible Conflicting Devices</th>
<th>Conflict Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>KBD</td>
<td>Echo, KBD, TTY, LSR, LSP</td>
<td>37</td>
</tr>
<tr>
<td>TTY</td>
<td>Echo, KBD, TTY, LSP</td>
<td>27</td>
</tr>
<tr>
<td>LSR</td>
<td>KBD, LSR</td>
<td>12</td>
</tr>
<tr>
<td>LSP</td>
<td>Echo, KBD, TTY, LSP</td>
<td>27</td>
</tr>
<tr>
<td>HSR</td>
<td>HSR</td>
<td>40</td>
</tr>
<tr>
<td>HSP</td>
<td>HSP</td>
<td>100</td>
</tr>
<tr>
<td>LPT</td>
<td>LPT</td>
<td>400</td>
</tr>
</tbody>
</table>

For each of the devices in the left hand column, all the possible conflicts are listed along with their respective conflict numbers. These numbers, representing bit patterns of the devices listed in column two above, are used to resolve any conflicting requests for devices. The appropriate number is masked with the conflict byte. If the result is zero, there are no conflicts and the device being tested has its bit set allowing data transfer to begin.

7.9.2 Device Interrupt Table (DIT)

Each device interrupt handler has associated with it a Device Interrupt Table (DIT) containing information that the handler needs:

<table>
<thead>
<tr>
<th>DIT</th>
<th>Checksum</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIT+2</td>
<td>Byte size from buffer header</td>
</tr>
<tr>
<td>DIT+4</td>
<td>Address of Mode byte in buffer header</td>
</tr>
<tr>
<td>DIT+6</td>
<td>Byte Location Pointer</td>
</tr>
<tr>
<td>DIT+10</td>
<td>Byte Count</td>
</tr>
<tr>
<td>DIT+12</td>
<td>Device code</td>
</tr>
</tbody>
</table>
INPUT/OUTPUT PROGRAMMING

DIT+14   Real time done-address
DIT+16   Address of device's data buffer register

The device interrupt routines gain access to the proper data by means of the DIT entry. When a transfer is complete, they set the appropriate bits in the buffer header pointed to by the DIT contents.

7.9.3  Device Status Table (DST)

The Device Status Table (DST) is used by IOX to check for EOF conditions. This table contains a word for each device indicating an EOM condition with a 1. When an EOM condition is recognized on input, IOX not only sets the appropriate bit in the buffer status byte associated with the data transfer, it also records this occurrence in the DST. When a data transfer command is given, IOX checks the DST for the EOM condition. If the appropriate word has a value of 1, IOX sets EOF in the Status byte of the current-command buffer. Since EOF is only possible for the LSR (code 3), and HSR (code 5), the words corresponding to those devices are the only ones that can ever be set to 1.

7.9.4  Teletype Hardware Tab Facility

If the Teletype model has a hardware tab facility, teleprinter output can be speeded up by:

1. For IOX, deleting the code from I.TTYCK+6 through I.TAB3+3.
2. For IOXLPT, skipping the code from I.IOFL through I.TAB3+3 (for the teleprinter only - not the line printer).

7.9.5  Adding Devices To IOX

In order to add a device to IOX the following tasks must be done:

1. Assign a legal code to the device
2. Modify the IOX tables
3. Provide an interrupt routine to handle data for the device.

The line printer (in IOXLPT) will be used as an example throughout this discussion.

7.9.5.1  Device Codes - The numbers from 7 to 17 (octal) are available for new-device codes, with the exception of 10 (octal) in the IOXLPT version. This code has been assigned to the line printer. The device code must be odd for an input device and even for an output device. This is so a check can be made for command/device correspondence; i.e., for a Read from an input device or a Write to an output device.

If the newest device was assigned a number that is higher than the codes of all the other devices, I.MAXDEV must be redefined to that value. This is so an out-of-range device specification in an Init command can be detected. In IOXLPT, I.MAXDEV=10.
INPUT/OUTPUT PROGRAMMING

Since each device code functions as an index in several word tables, the entries relating to a given device must be placed at the same relative position in each appropriate table. That is, the code number must indicate how many words into the table the entry for that device will be found. This, of course, means accounting for any unused space preceding the entry, if the codes are not assigned in strict sequence. Table entries for the line printer are found at the 10th (octal) word past the table tag, i.e., at Table+20.

7.9.5.2 Table Modification -

1. I.FUNC - Each entry is the octal value of the bit pattern in the device Control/Status Register that enables the corresponding device and/or any interrupt facility it has. Bit setting this number into the device's Control/Status register turns the device on; bit clearing turns it off. Determine this value for the device to be added, and place the entry in the appropriate device position in the table. For example, the line printer Control/Status Register has an Interrupt Enable facility in bit 6. This pattern of 100 is the EPT entry, and is located at I.FUNC+20.

2. I.SCRTAB - This table contains the addresses of the device Control/Status registers. The line printer entry I.LPTSCR has the value 177514, and is located at I.SCRTAB+20.

3. I.DST - (Refer to Section 7.9.3.) Create an entry of 0 for the device in the proper table location. Inserting a word of 0 at I.DST+20 created a device status entry for the line printer.

4. I.CONSIT - An entry in this table is used to set or clear a device's busy/free bit in the Conflict Byte (Conflict Word in IOXLPT). (See Section 7.9.1, and 5. below.) Each value is obtained by setting one bit only - the bit number corresponding to the device number. The line printer, being device 10 (octal), has a value of 400 (octal) (bit 10 set) and is located at I.CONSIT+20.

In the IOX version without the line printer, entries to this table are found in the high-order bytes of Table I.CONFLC. One more input device entry can be added to it. In IOXLPT, however, I.CONSIT is a separate word table, allowing eight more devices (four input and four output) to be added. Byte operations in the IOX I.CONSIT became word operations in IOXLPT to adapt to this expansion.

5. I.CONFLC - (Refer to Section 7.9.1 on Conflict Byte/Word.) Entries are bit patterns of conflicting devices. Since the line printer can only conflict with itself, the I.CONFLC entry is equal to the I.CONSIT entry. As in the I.CONSIT table, byte operations were changed to word operations for I.CONFLC in IOXLPT.

6. Create a DIT for the device (refer to Section 7.9.2) by assigning a DIT label and seven words of 0. If it is an output device, the address of the Device Buffer Register must be added as an eighth word.
INPUT/OUTPUT PROGRAMMING

7. I.INTAB – This is a table of DIT addresses. Place the label of the DIT (mentioned in 6. above) in the correct position in the table. I.INTAB+20 contains the line printer entry I.LPTDIT.

7.9.5.3 Interrupt Routines – Write (and assign a label to) an interrupt routine for the device to:

1. Get a character
2. Check for errors by means of the device Control/Status register
3. Do character interpretation according to the device and mode
4. Get a character in or out of the buffer
5. Update IOX's Byte Count
6. Compare IOX's Byte Count to User's Byte Count and Buffer size specification
7. Return for next character

Place the label of the interrupt routine at the address of the device vector, and follow it with the value of the interrupt priority in bits 7, 6, and 5. I.LPTIR, the address of the line printer interrupt routine, is at location 200. Location 202 contains the value 200 (indicating priority level 4).

If the device to be added is similar to the other single-character devices, steps 3-7 above can be performed by IOX as indicated below:

There are two routines, I.INPUT and I.OUTPUT, that are called from the interrupt routines. These routines mainly perform common functions for input and output devices. They are called as follows:

    JSR R5,I.INPUT and JSR R5,I.OUTPUT

At the location following one of these calls is the DIT for the proper device. The routine is thus able to use R5 to reference the DIT entries.

I.INPUT and I.OUTPUT also contain device-dependent code to perform functions such as tab counters for the teleprinter and line printer, and deletion of carriage returns in Formatted ASCII mode for the line printer. The device index value is used to identify the device. For the line printer, a symbol I.LPT, has been assigned the value 20 for convenient reference to the device index.
CHAPTER 8
FLOATING POINT MATH PACKAGE OVERVIEW

The new Floating-Point Math Package, FPMP-11, is designed to bring the 2/4 word floating point format of the FORTRAN environment to the paper tape software system of the PDP-11. The numerical routines in FPMP-11 are the same as those of the DOS/BATCH FORTRAN Operating Time System (OTS). TRAP and error handlers have been included to aid in interfacing with the FORTRAN routines.

FPMP-11 provides an easy means of performing basic arithmetic operations such as add, subtract, multiply, divide, and compare. It also provides transcendental functions (SIN, COS, etc.), type conversions (integer to floating-point, 2-word to 4-word, etc.), and ASCII conversions (ASCII to 2-word floating-point, etc.).

Floating-point notation is particularly useful for computations involving numerous multiply and divide operations where operand magnitudes may vary widely. FPMP-11 stores very large and very small numbers by saving only the significant digits and computing an exponent to account for leading and trailing zeros.

To conserve core space in a small system, FPMP-11 can be tailored to include only those routines needed to run a particular user program.

For more information on FPMP-11, refer to the FPMP-11 User's Manual (DEC-11-NFPMA-A-D) and to Appendix H of this manual.

FPMP is not currently available for the LSI-11 (PDP-11/03).
CHAPTER 9

PROGRAMMING TECHNIQUES

This chapter presents various programming techniques. They can be used to enhance your programming and to make optimum use of the PDP-11 processor. The reader is expected to be familiar with the PAL-11 assembly language (Chapters 1 & 2).

9.1 WRITING POSITION INDEPENDENT CODE

When a standard program is available for different users, it often becomes useful to be able to load the program into different areas of core and to run it there. There are several ways to do this:

1. Reassemble the program at the desired location.
2. Use a relocating loader which accepts specially coded binary from the assembler.
3. Have the program relocate itself after it is loaded.
4. Write code which is position independent.

On small machines, reassembly is often performed. When the required core is available, a relocating loader (usually called a linking loader) is preferable. It generally is not economical to have a program relocate itself since hundreds or thousands of addresses may need adjustment. Writing position independent code is usually not possible because of the structure of the addressing of the object machine. However, on the PDP-11, position independent code (PIC) is possible.

PIC is achieved on the PDP-11 by using addressing modes which form an effective memory address relative to the Program Counter (PC). Thus, if an instruction and its object(s) are moved in such a way that the relative distance between them is not altered, the same offset relative to the PC can be used in all positions in memory. Thus, PIC usually references locations relative to the current location. PIC may make absolute references as long as the locations referenced stay in the same place while the PIC is relocated. For example, references to interrupt and trap vectors are absolute, as are references to device registers in the external page and direct references to the general registers.
9.1.1 Position Independent Modes

There are three position independent modes or forms of instructions. They are:

1. Branches -- the conditional branches, as well as the unconditional branch, BR, are position independent since the branch address is computed as an offset to the PC.

2. Relative Memory References -- any relative memory reference of the form

   CLR X
   MOV X,Y
   JMP X

   is position independent because the assembler assembles it as an offset indexed by the PC. The offset is the difference between the referenced location and the PC. For example, assume the instruction CLR 200 is at address 100:

   100/ 005067 \#FIRST WORD OF CLR 200
   102/ 000074 \#OFFSET = 200-104

   The offset is added to the PC. The PC contains 104, i.e., the address of the word following the offset.

Although the form CLR X is position independent, the form CLR @X is not. Consider the following:

   S:  CLR @X \#CLEAR LOCATION A
   .
   .

   X:  .WORD A \#POINTER TO A
   .
   .

   A:  .WORD 0

   The contents of location X are used as the address of the operand in the location labeled A. Thus, if all of the code is relocated, the contents of location X must be altered to reflect the new address of A. If A, however, was the name associated with some fixed location (e.g., trap vector, device register), then statements S and X would be relocated and A would remain fixed. Thus, the following code is position independent.

   A = 36 \#ADDRESS OF SECOND WORD OF
   \#TRAP VECTOR
   S:  CLR @X \#CLEAR LOCATION A
   .
   .

   X:  .WORD A \#POINTER TO A

3. Immediate Operands -- The assembler addressing form #$X specifies immediate data, that is, the operand is in the instruction. Immediate data is position independent since it is a part of the instruction and is moved with the
programming techniques

instruction. Immediate data is fetched using the PC in the autoincrement mode.

As with direct memory references, the addressing form @#X is
not position independent. As before, the final effective
address is absolute and points to a fixed location not
relative to the PC.

9.1.2 Absolute Modes

Any time a memory location or register is used as a pointer to data,
the reference is absolute. If the referenced data is fixed in memory,
independent of the position of the PIC (e.g., trap-interrupt vectors,
device registers), the absolute modes must be used. 1 If the data is
relative to the PIC, the absolute modes must not be used unless the
pointers involved are modified. The absolute modes are:

@X                  Location X is a pointer
@#X                 The immediate word is a pointer
(R)                 The register is a pointer
(R)+ and -(R)       The register is a pointer
@(R)+ and @-(R)    The register points to a pointer
X(R)                R#6 or 7
                    The base, X, modified by (R) is
                    the address of the operand
@X(R)               The base, modified by (R), is a
                    pointer

The non-deferred index modes and stack operations require a little
clarification. As described in Sections 3.6.10 and 9.1.1, the form
X(7) is the normal mode to reference memory and is a relative mode.
Index mode, using a stack pointer (SP or other register) is also a
relative mode and may be used conveniently in PIC. Basically, the
stack pointer points to a dynamic storage area and index mode is used
to access data relative to the pointer. The stack pointer may be
initially set up by a position independent program as shown in Section
9.1.4.1. In any case, once the pointer is set up, all data on the
stack is referenced relative to the pointer. It should also be noted
that since the form 0 (SP) is considered a relative mode so is its
equivalent @SP. In addition, the forms (SP)+ and -(SP) are required
for stack pops and pushes.

9.1.3 Writing Automatic PIC

Automatic PIC is code which requires no alteration of addresses or
pointers. Thus, memory references are limited to relative modes
unless the location referenced is fixed (trap-interrupt vectors,
etc.). In addition to the above rules, the following must be
observed:

1. Start the program with .=0 to allow easy relocation using the
   Absolute Loader (see Chapter 6).

1When PIC is not being written, references to fixed locations may be
performed with either the absolute or relative forms.

9-3
PROGRAMMING TECHNIQUES

2. All location setting statements must be of the form .+=.tX or .+= function of tags within the PIC. For example, .+=A+10 where A is a local label.

3. There must not be any absolute location setting statements. This means that a block of PIC cannot set up trap and/or interrupt vectors at load time with statements such as:

```
.+34
.WORD TRAPH,340 ;TRAP VECTOR
```

The Absolute Loader, when it is relocating PIC, relocates all data by the load bias (see Chapter 6). Thus, the data for the vector would be relocated to some other place. Vectors may be set at execution time (see Section 9.1.4).

9.1.4 Writing Non-Automatic PIC

Often it is not possible or economical to write totally automated PIC. In these cases, some relocation may be easily performed at execution time. Some of the required methods of solution are presented below. Basically, the methods operate by examining the PC to determine where the PIC is actually located. Then a relocation factor can be easily computed. In all examples, it is assumed that the code is assembled at zero and has been relocated somewhere else by the Absolute Loader.

9.1.4.1 Setting Up The Stack Pointer - Often the first task of a program is to set the stack pointer (SP). This may be done as follows:

```
.+0 ;BEG IS THE FIRST INSTRUCTION OF
;THE PROGRAM
BEG: MOV PC,SP  ;SF=ADR BEG+2
TST -(SP) ;DECREMENT SP BY 2.
;A PUSH ONTO THE STACK WILL STORE
;THE DATA AT BEG-2.
```

9.1.4.2 Setting Up A Trap or Interrupt Vector - Assume the first word of the vector is to point to location INT which is in PIC.

```
X: MOV PC,R0 ;RO = ADR X+2
ADD #INT-X-2,R0 ;ADD OFFSET
MOV R0,@VECTOR ;MOVE POINTER TO VECTOR
```

The offset INT-X-2 is equivalent to INT-(X+2) and X+2 is the value of the PC moved by statement X. If PC is the PC that was assumed for the program when loaded at 0, and if PC is the current real PC, then the calculation is:

```
INT-PC0+PCn=INT+(PCn-PC0)
```

Thus, the relocation factor, PCn-PC0, is added to the assembled value of INT to produce the relocated value of INT.

9-4
9.1.4.3 Relocating Pointers - If pointers must be used, they may be relocated as shown above. For example, assume a list of data is to be accessed with the instruction

```
ADD (R0)+,R1
```

The pointer to the list, list L, may be calculated at execution time as follows:

```
M:
MOV PC,R0  $GET CURRENT PC
ADD $L-M-2,R0 $ADD OFFSET
```

Another variation is to gather all pointers into a table. The relocation factor may be calculated once and then applied to all pointers in the table in a loop.

```
X:
MOV PC,R0  $RELOCATE ALL ENTRIES IN PTRTBL
SUB $x+2,R0 $CALCULATE RELOCATION FACTOR
MOV #PTRTBL,R1 $GET AND RELOCATE A POINTER
ADD R0,R1  $ TO PTRTBL
MOV #TBLLEN,R2 $GET LENGTH OF TABLE
LOOP:
ADD R0,(R1)+ $RELOCATE AN ENTRY
DEC R2     $COUNT
BGE LOOP    $BRANCH IF NOT DONE
```

Care must be exercised when restarting a program which relocates a table of pointers. The restart procedure must not include the relocating again, i.e., the table must be relocated exactly once after each load.

9.2 LOADING UNUSED TRAP VECTORS

One of the features of the PDP-11 is the ability to trap on various conditions such as illegal instructions, reserved instructions, power failure, etc. However, if the trap vectors are not loaded with meaningful information, the occurrence of any of these traps will cause unpredictable results. By loading the vectors as indicated below, it is possible to avoid these problems as well as gain meaningful information about any unexpected traps that occur. This technique, which makes it easy to identify the source of a trap, is to load each unused trap vector with:

```
.trap address
.WORD .+2,HALT
```

This will load the first word of the vector with the address of the second word of the vector (which contains a HALT). Thus, for example, a halt at location 6 means that a trap through the vector at location 4 has occurred. The old PC and status may be examined by looking at the stack pointed to by register 6.

The trap vectors of interest are:

<table>
<thead>
<tr>
<th>Vector Location</th>
<th>Halt At Location</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>6</td>
<td>Bus Error; Illegal Instruction; Stack Overflow; Nonexistent Memory; Nonexistent Device; Word Referenced at Odd Address</td>
</tr>
<tr>
<td>10</td>
<td>12</td>
<td>Reserved Instruction</td>
</tr>
</tbody>
</table>
PROGRAMMING TECHNIQUES

14 16 Trace Trap Instruction (000003) or T-bit Set in Status Word (used by ODT)
20 22 IOT Executed (used by IOX)
24 26 Power Failure or Restoration
30 32 EMT Executed (used by FPP-11)
34 36 TRAP Executed

9.3 CODING TECHNIQUES

Because of the great flexibility in PDP-11 coding, time- and space-saving ways of performing operations may not be immediately apparent. Some comparisons follow.

9.3.1 Altering Register Contents

The techniques described in this section take advantage of the automatic stepping feature of autoincrement and autodecrement modes when used especially in TST and CMP instructions. These instructions do not alter operands. However, it is important to make note of the following:

- These alternative ways of altering register contents affect the condition codes differently.
- Register contents must be even when stepping by 2.
- These techniques work properly only if the registers are pointing to an existing memory location; otherwise, a trap is generated.

1. Adding 2 to a register might be accomplished by ADD #2,R0. However, this takes two words, whereas TST (R0)+ which also adds 2 to a register, takes only one word.

2. Subtracting 2 from a register can be done by the complementary instructions SUB #2,R0 or TST -(R0) with the same conditions as in adding 2.

3. This can be extended to adding or subtracting 2 from two different registers, or 4 from the same register, in one single-word instruction:

- CMP (R0)+,(R0)+  ADD 4 TO R0
- CMP -(R1),-(R1)  SUBTRACT 4 FROM R1
- CMP (R0)+,(R1)+(R0)+  ADD 2 TO R0, SUBTRACT 2 FROM R1
- CMP (R3)+,(R3)+  ADD 2 TO BOTH R3 AND R1
- CMP -(R3)+,(R1)+  ADD 2 TO BOTH R3 AND R0
4. Variations of the examples above can be employed if the instructions operate on bytes and one of the registers is the Stack Pointer. These examples depend on the fact that the Stack Pointer (as well as the PC) is always autoincremented or autodecremented by 2, whereas registers R0-R5 step by 1 in byte instructions.

\[
\begin{align*}
\text{CMPB (SP)+, (R3)+} & \quad \# \text{ADD 2 TO SP AND 1 TO R3} \\
\text{CMPB -(R3), -(SP)} & \quad \# \text{SUBTRACT 1 FROM R3 AND 2 FROM SP} \\
\text{CMPB (R3)+, -(SP)} & \quad \# \text{ADD 1 TO R3, SUBTRACT 2 FROM SP}
\end{align*}
\]

5. Popping an unwanted word off the processor stack (adding 2 to register 6) and testing another value can be two separate instructions or one combined instruction:

\[
\begin{align*}
\text{TST (SP)+} & \quad \# \text{POP WORD} \\
\text{TST COUNT} & \quad \# \text{SET CONDITION CODES FOR COUNT} \\
or\quad \\
\text{MOV COUNT, (SP)+} & \quad \# \text{POP WORD & SET CODES FOR COUNT}
\end{align*}
\]

The differences are that the TST instructions take three words and clear the Carry bit, and the MOV instruction takes two words and doesn't affect the Carry bit.

9.3.2 Subroutines

1. Condition codes set within a subroutine can be used to conditionally branch upon return to the calling program, since the RTS instruction does not affect condition codes.

\[
\begin{align*}
\text{JSR PC,X} & \quad \# \text{CALL SUBROUTINE X} \\
\text{BNE ABC} & \quad \# \text{BRANCH ON CONDITION SET} \\
\text{.} & \quad \# \text{IN SUBROUTINE X} \\
\text{.} & \quad \\
\text{X:} & \quad \# \text{SUBROUTINE ENTRY} \\
\text{.} & \quad \\
\text{CMP R2,DEF} & \quad \# \text{TEST CONDITION} \\
\text{RTS PC} & \quad \# \text{RETURN TO CALLING PROGRAM}
\end{align*}
\]

2. When a JSR first operand register is not the PC, data stored following a subroutine call can be accessed within the subroutine by referencing the register. (The register contains the return address.)

\[
\begin{align*}
\text{JSR R5,Y} & \quad \# \text{WORD HIGH} \\
\text{.} & \quad \# \text{WORD LOW} \\
\text{.} & \quad \# \text{LATEST R5 VALUE WILL POINT HERE} \\
\text{.} & \quad \\
\text{Y:} & \quad \text{MOV (R5)+, R2} \quad \# \text{VALUE OF HIGH ACCESS} \\
\text{.} & \quad \text{MOV (R5)+, R4} \quad \# \text{VALUE OF LOW ACCESS} \\
\text{.} & \quad \\
\text{.} & \quad \text{RTS R5} \quad \# \text{RETURN TO LOCATION} \\
\text{.} & \quad \# \text{CONTAINED IN R5}
\end{align*}
\]
Another possibility is:

```
JSR R5, SUB
BR PSTARG ; LOW-ORDER BYTE IS OFFSET TO RETURN
            ; ADDRESS, WHICH EQUALS NO. OF ARGS.
.WORD A ; ADDRESS OF ARG A
.WORD B ; ADDRESS OF ARG B
.WORD C ; ADDRESS OF ARG C
    ;
PSTARG: ; RETURN ADDRESS
    ;
SUB:   MOV B @ R5, COUNT ; GET NO. OF ARGs FROM LOW BYTE
       ; OF BR (IF DESIRED).
       MOV @14(R5), R2 ; E.G., GET 6TH ARGUMENT
       MOV @8(R5), R1 ; GET 3RD ARGUMENT
            ;
       RTS R5 ; RETURNS TO BRANCH WHICH JUMPS PAST
            ; ARG LIST TO REAL RETURN ADDRESS
```

In the example above, the branch instruction contributes two main advantages:

1. If R5 is unaltered when the RTS is executed, return will always be to the branch instruction. This ensures a return to the proper location even if the length of the argument list is shorter or longer than expected.

2. The operand of the branch, being an offset past the argument list, provides the number of arguments in the list.

Arguments can be made sharable by separating the data from the main code. This is easily accomplished by treating the JSR and its return as a subroutine itself:

```
CALL:  ;
       ; ARGLST:  JSR R5, SUB
       ; BR PSTARG
       JSR PC, ARGLST ; WORD A
       ;
       ;
```

3. The examples above all demonstrate the calling of subroutines from a non-reentrant program. The called subroutine can be either reentrant or non-reentrant in each case. The following example illustrates a method of also allowing calling programs to be reentrant. The arguments and linkage are first placed on the stack, simulating a JSR R5, SUB, so that arguments are accessed from the subroutine via X(R5). Return to the calling program is executed from the stack.
PROGRAMMING TECHNIQUES

CALL:

.
.
.

MOV R5,-(SP) ;SAVE R5 ON STACK.
MOV JSBR,-(SP) ;PUSH INSTRUCTION JSR R6,0R5 ON
               ;STACK. PUSH ADDRESSES OF ARGU-
               ;MENTS ON STACK IN REVERSE ORDER
               ;(SEE BELOW).
MOV BRN,-(SP) ;PUSH BRANCH INSTRUCTION ON STACK
X:     MOV SP,R5 ;MOVE ADDRESS OF BRANCH TO R5.
JSR FC,SUB ;CALL SUB AND SAVE RETURN ON STACK.
RET:    MOV (SP)+,R5 ;RESTORE OLD R5 UPON RETURN.
         ;DATA AREA OF PROGRAM.
.
.
JSBR:   JSR R6,0R5
BRN:    BR.+N+2 ;BRANCH PAST N WORD ARGUMENTS

The address of an argument can be pushed on the stack in several ways. Three are shown below.

a. The arguments A, B, and C are read-only constants which are in memory (not on the stack):

   MOV $C,-(SP) ;PUSH ADDRESS OF C
   MOV $B,-(SP) ;PUSH ADDRESS OF B
   MOV $A,-(SP) ;PUSH ADDRESS OF A

b. Arguments A, B, and C have their addresses on the stack at the Lth, Mth, and Nth bytes from the top of the stack.

   MOV N(SP),-(SP) ;PUSH ADDRESS OF C
   MOV M+2(SP),-(SP) ;PUSH ADDRESS OF B
   MOV L+4(SP),-(SP) ;PUSH ADDRESS OF A

   Note that the displacements from the top of the stack are adjusted by two for each previous push because the top of the stack is being moved on each push.

c. Arguments A, B, and C are on the stack at the Lth, Mth, and Nth bytes from the top but their addresses are not.

   MOV $N+2,-(SP) ;PUSH DISPLACEMENT TO ARGUMENT
   ADD SP,$SP
   MOV $M+4,-(SP) ;CALCULATE ACTUAL ADDRESS OF C
   ADD SP,$SP
   ADD SP,$SP
   MOV $L+6,-(SP) ;ADDRESS OF B
   ADD SP,$SP
   ADD SP,$SP
   ;ADDRESS OF A

9-9
PROGRAMMING TECHNIQUES

When subroutine SUB is entered, the stack appears as follows:

```
RET
BR. +N+N+2
A
B
!
JSR R6, @R5
OLD R5
```

Subroutine SUB returns by means of an RTS R5, which places R5 into the PC and pops the return address from the stack into R5. This causes the execution of the branch because R5 has been loaded (at location X) with the address of the branch. The JSR branched to then returns control to the calling program, and in so doing, moves the current PC value into the SP, thereby removing everything above the old R5 from the stack. Upon return at RET, this too is popped, restoring the original R5 and SP values.

4. The next example is a recursive subroutine (one that calls itself). Its function is to look for a matching right parenthesis for every left parenthesis encountered. The subroutine is called by JSR PC,A whenever a left parenthesis is encountered (R2 points to the character following it). When a right parenthesis is found, an RTS PC is executed, and if the right parenthesis is not the last legal one, another is searched for. When the final matching parenthesis is found, the RTS returns control to the main program.

```
A: MOVB (R2)+, R0   #GET SUCCESSIVE CHARACTERS.
    CMPB #', R0     #LOOK FOR LEFT PARENTHESIS.
    BNE B           #FOUND?
    JSR PC, A      #LEFT PAREN FOUND, CALL SEL.
    BR A           #GO LOOK AT NEXT CHARACTER

B: CMPB #', R0     #LEFT PAREN NOT FOUND, LOOK FOR
    BNE A           #RIGHT PAREN.
    RTS PC         #RETURN PAREN FOUND, IF NOT LAST,
                   #GO TO B. IF LAST, GO TO MAIN PROGRAM.
```

5. The example below illustrates the use of co-routines, called by JSR PC, @(SP)+. The program uses double buffering on both input and output, performing as follows:

```
Write 01 } concurrently   Write 02 } concurrently
Read I1 }    contemporarily  Read I2 }    concurrently
Process I2 }     Process I1 }
```

JSR PC, @(SP)+ always performs a jump to the address specified on top of the stack and replaces that address with the new return address. Each time the JSR at B is executed, it jumps to a different location; initially to A and thereafter to the location following the JSR executed prior to the one at B. All other JSR's jump to B+2.
PROGRAMMING TECHNIQUES

BEGIN:
(do I/O resets, inits, etc.)

IOT

;READ INTO I1 TO START PROCESS
;WORD I1
;BYTE READ, INSLOT
MOV #A, -(6)
;INITIALIZE STACK FOR FIRST JSR
B:
JSR PC,@(6)+
;DO I/O FOR 01 AND I1 OR 02 AND I2

;perform processing
BR B
;MORE I/O
;END OF MAIN LOOP
;I/O CO-ROUTINES
A:

IOT

;READ INTO I2
;WORD I2
;BYTE READ, INSLOT

;set parameters to process I1, 01

JSR PC,@(6)+
;RETURN TO PROCESS AT B+2
IOT
;WRITE FROM 01
;WORD 01
;BYTE WRITE, OUTSLOT
IOT
;READ INTO I1
;WORD I1
;BYTE READ, INSLOT

;set parameters to process I2, 02

JSR PC,@(6)+
;RETURN TO PROCESS AT B+2
IOT
;WRITE FROM 02
;WORD 02
;BYTE WRITE, OUTSLOT
BR A

6. The trap handler, below, simulates a two-word JSR instruction with a one-word TRAP instruction. In this example, all TRAP instructions in the program take an operand, and trap to the handler address at location 34. The table of subroutine addresses (e.g., A, B, ...) can be constructed as follows:

TABLE:
CALA=,-TABLE
;WORD A
;CALLED BY: TRAP CALA

CALB=,-TABLE
;WORD B
;CALLED BY: TRAP CALB

Another way to construct the table:

TABLE:
CALA=,-TABLE+TRAP
;WORD A
;CALLED BY: CALA

9-11
The TRAP handler for either of the above methods follows:

TRAP34: MOV @SP,2(SP) ; REPLACE STACKED PS WITH PC.
       SUB $2, @SP ; GET POINTER TO TRAP INSTRUCTION.
       MOV @SP++, -(SP) ; REPLACE ADDRESS OF TRAP WITH
                        ; TRAP INSTRUCTION ITSELF.
       ADD @(TABLE-TRAP),@SP ; CALCULATE SUBROUTINE ADDR.
       MOV @SP++, PC ; JUMP TO SUBROUTINE.

In the example above, if the third instruction had been written
MOV @(SP), (SP) it would have taken an extra word since @(SP) is in
Index Mode and assembles as @0(SP). In the final instruction, a jump
was executed by a MOV @(SP)+,PC because no equivalent JMP instruction
exists.

Following are some JMP and MOV equivalences (note that JMP does not
affect condition codes).

JMP (R4) = MOV R4,PC
JMP @(R4) = MOV (R4),PC
(2 words) (1 word)

none = MOV @(R4),PC
JMP -(R4) = none

JMP @(R4)+ = MOV (R4),PC
JMP -(R4) = MOV -(R4),PC

none = MOV @(R4)+,PC
none = MOV -(R4),PC

JMP X = MOV #X,PC
JMP @X = MOV X,PC

none = MOV @X,PC

The TRAP handler can be useful, also, as a patching technique.
Jumping out to a patch area is often difficult because a two-word jump
must be performed. However, the one-word TRAP instruction may be used
to dispatch to patch areas. A sufficient number of slots for patching
should first be reserved in the dispatch table of the TRAP handler.
The jump can then be accomplished by placing the address of the patch
area into the table and inserting the proper TRAP instruction where
the patch is to be made.

1Replacing the saved PS loses the T-bit status. If a breakpoint has
been set on the TRAP instruction, ODT will not gain control again to
reinsert the breakpoints because the T-bit trap will not occur.
### APPENDIX A

#### ASCII CHARACTER SET

<table>
<thead>
<tr>
<th>EVEN PARITY BIT</th>
<th>7-BIT OCTAL CODE</th>
<th>CHARACTER</th>
<th>REMARKS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>000</td>
<td>NUL</td>
<td>NULL, TAPE FEED, CONTROL SHIFT P.</td>
</tr>
<tr>
<td>1</td>
<td>001</td>
<td>SOH</td>
<td>START OF HEADING; ALSO SOM, START OF MESSAGE, CONTROL A,</td>
</tr>
<tr>
<td>1</td>
<td>002</td>
<td>STX</td>
<td>START OF TEXT; ALSO EOA, END OF ADDRESS, CONTROL B,</td>
</tr>
<tr>
<td>0</td>
<td>003</td>
<td>ETX</td>
<td>END OF TEXT: ALSO EOM, END OF MESSAGE CONTROL C,</td>
</tr>
<tr>
<td>1</td>
<td>004</td>
<td>EOT</td>
<td>END OF TRANSMISSION (END): SHUTS OFF TWX MACHINES, CONTROL D,</td>
</tr>
<tr>
<td>0</td>
<td>005</td>
<td>ENQ</td>
<td>ENQUIRY (ENQRY); ALSO WRU, CONTROL E,</td>
</tr>
<tr>
<td>0</td>
<td>006</td>
<td>ACK</td>
<td>ACKNOWLEDGE. ALSO RU, CONTROL F.</td>
</tr>
<tr>
<td>1</td>
<td>007</td>
<td>BEL</td>
<td>RINGS THE BELL. CONTROL G.</td>
</tr>
<tr>
<td>1</td>
<td>010</td>
<td>BS</td>
<td>BACKSPACE: ALSO FEO, FORMAT EFFECTOR. BACKSPACE SOME MACHINES, CONTROL H.</td>
</tr>
<tr>
<td>0</td>
<td>011</td>
<td>HT</td>
<td>HORIZONTAL TAB. CONTROL I.</td>
</tr>
<tr>
<td>0</td>
<td>012</td>
<td>LF</td>
<td>LINE FEED OR LINE SPACE (NEW LINE): ADVANCES PAPER TO NEXT LINE, DUPLICATED BY CONTROL J.</td>
</tr>
<tr>
<td>1</td>
<td>013</td>
<td>VT</td>
<td>VERTICAL TAB (VTAB). CONTROL K.</td>
</tr>
<tr>
<td>0</td>
<td>014</td>
<td>FF</td>
<td>FORM FEED TO TOP OF NEXT PAGE (PAGE). CONTROL L.</td>
</tr>
<tr>
<td>1</td>
<td>015</td>
<td>CR</td>
<td>CARRIAGE RETURN TO BEGINNING OF LINE. DUPLICATED BY CONTROL M.</td>
</tr>
<tr>
<td>1</td>
<td>016</td>
<td>SO</td>
<td>SHIFT OUT: CHANGES RIBBON COLOR TO RED. CONTROL N.</td>
</tr>
<tr>
<td>0</td>
<td>017</td>
<td>SI</td>
<td>SHIFT IN: CHANGES RIBBON COLOR TO BLACK. CONTROL O.</td>
</tr>
<tr>
<td>1</td>
<td>020</td>
<td>DLE</td>
<td>DATA LINK ESCAPE. CONTROL P (DC0).</td>
</tr>
<tr>
<td>0</td>
<td>021</td>
<td>DC1</td>
<td>DEVICE CONTROL 1,Turns TRANSMITTER (READER) ON, CONTROL Q (XON).</td>
</tr>
<tr>
<td>0</td>
<td>022</td>
<td>DC2</td>
<td>DEVICE CONTROL 2, TURNS PUNCH OR AUXILIARY ON. CONTROL R (TAPE, AUX ON).</td>
</tr>
<tr>
<td>1</td>
<td>023</td>
<td>DC3</td>
<td>DEVICE CONTROL e, TURNS TRANSMITTER (READER) OFF, CONTROL S (XOFF).</td>
</tr>
<tr>
<td>0</td>
<td>024</td>
<td>DC4</td>
<td>DEVICE CONTROL 4. TURNS PUNCH OR AUXILIARY OFF. CONTROL T (TAPE, AUX OFF).</td>
</tr>
<tr>
<td>1</td>
<td>025</td>
<td>NAK</td>
<td>NEGATIVE ACKNOWLEDGE: ALSO ERR. ERROR. CONTROL U.</td>
</tr>
<tr>
<td>1</td>
<td>026</td>
<td>SYN</td>
<td>SYNCHRONOUS IDLE (SYNC). CONTROL V.</td>
</tr>
<tr>
<td>0</td>
<td>027</td>
<td>ETB</td>
<td>END OF TRANSMISSION BLOCK: ALSO LEM. LOGICAL END OF MEDIUM. CONTROL W.</td>
</tr>
<tr>
<td>0</td>
<td>030</td>
<td>CAN</td>
<td>CANCEL (CANCEL). CONTROL X.</td>
</tr>
<tr>
<td>1</td>
<td>031</td>
<td>EM</td>
<td>END OF MEDIUM. CONTROL Y.</td>
</tr>
<tr>
<td>1</td>
<td>032</td>
<td>SUB</td>
<td>SUBSTITUTE. CONTROL Z.</td>
</tr>
<tr>
<td>0</td>
<td>033</td>
<td>ESC</td>
<td>ESCAPE. PREFIX.</td>
</tr>
</tbody>
</table>
### ASCII CHARACTER SET

<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>034</td>
<td>FS</td>
</tr>
<tr>
<td>035</td>
<td>GS</td>
</tr>
<tr>
<td>036</td>
<td>RS</td>
</tr>
<tr>
<td>037</td>
<td>US</td>
</tr>
<tr>
<td>040</td>
<td>SP</td>
</tr>
<tr>
<td>041</td>
<td>!</td>
</tr>
<tr>
<td>042</td>
<td>$</td>
</tr>
<tr>
<td>043</td>
<td>#</td>
</tr>
<tr>
<td>044</td>
<td>&amp;</td>
</tr>
<tr>
<td>045</td>
<td>(</td>
</tr>
<tr>
<td>046</td>
<td>)</td>
</tr>
<tr>
<td>047</td>
<td>+</td>
</tr>
<tr>
<td>050</td>
<td>,</td>
</tr>
<tr>
<td>051</td>
<td>.</td>
</tr>
<tr>
<td>052</td>
<td>/</td>
</tr>
<tr>
<td>053</td>
<td>0</td>
</tr>
<tr>
<td>054</td>
<td>1</td>
</tr>
<tr>
<td>055</td>
<td>2</td>
</tr>
<tr>
<td>056</td>
<td>3</td>
</tr>
<tr>
<td>057</td>
<td>4</td>
</tr>
<tr>
<td>058</td>
<td>5</td>
</tr>
<tr>
<td>059</td>
<td>6</td>
</tr>
<tr>
<td>060</td>
<td>7</td>
</tr>
<tr>
<td>061</td>
<td>8</td>
</tr>
<tr>
<td>062</td>
<td>9</td>
</tr>
<tr>
<td>063</td>
<td>;</td>
</tr>
<tr>
<td>064</td>
<td>&lt;</td>
</tr>
<tr>
<td>065</td>
<td>=</td>
</tr>
<tr>
<td>066</td>
<td>&gt;</td>
</tr>
<tr>
<td>067</td>
<td>?</td>
</tr>
<tr>
<td>068</td>
<td>@</td>
</tr>
<tr>
<td>069</td>
<td>A</td>
</tr>
<tr>
<td>070</td>
<td>B</td>
</tr>
<tr>
<td>071</td>
<td>C</td>
</tr>
<tr>
<td>072</td>
<td>D</td>
</tr>
<tr>
<td>073</td>
<td>E</td>
</tr>
<tr>
<td>074</td>
<td>F</td>
</tr>
<tr>
<td>075</td>
<td>G</td>
</tr>
<tr>
<td>076</td>
<td>H</td>
</tr>
<tr>
<td>077</td>
<td>I</td>
</tr>
<tr>
<td>078</td>
<td>J</td>
</tr>
<tr>
<td>079</td>
<td>K</td>
</tr>
<tr>
<td>080</td>
<td>L</td>
</tr>
<tr>
<td>081</td>
<td>M</td>
</tr>
<tr>
<td>082</td>
<td>N</td>
</tr>
<tr>
<td>083</td>
<td>O</td>
</tr>
<tr>
<td>084</td>
<td>P</td>
</tr>
<tr>
<td>085</td>
<td>Q</td>
</tr>
<tr>
<td>086</td>
<td>R</td>
</tr>
<tr>
<td>087</td>
<td>S</td>
</tr>
<tr>
<td>088</td>
<td>T</td>
</tr>
<tr>
<td>089</td>
<td>U</td>
</tr>
<tr>
<td>090</td>
<td>V</td>
</tr>
<tr>
<td>091</td>
<td>W</td>
</tr>
<tr>
<td>092</td>
<td>X</td>
</tr>
<tr>
<td>093</td>
<td>Y</td>
</tr>
<tr>
<td>094</td>
<td>Z</td>
</tr>
</tbody>
</table>

**SHIFT K**
<table>
<thead>
<tr>
<th>Code</th>
<th>Character</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>134</td>
<td>SHIFT L</td>
</tr>
<tr>
<td>1</td>
<td>135</td>
<td>SHIFT M</td>
</tr>
<tr>
<td>1</td>
<td>136</td>
<td>SHIFT N</td>
</tr>
<tr>
<td>0</td>
<td>137</td>
<td>+</td>
</tr>
<tr>
<td>0</td>
<td>140</td>
<td>+</td>
</tr>
</tbody>
</table>
| 0    | 175       | ACCENT GRAVE.
| 0    | 176       | THIS CODE GENERATED BY ALT MODE.
| 1    | 177       | THIS CODE GENERATED BY ESC KEY (IF PRESENT) DELETE, RUB OUT.
| 1    | 141       | a           |
| 1    | 142       | b           |
| 0    | 143       | c           |
| 1    | 144       | d           |
| 0    | 145       | e           |
| 0    | 146       | f           |
| 1    | 147       | g           |
| 1    | 150       | h           |
| 0    | 151       | i           |
| 0    | 152       | j           |
| 1    | 153       | k           |
| 0    | 154       | l           |
| 1    | 155       | m           |
| 1    | 156       | n           |
| 0    | 157       | o           |
| 1    | 160       | p           |
| 0    | 161       | q           |
| 0    | 162       | r           |
| 1    | 163       | s           |
| 0    | 164       | t           |
| 1    | 165       | u           |
| 1    | 166       | v           |
| 0    | 167       | w           |
| 0    | 170       | x           |
| 1    | 171       | y           |
| 1    | 172       | z           |
| 0    | 173       |              |
| 1    | 174       |              |

**Lower case alphabet follows (Teletype Model 37 only).**
APPENDIX B
PAL-11S ASSEMBLY LANGUAGE AND ASSEMBLER

B.1 TERMINATORS

The list below defines all characters which are considered to be terminators. The order of the list implies the descending hierarchy of significance.

<table>
<thead>
<tr>
<th>Character</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTRL/FORM</td>
<td>Source line terminator.</td>
</tr>
<tr>
<td>LINE FEED</td>
<td>Source line terminator.</td>
</tr>
<tr>
<td>RETURN</td>
<td>Source line terminator</td>
</tr>
<tr>
<td>:</td>
<td>Label terminator</td>
</tr>
<tr>
<td>=</td>
<td>Direct assignment delineator</td>
</tr>
<tr>
<td>%</td>
<td>Register term delineator</td>
</tr>
<tr>
<td>TAB</td>
<td>Item terminator</td>
</tr>
<tr>
<td></td>
<td>Field terminator</td>
</tr>
<tr>
<td>BLANK or</td>
<td>Item terminator</td>
</tr>
<tr>
<td>SPACE</td>
<td>Field terminator</td>
</tr>
<tr>
<td>#</td>
<td>Immediate expression field indicator</td>
</tr>
<tr>
<td>@</td>
<td>Deferred addressing indicator</td>
</tr>
<tr>
<td>(</td>
<td>Initial register field indicator</td>
</tr>
<tr>
<td>)</td>
<td>Terminal register field indicator</td>
</tr>
<tr>
<td>'</td>
<td>Operand field separator</td>
</tr>
<tr>
<td>;</td>
<td>Comments field delimiter</td>
</tr>
<tr>
<td>+</td>
<td>Arithmetic addition operator</td>
</tr>
<tr>
<td>-</td>
<td>Arithmetic subtraction operator</td>
</tr>
<tr>
<td>&amp;</td>
<td>Logical AND operator</td>
</tr>
<tr>
<td>!</td>
<td>Logical OR operator</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double ASCII text indicator</td>
</tr>
<tr>
<td>'</td>
<td>Single ASCII text indicator</td>
</tr>
</tbody>
</table>

B-1
B.2 ADDRESS MODE SYNTAX

r is an integer between 0 and 7.

R is a register expression, E is an expression, ER is either a register expression or an absolute expression in the range of 0 to 7.

<table>
<thead>
<tr>
<th>Address Mode Name</th>
<th>Address Mode Number</th>
<th>Symbol in Operand Field</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>0r</td>
<td>R</td>
<td>Register R contains the operand. R is a register expression.</td>
</tr>
<tr>
<td>Deferred Register</td>
<td>1r</td>
<td>@R or (R)</td>
<td>Register R contains the operand address.</td>
</tr>
<tr>
<td>(ER)+</td>
<td>2r</td>
<td>deferred</td>
<td>The contents of the register specified by ER is incremented after being used as the address of the operand.</td>
</tr>
<tr>
<td>Deferred Autoincrement</td>
<td>3r</td>
<td>@(ER)+</td>
<td>ER contains the pointer to the address of the operand. ER is incremented after use.</td>
</tr>
<tr>
<td>Autodecrement</td>
<td>4r</td>
<td>-(ER)</td>
<td>The contents of register ER is decremented before it is used as the address of the operand.</td>
</tr>
<tr>
<td>Deferred Autodecrement</td>
<td>5r</td>
<td>@-(ER)</td>
<td>The contents of register ER is decremented before it is used as the pointer to the address of the operand.</td>
</tr>
<tr>
<td>Index by the register Specified</td>
<td>6r</td>
<td>E(ER)</td>
<td>E plus the contents of the register specified, ER, is the address of the operand.</td>
</tr>
<tr>
<td>Deferred index by the register specified</td>
<td>7r</td>
<td>@E(ER)</td>
<td>E added to ER gives the pointer to the address of the operand.</td>
</tr>
<tr>
<td>Immediate Operand</td>
<td>27</td>
<td>#E</td>
<td>E is the operand.</td>
</tr>
<tr>
<td>Absolute address</td>
<td>37</td>
<td>@@E</td>
<td>E is the operand address.</td>
</tr>
<tr>
<td>Relative address</td>
<td>67</td>
<td>E</td>
<td>E is the address of the operand.</td>
</tr>
<tr>
<td>Deferred relative address</td>
<td>77</td>
<td>@E</td>
<td>E is the pointer to the address of the operand.</td>
</tr>
</tbody>
</table>

B.3 INSTRUCTIONS

The tables of instructions which follow are grouped according to the operands they take and according to the bit patterns of their op-codes.
In the representation of op-codes, the following symbols are used:

- **SS** Source operand specified by a 6-bit address mode
- **DD** Destination operand specified by a 6-bit address mode
- **XX** 8-bit offset to a location (branch instructions)
- **R** Integer between 0 and 7 representing a general register

Symbols used in the description of instruction operations are:

- **SE** Source effective address
- **DE** Destination effective address
- () contents of
- + becomes
- **PS** Processor Status word

The condition codes in the processor status word (PS) are affected by the instructions; these condition codes are represented as follows:

- **N** Negative bit: set if the result is negative
- **Z** Zero bit: set if the result is zero
- **V** Overflow bit: set if the result had an overflow
- **C** Carry bit: set if the result had a carry

In the representation of the instruction's effect on the condition codes, the following symbols are used:

- * Conditionally set
- - Not affected
- 0 Cleared
- 1 Set

To set conditionally means to use the instruction's result to determine the state of the code.

Logical operators are represented by the following symbols:

- ! Inclusive OR
- (Exclusive OR
- & AND
- - (used over a symbol) NOT (i.e., l's complement)
### B.3.1 Double Operand Instructions  \( \text{OP A,A} \)

<table>
<thead>
<tr>
<th>Op-code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
<th>Operation</th>
<th>Condition Codes</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>01ssdd</td>
<td>mov</td>
<td>move</td>
<td>((\text{SE})\rightarrow\text{DE})</td>
<td>* * 0 -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11ssdd</td>
<td>movb</td>
<td>move Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>02SSDD</td>
<td>CMP</td>
<td>CoMPare</td>
<td>((\text{SE})-(\text{DE}))</td>
<td>* * * *</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12SSDD</td>
<td>CMPB</td>
<td>CoMPare Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>03SSDD</td>
<td>BIT</td>
<td>BIT Test</td>
<td>((\text{SE})&amp;(\text{DE}))</td>
<td>* * 0 -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13SSDD</td>
<td>BITB</td>
<td>BIT Test Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>04SSDD</td>
<td>BIC</td>
<td>BIT Clear</td>
<td>((\text{SE}) &amp;(\text{DE})\rightarrow\text{DE})</td>
<td>* * 0 -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14SSDD</td>
<td>BICB</td>
<td>BIT Clear Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>05SSDD</td>
<td>BIS</td>
<td>BIT Set</td>
<td>((\text{SE})\rightarrow\text{DE})</td>
<td>* * 0 -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15SSDD</td>
<td>BISB</td>
<td>BIT Set Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>06SSDD</td>
<td>ADD</td>
<td>ADD</td>
<td>((\text{SE})\rightarrow\text{DE})</td>
<td>* * * *</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>16SSDD</td>
<td>SUB</td>
<td>SUBtract</td>
<td>((\text{DE})\rightarrow\text{DE})</td>
<td>* * * *</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### B.3.2 Single Operand Instructions  \( \text{OP A} \)

<table>
<thead>
<tr>
<th>Op-code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
<th>Operation</th>
<th>Condition Codes</th>
<th>N</th>
<th>Z</th>
<th>V</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>0050DD</td>
<td>CLR</td>
<td>CLeaR</td>
<td>(0\rightarrow\text{DE})</td>
<td>0 1 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1050DD</td>
<td>CLRB</td>
<td>CLeaR Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0051DD</td>
<td>COM</td>
<td>COMplement</td>
<td>((\text{DE})\rightarrow\text{DE})</td>
<td>* * 0 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1051DD</td>
<td>COMB</td>
<td>COMplement Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0052DD</td>
<td>INC</td>
<td>INCrement</td>
<td>((\text{DE})\rightarrow\text{DE})</td>
<td>* * * 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1052DD</td>
<td>INCB</td>
<td>INCrement Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0053DD</td>
<td>DEC</td>
<td>DECrement</td>
<td>((\text{DE})\rightarrow\text{DE})</td>
<td>* * * -</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1063DD</td>
<td>DECB</td>
<td>DECrement Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0054DD</td>
<td>NEG</td>
<td>NEGate</td>
<td>((\text{DE})\rightarrow\text{DE})</td>
<td>* * * *</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1054DD</td>
<td>NEGB</td>
<td>NEGate Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0055DD</td>
<td>ADC</td>
<td>ADD Carry</td>
<td>((\text{DE})\rightarrow\text{DE})</td>
<td>* * * *</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1055DD</td>
<td>ADCB</td>
<td>ADD Carry Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0056DD</td>
<td>SBC</td>
<td>SuBtract Carry</td>
<td>((\text{DE})\rightarrow\text{DE})</td>
<td>* * * *</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1056DD</td>
<td>SBCB</td>
<td>SuBtract Carry Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0057DD</td>
<td>TST</td>
<td>TeST</td>
<td>((\text{DE})\rightarrow\text{DE})</td>
<td>* * 0 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1057DD</td>
<td>TSTB</td>
<td>TeST Byte</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### B.3.3 Rotate/Shift

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
<th>Operation</th>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0060DD</td>
<td>ROR</td>
<td>Rotate Right</td>
<td>C</td>
<td>** ** ** **</td>
</tr>
<tr>
<td>1060DD</td>
<td>RORB</td>
<td>Rotate Right Byte</td>
<td>C</td>
<td>** ** ** **</td>
</tr>
<tr>
<td>0061DD</td>
<td>ROL</td>
<td>Rotate Left</td>
<td>C</td>
<td>** ** ** **</td>
</tr>
<tr>
<td>1061DD</td>
<td>ROLB</td>
<td>Rotate Left Byte</td>
<td>C</td>
<td>** ** ** **</td>
</tr>
<tr>
<td>0062DD</td>
<td>ASR</td>
<td>Arithmetic Shift Right</td>
<td>C</td>
<td>** ** ** **</td>
</tr>
<tr>
<td>1062DD</td>
<td>ASRB</td>
<td>Arithmetic Shift Right Byte</td>
<td>C</td>
<td>** ** ** **</td>
</tr>
<tr>
<td>0063DD</td>
<td>ASL</td>
<td>Arithmetic Shift Left</td>
<td>C</td>
<td>** ** ** **</td>
</tr>
<tr>
<td>1063DD</td>
<td>ASLB</td>
<td>Arithmetic Shift Left Byte</td>
<td>C</td>
<td>** ** ** **</td>
</tr>
<tr>
<td>0001DD</td>
<td>JMP</td>
<td>JUMP</td>
<td>DE\rightarrow PC</td>
<td>** ** ** **</td>
</tr>
<tr>
<td>0003DD</td>
<td>SWAB</td>
<td>SWAp Bytes</td>
<td>** ** 0 0</td>
<td>** ** ** **</td>
</tr>
</tbody>
</table>

### B.3.4 Operation Instructions Op

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
<th>Operation</th>
<th>Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>HALT</td>
<td>HALT</td>
<td>The computer stops all functions.</td>
<td>** ** ** **</td>
</tr>
<tr>
<td>000001</td>
<td>WAIT</td>
<td>WAIT</td>
<td>The computer stops and waits for an interrupt.</td>
<td>** ** ** **</td>
</tr>
</tbody>
</table>
| 000002  | RTI      | ReTurn from Interrupt | The PC and PS are popped off the SP stack: 
((SP))\rightarrow PC
(SP)+2\rightarrow SP
(SP)\rightarrow PS | ** ** ** **     |
| 000003  | 000003   | breakpoint trap | Trap to location 14. This is used to call ODT-11. | ** ** ** **     |
| 000004  | IOT      | Input/Output Trap | Trap to location 20. This is used to call IOX. | ** ** ** **     |
| 000005  | RESET    | RESET      | Returns all I/O device handlers to power-on state. | ** ** ** **     |
PAL-11S ASSEMBLY LANGUAGE AND ASSEMBLER

<table>
<thead>
<tr>
<th>Trapping Op or Op</th>
<th>E where 0 ≤ E ≤ 377</th>
</tr>
</thead>
<tbody>
<tr>
<td>104000-104377 EMT EMulator Trap</td>
<td></td>
</tr>
<tr>
<td>Trap to location * * * * 30. This is used to call system programs.</td>
<td></td>
</tr>
<tr>
<td>104400-104777 TRAP TRAP</td>
<td></td>
</tr>
<tr>
<td>Trap to location * * * * 34. This is used to call any routine desired by the programmer.</td>
<td></td>
</tr>
</tbody>
</table>

**CONDITION CODE OPERATES**

<table>
<thead>
<tr>
<th>Op-code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
</tr>
</thead>
<tbody>
<tr>
<td>000241</td>
<td>CLC</td>
<td>CLean CArry bit in PS.</td>
</tr>
<tr>
<td>000261</td>
<td>SEC</td>
<td>SET CArry bit.</td>
</tr>
<tr>
<td>000252</td>
<td>CLV</td>
<td>CLean oVerflow bit.</td>
</tr>
<tr>
<td>000262</td>
<td>SEV</td>
<td>SET oVerflow bit.</td>
</tr>
<tr>
<td>000244</td>
<td>CLZ</td>
<td>CLean Zero bit.</td>
</tr>
<tr>
<td>000264</td>
<td>SEZ</td>
<td>SET Zero bit.</td>
</tr>
<tr>
<td>000250</td>
<td>CLN</td>
<td>CLean Negative bit.</td>
</tr>
<tr>
<td>000270</td>
<td>SEN</td>
<td>SET Negative bit.</td>
</tr>
<tr>
<td>000254</td>
<td>CNZ</td>
<td>CLean Negative and Zero bits.</td>
</tr>
<tr>
<td>000257</td>
<td>CCC</td>
<td>CLean all Condition Codes.</td>
</tr>
<tr>
<td>000277</td>
<td>SCC</td>
<td>Set all Condition Codes.</td>
</tr>
<tr>
<td>000240</td>
<td>NOP</td>
<td>No-operation.</td>
</tr>
</tbody>
</table>

**B.3.5 Branch Instructions Op E Where -128_{10} < (E-.2)/2 < 127_{10}**

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
<th>Condition to be met if branch is to occur</th>
</tr>
</thead>
<tbody>
<tr>
<td>0004XX</td>
<td>BR</td>
<td>BrAnch always</td>
<td></td>
</tr>
<tr>
<td>0010XX</td>
<td>BNE</td>
<td>BrAnch if Not Equal to Zero Z=0</td>
<td></td>
</tr>
<tr>
<td>0014XX</td>
<td>BEQ</td>
<td>BrAnch if EQual (to zero) Z=1</td>
<td></td>
</tr>
<tr>
<td>0020XX</td>
<td>BGE</td>
<td>BrAnch if Greater than or equal (to zero) N(\bar{V})=0</td>
<td></td>
</tr>
<tr>
<td>0024XX</td>
<td>BLT</td>
<td>BrAnch if Less Than (zero) N(\bar{V}) = 1</td>
<td></td>
</tr>
<tr>
<td>0030XX</td>
<td>BGT</td>
<td>BrAnch if Greater Than (zero) Z(\bar{N})(\bar{V}) = 0</td>
<td></td>
</tr>
<tr>
<td>0034XX</td>
<td>BLE</td>
<td>BrAnch if Less than or Equal (to zero) Z(\bar{N})(\bar{V}) = 1</td>
<td></td>
</tr>
</tbody>
</table>

B-6
B.3.6 Subroutine Call JSR ER,A

Op-code | MNEMONIC | Stands for | Operation
---|---|---|---
004RDD | JSR | Jump to Sub-Routine | Push register on the SP stack, put the PC in the register: 
DE ← TEMP -a temporary storage register internal to processor
(SP) ← SP
(REG) ← (SP)
(PC) + m REG -m depends upon the address mode.
(TEMP) ← PC

B.3.7 Subroutine Return

Op-code | MNEMONIC | Stands for | Operation
---|---|---|---
00020R | RTS | ReTurn from Subroutine | Put register contents in PC and pop old contents from SP stack into register.

B.3.8 Extensions for the LSI-11 Version Of PAL-11S

Op-code | MNEMONIC | Stands for | Operation | Condition Codes
---|---|---|---|---
0067dd | SXT | Sign eXtend | Nx(-1) DE | N Z V C
1067dd | MFPS | Move byte From PS | (PS) DE | * * 0 -

¹These extensions are available only with the LSI-11 version of PAL-11S.
### PAL-11S Assembly Language and Assembler

<table>
<thead>
<tr>
<th>Decimal</th>
<th>mnemonic</th>
<th>operation</th>
<th>syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>1064ss</td>
<td>MTPS</td>
<td>Move byte To PS</td>
<td>(SE) PS * * * *</td>
</tr>
<tr>
<td>074rdd</td>
<td>XOR</td>
<td>eXclusive OR</td>
<td>r ! (DE) DE * * 0 -</td>
</tr>
<tr>
<td>070rss</td>
<td>MUL</td>
<td>MULTiply</td>
<td>r x (SE) r * * 0 *</td>
</tr>
<tr>
<td>071rss</td>
<td>DIV</td>
<td>DIVide</td>
<td>r / (SE) r * * * *</td>
</tr>
<tr>
<td>072rss</td>
<td>ASH</td>
<td>Arithmetic Shift</td>
<td>* * * *</td>
</tr>
<tr>
<td>073rss</td>
<td>ASHC</td>
<td>Arithmetic Shift Combined</td>
<td>* * * *</td>
</tr>
<tr>
<td>0064nn</td>
<td>MARK</td>
<td>MARK</td>
<td>SP+2xnn SP SP^ R5 PC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>R5 PC</td>
</tr>
<tr>
<td>077rnn</td>
<td>SOB</td>
<td>Subtract One and Branch if 0</td>
<td>(r)-1 r; if (r) 0 then (PC)-2xnn PC</td>
</tr>
<tr>
<td>000006</td>
<td>RTT</td>
<td>ReTurn from Trap loaded from stack</td>
<td>((SP)) PC ((SP)) +2 SP ((SP)) PS (SP)+2 SP</td>
</tr>
</tbody>
</table>

### B.4 Assembler Directives

<table>
<thead>
<tr>
<th>mnemonic</th>
<th>operand</th>
<th>stands for</th>
<th>operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>.EOT</td>
<td>none</td>
<td>End Of Tape</td>
<td>Indicates the physical end of the source input medium</td>
</tr>
<tr>
<td>.EVEN</td>
<td>none</td>
<td>EVEN</td>
<td>Insures that the assembly location counter is even by adding 1 if it is odd.</td>
</tr>
<tr>
<td>.END</td>
<td>E</td>
<td>END</td>
<td>Indicates the physical and logical end of the program and optionally specifies the entry point (E)</td>
</tr>
<tr>
<td></td>
<td>(E optional)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.WORD</td>
<td>E,E,...</td>
<td>WORD</td>
<td>Generates words of data</td>
</tr>
<tr>
<td></td>
<td>(the void operator)</td>
<td></td>
<td>Generates words of data</td>
</tr>
<tr>
<td>.BYTE</td>
<td>E,E,...</td>
<td>BYTE</td>
<td>Generates bytes of data</td>
</tr>
<tr>
<td>.ASCII</td>
<td>/xxx...x/</td>
<td>ASCII</td>
<td>Generates 7-bit ASCII characters for text enclosed by delimiters.</td>
</tr>
<tr>
<td>.TITLE</td>
<td>NAME</td>
<td>TITLE</td>
<td>Generates a name for the object module.</td>
</tr>
<tr>
<td>.ASECTION</td>
<td>none</td>
<td>ASECTION</td>
<td>Initiates the Absolute section.</td>
</tr>
<tr>
<td>.OFFSET</td>
<td>none</td>
<td>OFFSET</td>
<td>Initiates the Relocatable Control section.</td>
</tr>
<tr>
<td>.LIMIT</td>
<td>none</td>
<td>LIMIT</td>
<td>Generates two words containing the limit.</td>
</tr>
<tr>
<td>Command</td>
<td>Description</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-------------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.GLOBL NAME,NAME,... GLOBAL</td>
<td>Specifies each name to be a global symbol</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.RAD50 /XXX/ RADIX 50</td>
<td>Generates the RADIX 50 representation of the ASCII character in delimiters.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IFZ E IF E=0</td>
<td>Assemble what follows up to the terminating .ENDC if the expression E is 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IFNZ E IF E≠0</td>
<td>Assemble what follows up to the terminating .ENDC, if the expression E is not 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IFL E IF E&lt;0</td>
<td>Assemble what follows up to the terminating .ENDC, if the expression E is less than 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IFLE E IF E≤0</td>
<td>Assemble what follows up to the terminating .ENDC, if the expression E is less than or equal to 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IFG E IF E&gt;0</td>
<td>Assemble what follows up to the terminating .ENDC, if the expression E is greater than 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IFGE E IF E≥0</td>
<td>Assemble what follows up to the terminating .ENDC, if the expression E is greater than or equal to 0.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IFDF NAME IF NAME defined</td>
<td>Assemble what follows up to the terminating .ENDC if the symbol NAME is defined.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.IFNDF NAME IF NAME undefined</td>
<td>Assemble what follows up to the terminating .ENDC if the symbol NAME is undefined.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>.ENDC none End of Conditional</td>
<td>Terminates the range of a conditional directive.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Error Code</td>
<td>Meaning</td>
<td></td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>---------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td>Addressing error. An address within the instruction is incorrect. Also includes relocation errors.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>Bounding error. Instructions or word data are being assembled at an odd address in memory.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td>Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L</td>
<td>Line buffer overflow. All extra characters beyond 72 are ignored.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a previously encountered label.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>N</td>
<td>Number containing an 8 or 9 was not terminated by a decimal point.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>Phase error. A label's definition or value varies from one pass to another.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Q</td>
<td>Questionable syntax. There are missing arguments or the instruction scan was not completed, or a carriage return was not followed by a linefeed or form feed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R</td>
<td>Register-type error. An invalid use of or reference to a register has been made.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>S</td>
<td>Symbol table overflow. When the quantity of user-defined symbols exceeds the allocated space available in the user's symbol table, the assembler outputs the current source line with the S error code, then returns to the command string interpreter to await the next command string to be typed.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T</td>
<td>Truncation error. More than the allotted number of bits were input so the leftmost bits were truncated. T error does not occur for the result of an expression.</td>
<td></td>
<td></td>
</tr>
<tr>
<td>U</td>
<td>Undefined symbol. An undefined symbol was encountered during the evaluation of an expression. Relative to the expression, the undefined symbol is assigned a value of zero.</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
B.6 INITIAL OPERATING PROCEDURES

Loading: Use Absolute Loader. The start address of the Loader must be in the console switches.

Storage Requirements: PAL-11S uses 8K of memory.

Starting: Immediately upon loading, PAL-11S will be in control and initiate dialogue.

Initial Dialogue:

Printout        Inquiry
*S             What is the input device of the Source symbolic tape?
*B             What is the output device of the Binary object tape?
*L             What is the output device of the assembly Listing?
*T             What is the output device of the symbol Table?

Each of these questions may be answered by any one of the following characters:

Character       Answer Indicated
T               Teleprinter keyboard
L               Low-speed reader or punch
H               High-speed reader or punch
P               Line Printer

Each of these answers may be followed by the other characters indicating options:

Option Typed    Function to be performed
/1              on pass 1
/2              on pass 2
/3              on pass 3
/E              errors to be listed on the Teletype on the same pass (meaningful only for *B or *L).

Each answer is terminated by typing the RETURN key. A RETURN alone as answer will delete the function.

Dialogue During Assembly:

Printout        Response
EOF ?            Place next tape in reader and type RETURN. A .END statement may be forced by typing E followed by RETURN.
END ?            Start next pass by placing first tape in reader and typing RETURN.
EOM? If the end-of-medium is on the listing device, the device may be readied and the assembly may be continued by typing RETURN.

If the end-of-medium is on the binary device, the assembler will discontinue the assembly and restart itself.

Restarting: Type CTRL/P. The initial dialogue will be started again.
APPENDIX C
PAL-11A ASSEMBLY LANGUAGE AND ASSEMBLER

C.1 SPECIAL CHARACTERS

<table>
<thead>
<tr>
<th>Character</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>form feed</td>
<td>Source line terminator</td>
</tr>
<tr>
<td>line feed</td>
<td>Source line terminator</td>
</tr>
<tr>
<td>carriage return</td>
<td>Source statement terminator</td>
</tr>
<tr>
<td>:</td>
<td>Label terminator</td>
</tr>
<tr>
<td>=</td>
<td>Direct assignment indicator</td>
</tr>
<tr>
<td>%</td>
<td>Register term indicator</td>
</tr>
<tr>
<td>tab</td>
<td>Item terminator</td>
</tr>
<tr>
<td>space</td>
<td>Field terminator</td>
</tr>
<tr>
<td>#</td>
<td>Immediate expression indicator</td>
</tr>
<tr>
<td>@</td>
<td>Deferred addressing indicator</td>
</tr>
<tr>
<td>(</td>
<td>Initial register indicator</td>
</tr>
<tr>
<td>)</td>
<td>Terminal register indicator</td>
</tr>
<tr>
<td>,</td>
<td>Operand field separator</td>
</tr>
<tr>
<td>;</td>
<td>Comment field indicator</td>
</tr>
<tr>
<td>+</td>
<td>Arithmetic addition operator</td>
</tr>
<tr>
<td>-</td>
<td>Arithmetic subtraction operator</td>
</tr>
<tr>
<td>&amp;</td>
<td>Logical AND operator</td>
</tr>
<tr>
<td>!</td>
<td>Logical OR operator</td>
</tr>
<tr>
<td>&quot;</td>
<td>Double ASCII character indicator</td>
</tr>
<tr>
<td>'</td>
<td>Single ASCII character indicator</td>
</tr>
<tr>
<td>.</td>
<td>Assembly location counter</td>
</tr>
</tbody>
</table>
### C.2 ADDRESS MODE SYNTAX

n is an integer between 0 and 7 representing a register. R is a register expression, E is an expression, ER is either a register expression or an expression in the range 0 to 7.

<table>
<thead>
<tr>
<th>Format</th>
<th>Address Mode Name</th>
<th>Address Mode Number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>Register</td>
<td>0n</td>
<td>Register R contains the operand. R is a register expression.</td>
</tr>
<tr>
<td>@(R) or (ER)</td>
<td>Deferred Register</td>
<td>1n</td>
<td>Register R contains the operand address.</td>
</tr>
<tr>
<td>(ER)+</td>
<td>Autoincrement</td>
<td>2n</td>
<td>The contents of the register specified by ER are incremented after being used as the address of the operand.</td>
</tr>
<tr>
<td>@-(ER)</td>
<td>Autodecrement</td>
<td>4n</td>
<td>The contents of register ER are decremented before being used as the address of the operand.</td>
</tr>
<tr>
<td>@-(ER)</td>
<td>Deferred Autodecrement</td>
<td>5n</td>
<td>The contents of register ER are decremented before being used as the pointer to the address of the operand.</td>
</tr>
<tr>
<td>E(ER)</td>
<td>Index</td>
<td>6n</td>
<td>E plus the contents of the register specified, ER, is the address of the operand.</td>
</tr>
<tr>
<td>@E(ER)</td>
<td>Deferred Index</td>
<td>7n</td>
<td>E added to ER gives the pointer to the address of the operand.</td>
</tr>
<tr>
<td>#E</td>
<td>Immediate</td>
<td>27</td>
<td>E is the operand.</td>
</tr>
<tr>
<td>@##E</td>
<td>Absolute</td>
<td>37</td>
<td>E is the address of the operand.</td>
</tr>
<tr>
<td>E</td>
<td>Relative</td>
<td>67</td>
<td>E is the address of the operand.</td>
</tr>
<tr>
<td>@E</td>
<td>Deferred Relative</td>
<td>77</td>
<td>E is the pointer to the address of the operand.</td>
</tr>
</tbody>
</table>
C.3 INSTRUCTIONS

The instructions which follow are grouped according to the operands they take and the bit patterns of their op-codes.

In the representation of op-codes, the following symbols are used:

- **SS**: Source operand specified by a 6-bit address mode.
- **DD**: Destination operand specified by a 6-bit address mode.
- **XX**: 8-bit offset to a location (branch instructions)
- **R**: Integer between 0 and 7 representing a general register.

Symbols used in the description of instruction operations are:

- **SE**: Source Effective address
- **DE**: Destination Effective address
- **()**: Contents of
- **→**: Is transferred to
- **PS**: Processor Status word

The condition codes in the processor status word (PS) are affected by the instructions. These condition codes are represented as follows:

- **N**: Negative bit: set if the result is negative
- **Z**: Zero bit: set if the result is zero
- **V**: Overflow bit: set if the operation caused an overflow
- **C**: Carry bit: set if the operation caused a carry

In the representation of the instruction's effect on the condition codes, the following symbols are used:

- *****: Conditionally set
- **-**: Not affected
- **0**: Cleared
- **1**: Set

To set conditionally means to use the instruction's result to determine the state of the code (see the PDP-11 Processor Handbook).

Logical operations are represented by the following symbols:

- **l**: Inclusive OR
- **!**: Exclusive OR
- **&**: AND
- **-** (used over a symbol) NOT (i.e., 1's complement)
### C.3.1 Double-Operand Instructions  Op A,A

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
<th>Operation</th>
<th>Status Word Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>01SSDD</td>
<td>MOV</td>
<td>MOVE</td>
<td>(SE) → DE</td>
<td>*   *   0   -</td>
</tr>
<tr>
<td>11SSDD</td>
<td>MOVB</td>
<td>MOVE Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>02SSDD</td>
<td>CMP</td>
<td>CoMPare</td>
<td>(SE) − (DE)</td>
<td>*   *   *   *</td>
</tr>
<tr>
<td>12SSDD</td>
<td>CMPB</td>
<td>CoMPare Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>03SSDD</td>
<td>BIT</td>
<td>BIT Test</td>
<td>(SE) &amp; (DE)</td>
<td>*   *   0   -</td>
</tr>
<tr>
<td>13SSDD</td>
<td>BITB</td>
<td>BIT Test Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>04SSDD</td>
<td>BIC</td>
<td>BIT Clear</td>
<td>(SE) &amp; (DE) → DE</td>
<td>*   *   0   -</td>
</tr>
<tr>
<td>14SSDD</td>
<td>BICB</td>
<td>BIT Clear Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>05SSDD</td>
<td>BIS</td>
<td>BIT Set</td>
<td>(SE) !(DE) → DE</td>
<td>*   *   0   -</td>
</tr>
<tr>
<td>15SSDD</td>
<td>BISB</td>
<td>BIT Set Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>06SSDD</td>
<td>ADD</td>
<td>ADD</td>
<td>(SE) +(DE) → DE</td>
<td>*   *   *   *</td>
</tr>
<tr>
<td>16SSDD</td>
<td>SUB</td>
<td>SUBtract</td>
<td>(DE) − (SE) → DE</td>
<td>*   *   *   *</td>
</tr>
</tbody>
</table>

### C.3.2 Single-Operand Instructions  Op A

<table>
<thead>
<tr>
<th>Op-Codes</th>
<th>MNEMONIC</th>
<th>Stands for</th>
<th>Operation</th>
<th>Status Word Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0050DD</td>
<td>CLR</td>
<td>CLeaR</td>
<td>0 DE</td>
<td>0   1   0   0</td>
</tr>
<tr>
<td>1050DD</td>
<td>CLRB</td>
<td>CLeaR Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0051DD</td>
<td>COM</td>
<td>COMplement</td>
<td>(DE) DE</td>
<td>*   *   0   1</td>
</tr>
<tr>
<td>1051DD</td>
<td>COMB</td>
<td>COMplement Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0052DD</td>
<td>INC</td>
<td>INCrement</td>
<td>(DE) +1 DE</td>
<td>*   *   *   -</td>
</tr>
<tr>
<td>1052DD</td>
<td>INCB</td>
<td>INCrement Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0053DD</td>
<td>DEC</td>
<td>DECrement</td>
<td>(DE) −1 DE</td>
<td>*   *   *   -</td>
</tr>
<tr>
<td>1053DD</td>
<td>DECB</td>
<td>DECrement Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0054DD</td>
<td>NEG</td>
<td>NEGate</td>
<td>(DE) +1 DE</td>
<td>*   *   *   *</td>
</tr>
<tr>
<td>1054DD</td>
<td>NEGB</td>
<td>NEGate Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0055DD</td>
<td>ADC</td>
<td>Add Carry</td>
<td>(DE) +(C) → DE</td>
<td>*   *   *   *</td>
</tr>
<tr>
<td>1055DD</td>
<td>ADCB</td>
<td>Add Carry Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0056DD</td>
<td>SBC</td>
<td>SubTract Carry</td>
<td>(DE) − (C) → DE</td>
<td>*   *   *   *</td>
</tr>
<tr>
<td>1056DD</td>
<td>SBCB</td>
<td>SubTract Carry Byte</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0057DD</td>
<td>TST</td>
<td>TeST</td>
<td>(DE) −0 → DE</td>
<td>*   *   0   0</td>
</tr>
<tr>
<td>1057DD</td>
<td>TSTB</td>
<td>TeST Byte</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### C.3.3 Rotate/Shift Instructions Op A

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
<th>Operation</th>
<th>Status Word Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>0060DD</td>
<td>ROR</td>
<td>ROTate Right</td>
<td><img src="#" alt="Diagram" /></td>
<td>* * * *</td>
</tr>
<tr>
<td>1060DD</td>
<td>RORB</td>
<td>ROTate Right Byte</td>
<td>even or odd byte</td>
<td>* * * *</td>
</tr>
<tr>
<td>0061DD</td>
<td>ROL</td>
<td>ROTate Left</td>
<td><img src="#" alt="Diagram" /></td>
<td>* * * *</td>
</tr>
<tr>
<td>1061DD</td>
<td>ROLB</td>
<td>ROTate Left Byte</td>
<td>even or odd byte</td>
<td>* * * *</td>
</tr>
<tr>
<td>0062DD</td>
<td>ASR</td>
<td>Arithmetic Shift Right</td>
<td><img src="#" alt="Diagram" /></td>
<td>* * * *</td>
</tr>
<tr>
<td>1062DD</td>
<td>ASRB</td>
<td>Arithmetic Shift Right Byte</td>
<td>even or odd byte</td>
<td>* * * *</td>
</tr>
<tr>
<td>0063DD</td>
<td>ASL</td>
<td>Arithmetic Shift Left</td>
<td><img src="#" alt="Diagram" /></td>
<td>* * * *</td>
</tr>
<tr>
<td>1063DD</td>
<td>ASLB</td>
<td>Arithmetic Shift Left Byte</td>
<td>even or odd byte</td>
<td>* * * *</td>
</tr>
<tr>
<td>0001DD</td>
<td>JMP</td>
<td>Jump</td>
<td>DE → PC</td>
<td></td>
</tr>
<tr>
<td>0003DD</td>
<td>SWAB</td>
<td>Swap Bytes</td>
<td><img src="#" alt="Diagram" /></td>
<td>* * 0 0</td>
</tr>
</tbody>
</table>

### C.3.4 Operate Instructions Op

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
<th>Operation</th>
<th>Status Word Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>HALT</td>
<td>HALT</td>
<td>The computer stops all functions.</td>
<td>- - - -</td>
</tr>
<tr>
<td>000001</td>
<td>WAIT</td>
<td>WAIT</td>
<td>The computer stops and waits for an interrupt.</td>
<td>- - - -</td>
</tr>
</tbody>
</table>
## C.3.5 Trap Instructions

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
<th>Operation</th>
<th>Status Word Condition Codes</th>
</tr>
</thead>
<tbody>
<tr>
<td>000003</td>
<td>(none)</td>
<td>(breakpoint Trap to location 14. This is used to call ODT.)</td>
<td>* * * *</td>
<td>N Z V C</td>
</tr>
<tr>
<td>000004</td>
<td>IOT</td>
<td>Input/Output Trap</td>
<td>Trap to location 20. This is used to call IOX.</td>
<td>* * *</td>
</tr>
<tr>
<td>104000-104377</td>
<td>EMT</td>
<td>EMulator Trap</td>
<td>Trap to location 30. This is used to call system programs.</td>
<td>* * *</td>
</tr>
<tr>
<td>104400-104777</td>
<td>TRAP</td>
<td>TRAP</td>
<td>Trap to location 34. This is used to call any routine desired by the programmer.</td>
<td>* * *</td>
</tr>
</tbody>
</table>

*Op (only)

### CONDITION CODE OPERATES

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
</tr>
</thead>
<tbody>
<tr>
<td>000241</td>
<td>CLC</td>
<td>Clear Carry bit in PS.</td>
</tr>
<tr>
<td>000261</td>
<td>SEC</td>
<td>Set Carry bit.</td>
</tr>
<tr>
<td>000242</td>
<td>CLV</td>
<td>Clear oVerflow bit.</td>
</tr>
<tr>
<td>000262</td>
<td>SEV</td>
<td>Set oVerflow bit.</td>
</tr>
<tr>
<td>000244</td>
<td>CLZ</td>
<td>Clear Zero bit.</td>
</tr>
<tr>
<td>000264</td>
<td>SEZ</td>
<td>Set Zero bit.</td>
</tr>
<tr>
<td>000250</td>
<td>CLN</td>
<td>Clear Negative bit.</td>
</tr>
<tr>
<td>000270</td>
<td>SEN</td>
<td>Set Negative bit.</td>
</tr>
<tr>
<td>000254</td>
<td>CNZ</td>
<td>Clear Negative and Zero bits.</td>
</tr>
<tr>
<td>000257</td>
<td>CCC</td>
<td>Clear all Condition Codes.</td>
</tr>
<tr>
<td>000277</td>
<td>SCC</td>
<td>Set all Condition Codes.</td>
</tr>
</tbody>
</table>
### C.3.6 Branch Instructions Op E where $-128_{10} \leq \frac{(E-2)}{2} \leq 127_{10}$

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
<th>Condition to be met if branch is to occur</th>
</tr>
</thead>
<tbody>
<tr>
<td>0004XX</td>
<td>BR</td>
<td>Branch always</td>
<td>N/A</td>
</tr>
<tr>
<td>0010XX</td>
<td>BNE</td>
<td>Branch if Not Equal (to zero)</td>
<td>Z=0</td>
</tr>
<tr>
<td>0014XX</td>
<td>BEQ</td>
<td>Branch if Equal (to zero)</td>
<td>Z=1</td>
</tr>
<tr>
<td>0020XX</td>
<td>BGE</td>
<td>Branch if Greater than or Equal (to zero)</td>
<td>N:(\bar{V}=0)</td>
</tr>
<tr>
<td>0024XX</td>
<td>BLT</td>
<td>Branch if Less Than (zero)</td>
<td>N:(\bar{V}=1)</td>
</tr>
<tr>
<td>0030XX</td>
<td>BGT</td>
<td>Branch if Greater than (zero)</td>
<td>Z! (N:(\bar{V})=0)</td>
</tr>
<tr>
<td>0034XX</td>
<td>BLE</td>
<td>Branch if Less than or Equal (to zero)</td>
<td>Z! (N:(\bar{V})=1)</td>
</tr>
<tr>
<td>1000XX</td>
<td>BPL</td>
<td>Branch if Plus</td>
<td>N=0</td>
</tr>
<tr>
<td>1004XX</td>
<td>BMI</td>
<td>Branch if Minus</td>
<td>N=1</td>
</tr>
<tr>
<td>1010XX</td>
<td>BHI</td>
<td>Branch if Higher</td>
<td>C ! Z = 0</td>
</tr>
<tr>
<td>1014XX</td>
<td>BLO</td>
<td>Branch if Lower or Same</td>
<td>C ! Z = 1</td>
</tr>
<tr>
<td>1020XX</td>
<td>BVC</td>
<td>Branch if Overflow Clear</td>
<td>V=0</td>
</tr>
<tr>
<td>1024XX</td>
<td>BVS</td>
<td>Branch if Overflow Set</td>
<td>V=1</td>
</tr>
<tr>
<td>1030XX</td>
<td>BCC</td>
<td>Branch if Carry Clear (or Branch if Higher or Same)</td>
<td>C=0</td>
</tr>
<tr>
<td>1034XX</td>
<td>BCS</td>
<td>Branch if Carry Set (or BLO)</td>
<td>C=1</td>
</tr>
</tbody>
</table>

### C.3.7 Subroutine Call Op ER, A

<table>
<thead>
<tr>
<th>Op-Code</th>
<th>MNEMONIC</th>
<th>Stands for</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>004RDD</td>
<td>JSR</td>
<td>Jump to SubRoutine</td>
<td>Push register on the SP stack, put the PC in the register:</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>DE (TEMP) - a temporary storage register</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>internal to processor.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(SP)-2 \rightarrow SP</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(REG) \rightarrow (SP)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(PC) \rightarrow REG</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>(TEMP) \rightarrow PC</td>
</tr>
</tbody>
</table>

C-7
C.3.8 Subroutine Return Op ER

Op-Code  MNEMONIC   Stands for                        Operation

00020R   RTS   ReTurn from Sub-                                    Put register contents into PC and pop old contents from SP stack into register.
          routine

C.4 ASSEMBLER DIRECTIVES

Op-Code  MNEMONIC   Stands for                        Operation

.EOT   End Of Tape                                      Indicates the physical end of the source input medium
.EVEN   EVEN                                             Ensures that the assembly location counter is even by adding 1 if it is odd
.END m   END (m optional)                                Indicates the physical and logical end of the program and optionally specifies the entry point (m)

.WORD   WORD                                             Generates words of data
          E,E,...                                             Generates words of data
          (the void operator)
.BYTE   BYTE                                             Generates bytes of data
          E,E,...                                             Generates 7-bit Ascii character for the text enclosed by delimiters
.ASCII   ASCII
          /xxx...x/                                          

C.5 ERROR CODES

Error Code  Meaning

A   Addressing error. An address within the instruction is Incorrect.
B   Bounding error. Instructions or word data are being assembled at an odd address in memory.
D   Doubly-defined symbol referenced. Reference was made to a symbol which is defined more than once.
I   Illegal character detected. Illegal characters which are also non-printing are replaced by a ? on the listing.
L   Line buffer overflow. Extra characters (more than 72\textsubscript{10}) are ignored.
M   Multiple definition of a label. A label was encountered which was equivalent (in the first six characters) to a previously encountered label.
N   Number containing an 8 or 9 has a decimal point missing.
P   Phase error. A label's definition or value varies from one pass to another.

C-8
Questionable syntax. There are missing arguments or the instruction scan was not completed, or a carriage return was not followed by a line feed or form feed.

Register-type error. An invalid use of or reference to a register has been made.

Symbol-table overflow. When the quantity of user-defined symbols exceeds the allocated space available in the user's symbol table, the assembler outputs the current source line with the S error code, then returns to the command string interpreter to await the next command string to be typed.

Truncation error. A number was too big for the allotted number of bits; the leftmost bits were truncated. T error does not occur for the result of an expression.

Undefined symbol. An undefined symbol was encountered during the evaluation of an expression. Relative to the expression, the undefined symbol is assigned a value of zero.

C.6 INITIAL OPERATING PROCEDURES

Loading: Use Absolute Loader (see Chapter 6). Make sure that the start address of the absolute loader is in the switches when the assembler is loaded.

Storage Requirements: PAL-11A exists in 4K and 8K versions.

Starting Immediately upon loading, PAL-11A will be in control and initiate dialogue.

Initial Dialogue:

Printout Inquiry
*S What is the input device of the Source symbolic tape?
*B What is the output device of the Binary object tape?
*L What is the output device of the assembly Listing?
*T What is the output device of the symbol Table?

Each of these questions may be answered by one of the following characters:

<table>
<thead>
<tr>
<th>Character</th>
<th>Answer Indicated</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>Teletype keyboard</td>
</tr>
<tr>
<td>L</td>
<td>Low-speed reader or punch</td>
</tr>
<tr>
<td>H</td>
<td>High-speed reader or punch</td>
</tr>
<tr>
<td>P</td>
<td>line Printer (8K version only)</td>
</tr>
</tbody>
</table>
PAL-11A ASSEMBLY LANGUAGE AND ASSEMBLER

Each of these answers may be followed by other characters indicating options:

<table>
<thead>
<tr>
<th>Option Typed</th>
<th>Function to be Performed</th>
</tr>
</thead>
<tbody>
<tr>
<td>/1</td>
<td>on pass 1</td>
</tr>
<tr>
<td>/2</td>
<td>on pass 2</td>
</tr>
<tr>
<td>/3</td>
<td>on pass 3</td>
</tr>
<tr>
<td>/E</td>
<td>errors to be listed on the Teletype on the same pass (meaningful or *B or *L only)</td>
</tr>
</tbody>
</table>

Each answer is terminated by typing the RETURN key. A RETURN alone as answer will delete the function.

Dialogue during assembly:

<table>
<thead>
<tr>
<th>Printout</th>
<th>Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>EOF ?</td>
<td>Place next tape in reader and type RETURN. A .END statement may be forced by typing E followed by RETURN.</td>
</tr>
<tr>
<td>END ?</td>
<td>Start next pass by placing first tape in reader and typing RETURN.</td>
</tr>
<tr>
<td>EOM ?</td>
<td>If listing on HSP or LPT, replenish tape or paper and type RETURN. If binary on HSP, start assembly again.</td>
</tr>
<tr>
<td>Restarting:</td>
<td>Type CTRL/P. The initial dialogue will be started again.</td>
</tr>
</tbody>
</table>
APPENDIX D
TEXT EDITOR, ED-11

D.1 INPUT/OUTPUT COMMANDS

R Reads a page of text from input device, and appends it to the contents (if any) of the page buffer. Dot is moved to the beginning of the page and Marked. (See B and M below.)

O Opens the input device when the user wishes to continue input with a new tape in the reader.

ARGUMENTS

\[
\begin{array}{ll}
\text{n} & \text{(n) beginning at Dot and ending with nth line feed character.} \\
\text{-n} & \text{(-n) beginning with 1st character following the (n+1)th previous line feed and terminating at Dot.} \\
\text{0} & \text{(0) beginning with 1st character of current line and ending at Dot.} \\
\text{@} & \text{(0) bounded by Dot and the Marked location (see M).} \\
\text{P} & \text{Punches the character string \text{/} beginning at Dot and ending with the last character in the page.} \\
\end{array}
\]

F Outputs a Form Feed character and four inches of blank tape.

nT Punches four inches of Trailer (blank tape) n times.

nN Punches contents of the page buffer (followed by a trailer if a form feed is present), deletes the contents of the buffer, and reads the next page into the page buffer. It does this n times. At completion, Dot and Mark are located at the beginning of the page buffer.

V Lists the entire line containing Dot (i.e., from previous line feed to next line feed or form feed).

< Same as -1L. If Dot is located at the beginning of a line, this simply lists the line preceding the current line.

> Lists the line following the current line.
D.2 POINTER-POSITIONING COMMANDS

B     Moves Dot to the beginning of the page.
E     Moves Dot to the end of the page.
M     Marks the current position of Dot for later reference in a command using the argument @. Certain commands implicitly move Mark.

\[ \begin{array}{cccc}
0 & J & \text{Moves Dot:} & (0) \quad \text{to the beginning of the current line} \\
@ & & (\@) \quad \text{to the Marked location} \\
/ & & (/) \quad \text{to the end of the page} \\
\end{array} \]

\[ \begin{array}{cccc}
n & (n) \quad \text{forward past n characters} \\
-n & (-n) \quad \text{backward past n characters} \\
\end{array} \]

\[ \begin{array}{cccc}
0 & A & \text{Moves Dot:} & (0) \quad \text{to the beginning of current line} \\
@ & & (\@) \quad \text{to the Marked location} \\
/ & & (/) \quad \text{to the end of the page} \\
\end{array} \]

D.3 SEARCH COMMANDS

nG     Gets (searches for) the nth occurrence of the specified character string on the current page. Dot is set immediately after the last character in the found text, and the characters from the beginning of the line to Dot are listed on the teleprinter. If the search is unsuccessful, Dot will be at the end of the buffer and a ? will be printed out.

XXXX  Searches the whole file for the next occurrence of the specified character string. Combines G and N commands. If search is not successful on current page, it continues on Next page. Dot is set immediately after the last character in the found text and the characters from the beginning of the line to Dot are listed on the teleprinter. If the Search object is not found, Dot will be at the end of the buffer and a ? will be printed out. In such a case, all text scanned is copied to the output tape.

D.4 COMMANDS TO MODIFY THE TEXT

<table>
<thead>
<tr>
<th>Character-Oriented</th>
<th>Line-Oriented</th>
</tr>
</thead>
<tbody>
<tr>
<td>nD</td>
<td>nK</td>
</tr>
<tr>
<td>nC XXXX</td>
<td>nX eXchanges</td>
</tr>
<tr>
<td></td>
<td>XXXX</td>
</tr>
<tr>
<td>-nD</td>
<td>-nK</td>
</tr>
<tr>
<td>-nC XXXX</td>
<td>-nX eXchanges</td>
</tr>
<tr>
<td></td>
<td>XXXX</td>
</tr>
</tbody>
</table>

the character string beginning at Dot and ending at the nth end-of-line.

the character string beginning with the first character following the (n+1)th previous end-of-line and ending at Dot.
D.5 SYMBOLS

Dot
Location following the most recent character operated upon.

† Holding down the CTRL key (not the † key) in combination with another keyboard character.

RETURN
If in command mode, it executes the current command; goes into Text Mode if required. If in Text Mode, it terminates the current line, enters a carriage return and line feed into the buffer and stays in text mode. At all times causes the carriage to move to the beginning of a new line. (RETURN is often symbolized as ↵).

† (Typing the LINE FEED key) Terminates Text Mode unless the first character typed in Text Mode; executes the current command.

CTRL/FORM A Form feed which terminates, and thus defines, a page of the user's text.

D.6 GROUPING OF COMMANDS

<table>
<thead>
<tr>
<th>No Arguments</th>
<th>Argument n only</th>
<th>All Arguments [n,-n,0,θ,/]</th>
</tr>
</thead>
<tbody>
<tr>
<td>V (Verify:</td>
<td>G (Get)</td>
<td>A (Advance)</td>
</tr>
<tr>
<td>Lists current line)</td>
<td>N (Next)</td>
<td>C (Change)</td>
</tr>
<tr>
<td>&lt; (Lists previous line)</td>
<td>T (Trailer)</td>
<td>D (Delete)</td>
</tr>
<tr>
<td>&gt; (Lists next line)</td>
<td></td>
<td>J (Jump)</td>
</tr>
<tr>
<td>B (Begin)</td>
<td>K (Kill)</td>
<td></td>
</tr>
<tr>
<td>E (End)</td>
<td>L (List)</td>
<td></td>
</tr>
<tr>
<td>F (Form feed)</td>
<td>P (Punch)</td>
<td></td>
</tr>
<tr>
<td>H (wHole)</td>
<td>X (eXchange)</td>
<td></td>
</tr>
<tr>
<td>I (Insert)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>M (Mark)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>O (Open)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>R (Read)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

D-3
D.7 OPERATING PROCEDURES

### D.7.1 Loading:
Use Absolute Binary Loader (see Chapter 5).

### D.7.2 Storage Requirements:
ED-11 uses all of core.

### D.7.3 Starting:
Immediately upon loading, ED-11 will be in control.

### D.7.4 Initial Dialogue:

<table>
<thead>
<tr>
<th>Program Types</th>
<th>User Response</th>
</tr>
</thead>
<tbody>
<tr>
<td>*I</td>
<td>L±/ (if LSR is to be used for source input)</td>
</tr>
<tr>
<td></td>
<td>H±/ (if HSR is to be used for source input)</td>
</tr>
<tr>
<td>*O</td>
<td>L±/ (if LSP is to be used for edited output)</td>
</tr>
<tr>
<td></td>
<td>H±/ (if HSP is to be used for edited output)</td>
</tr>
</tbody>
</table>

If the output device is the high-speed punch (HSP), Editor enters command mode to accept input. Otherwise the sequence continues with:

LSP OFF? /

(when LSP is off)

Upon input of / from the keyboard, Editor enters command mode and is ready to accept input.

### D.7.5 Restarting:

Type CTRL/P twice, initiating the normal initial dialogue. The text to be edited should be loaded (or reloaded) at this time.
APPENDIX E

DEBUGGING OBJECT PROGRAMS ON-LINE, ODT-11 AND ODT-11X

E.1 SUMMARY OF CONTENTS

ODT indicates readiness to accept commands by typing * or by opening a
location by printing its contents.

1. ODT-11
   n/       opens word n
   \       reopens last word opened
   RETURN key  closes open location
   +       opens next location
   †       opens previous location
   +       opens relatively addressed word
   $n/     opens general register n (0-7)
   n;G     goes to word n and starts execution
   n;B     sets breakpoint at word n
   ;B      removes breakpoint
   $B/     opens breakpoint status word
   ;P      proceeds from breakpoint, stops again on next
          encounter
   n;P     proceeds from breakpoint, stops again on nth
          encounter
   $M/     opens mask for word search
   n;W     searches for words which match n in bits specified
          in $M
   n;E     searches for words which address word n
   n/ (con-
tents) m;0  calculates offsets from n to m
   $S/     opens location containing user program's status
          register
   $P/     opens location containing ODT's priority level
DEBUGGING OBJECT PROGRAMS ON-LINE, ODT-11 AND ODT-11X

NOTE

If a word is currently open, new contents for the word may be typed followed by any of the commands RETURN, +, +, + or . The open word will be modified and closed before the new command is executed.

2. ODT-11X

In addition to the commands of the regular version, the extended version has the following:

n\  opens byte
\   reopens last byte opened
@   opens the absolutely addressed word
>   opens the word to which the branch refers
<   opens next location of previous sequence
n;rB  (r between 0 and 7) sets breakpoint r at word n
;rB   removes breakpoint r
;B    removes all breakpoints
$B/  opens breakpoint 0 status word. Successive LINE FEEDs open words for other breakpoints and single-instruction mode.
;nS   enables Single-instruction mode (n can have any value and is not significant)
n;P   in single-instruction mode, Proceeds with program run for next n instructions before reentering ODT (if n is missing, it is assumed to be 1)
;S    disables Single-instruction mode
APPENDIX F
LOADING AND DUMPING CORE MEMORY

F.1 THE BOOTSTRAP LOADER

This appendix pertains only to systems with a Switch Register.

F.1.1 Loading The Bootstrap Loader

The Bootstrap Loader should be toggled into the highest core memory bank.

```
x x 7 7 4 4  0 1 6 7 0 1
x x 7 7 4 6  0 0 0 0 2 6
x x 7 7 5 0  0 1 2 7 0 2
x x 7 7 5 2  0 0 0 3 5 2
x x 7 7 5 4  0 0 5 2 1 1
x x 7 7 5 6  1 0 5 7 1 1
x x 7 7 6 0  1 0 0 3 7 6
x x 7 7 6 2  1 1 6 1 6 2
x x 7 7 6 4  0 0 0 0 0 2
x x 7 7 6 6  x x 7 4 0 0
x x 7 7 7 0  0 0 5 2 6 7
x x 7 7 7 2  1 7 7 7 5 6
x x 7 7 7 4  0 0 0 7 6 5
x x 7 7 7 6  y y y y y y
```

xx represents the highest available memory bank. For example, the first location of the loader would be one of the following, depending on memory size, and xx in all subsequent locations would be the same as the first.

<table>
<thead>
<tr>
<th>Location</th>
<th>Memory Bank</th>
<th>Memory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>017744</td>
<td>0</td>
<td>4K</td>
</tr>
<tr>
<td>037744</td>
<td>1</td>
<td>8K</td>
</tr>
<tr>
<td>057744</td>
<td>2</td>
<td>12K</td>
</tr>
<tr>
<td>077744</td>
<td>3</td>
<td>16K</td>
</tr>
<tr>
<td>117744</td>
<td>4</td>
<td>20K</td>
</tr>
<tr>
<td>137744</td>
<td>5</td>
<td>24K</td>
</tr>
<tr>
<td>157744</td>
<td>6</td>
<td>28K</td>
</tr>
</tbody>
</table>

The contents of location xx7776 (yyyyyy) in the Instruction column above should contain the device status register address of the paper tape reader to be used when loading the bootstrap formatted tapes specified as follows:

Teletype Paper Tape Reader -- 177560

High-speed Paper Tape Reader -- 177550
LOADING AND DUMPING CORE MEMORY

Figure F-1 Loading and Verifying the Bootstrap Loader
Figure F-2  Loading Bootstrap Tapes into Core

F.2  THE ABSOLUTE LOADER

1. Loading the Absolute Loader

The Bootstrap Loader is used to load the Absolute Loader into core. (See Figure F-2.) The Absolute Loader occupies locations xx7474 through xx7743, and its starting address is xx7500.

2. Loading with the Absolute Loader

When using the Absolute Loader, there are three types of loads available: normal, relocated to specific address, and continued relocation.

Optional switch register settings for the three types of loads are listed below.
LOADING AND DUMPING CORE MEMORY

<table>
<thead>
<tr>
<th>Type of load</th>
<th>Switch Register</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Bits 1-14</td>
</tr>
<tr>
<td></td>
<td>(ignored)</td>
</tr>
<tr>
<td>Relocated - continue loading where left off</td>
<td>0</td>
</tr>
<tr>
<td>Relocated - load in specified area of core</td>
<td>nnnnn</td>
</tr>
<tr>
<td></td>
<td>(specified address)</td>
</tr>
</tbody>
</table>

F.3  CORE MEMORY DUMPS

The two dump programs are

DUMPTT, which dumps the octal representation of the contents of all or specified portions of core onto the teleprinter, low-speed or high-speed punch, or line printer.

DUMPAB, which dumps the absolute binary code of the contents of specified portions of core onto the low-speed (Teletype) or high-speed punch.

Both dumps are supplied on punched paper tape in bootstrap and absolute binary formats. The following figure summarizes loading and using the Absolute binary tapes.
LOADING AND DUMPING CORE MEMORY

Figure F-3  Loading with the Absolute Loader
LOADING AND DUMPING CORE MEMORY

Figure F-4 Dumping Using DUMPAB or DUMPTT
LOADING AND DUMPING CORE MEMORY

Figure F-4 (continued). Dumping Using DUMPAB or DUMPTT
APPENDIX G
INPUT/OUTPUT PROGRAMMING, IOX

G.1 INSTRUCTION SUMMARY

1. Format

IOT
.WORD   (an address)
.BYTE   (a command code, a slot number of a device)
.WORD   (done address) ;READR AND WRITR ONLY

2. Command Codes:

INIT   = 1
RESET  = 2
RSTRT  = 3
WAITR  = 4
SEEK   = 5
READ   = 11
WRITE  = 12
READR  = 13
WRITR  = 14

G.2 PROGRAM FLOW SUMMARY

1. Set up buffer header:

<table>
<thead>
<tr>
<th>Location</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Buffer and Buffer+1</td>
<td>Maximum number of data bytes (unsigned integer)</td>
</tr>
<tr>
<td>BUFFER HEADER</td>
<td>Mode of data (byte)</td>
</tr>
<tr>
<td>Buffer+3 and Buffer+4</td>
<td>Status of data (byte)</td>
</tr>
<tr>
<td>Buffer+5</td>
<td>Number of data bytes involved in transfer (unsigned integer)</td>
</tr>
<tr>
<td>Buffer+6</td>
<td>Actual data begins here.</td>
</tr>
</tbody>
</table>

Mode Byte Format

<table>
<thead>
<tr>
<th>Bits</th>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
<th>Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>NO ECHO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>UNFORMATTED</td>
</tr>
<tr>
<td>0</td>
<td>ECHO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FORMATTED</td>
</tr>
</tbody>
</table>

G-1
Coding Mode Byte

Formatted ASCII 0 (or 200 to suppress echo)
Formatted Binary 1
Unformatted ASCII 2 (or 202 to suppress echo)
Unformatted Binary 3

Status Byte Format

<table>
<thead>
<tr>
<th>7</th>
<th>6</th>
<th>5</th>
<th>4</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 = DONE</td>
<td>1 = EOM</td>
<td>1 = EOF</td>
<td>SEE CODES</td>
<td>NON-FATAL ERRORS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Coding Non-Fatal Errors

2  =  checksum error (formatted binary)
3  =  truncation of a long line
4  =  an improper mode

2. Assign devices to slots in Device Assignment Table:

(RESET and INIT commands)

Slot numbers are in the range 0 to 7.

Device Codes:

- KBD = 1
- TTY = 2
- LSR = 3
- LSP = 4
- HSR = 5
- LPT = 10
- HSP = 6

3. Use a data transfer command to initiate the transfer.

G.3 FATAL ERRORS

Fatal errors result in a jump to 40 with R0 set to the error code.
R1 is set to the value of the PC for error code 0. Errors 1-5 cause
R1 to be set to an IOT argument or to the instruction following the
arguments.

<table>
<thead>
<tr>
<th>Fatal Error Code</th>
<th>Reason</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Illegal Memory Reference, SP overflow, illegal instruction</td>
</tr>
<tr>
<td>1</td>
<td>Illegal command</td>
</tr>
<tr>
<td>2</td>
<td>Slot out of range</td>
</tr>
<tr>
<td>3</td>
<td>Device out of range</td>
</tr>
<tr>
<td>4</td>
<td>Slot not init</td>
</tr>
<tr>
<td>5</td>
<td>Illegal data mode</td>
</tr>
<tr>
<td></td>
<td>G-2</td>
</tr>
</tbody>
</table>

APPENDIX H

SUMMARY OF FLOATING POINT MATH PACKAGE, FPMP-11

This appendix lists all the global entry points of FPMP-11 and provides a brief description of the purposes of each. Sections H.1 and H.2 are for reference when it is desired to call FPMP-11 routines directly (i.e., without the use of the TRAP handler). Entry names preceded by an octal number can be referenced via the TRAP handler. The number is the "routine number" referred to in the FPMP-11 manual. If the number is enclosed in parentheses, the routine cannot be accessed by the present TRAP handler, but has been assigned a number for future use. For a more detailed explanation of the Floating Point Math Package, refer to the FPMP-11 User's Manual DEC-11-NFPMA-A-D.

Examples of the calling conventions are:

POLISH MODE:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR R4,$POLISH</td>
<td>enter Polish mode</td>
</tr>
<tr>
<td>$subr1</td>
<td>call desired subroutines</td>
</tr>
<tr>
<td>$subr2</td>
<td></td>
</tr>
<tr>
<td>$subrn</td>
<td>call last subroutine desired</td>
</tr>
<tr>
<td>.WORD .+2</td>
<td>leave Polish mode</td>
</tr>
</tbody>
</table>


J5RR:

<table>
<thead>
<tr>
<th>Statement</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>JSR R5,subr</td>
<td>call desired subroutine</td>
</tr>
<tr>
<td>BR XX</td>
<td></td>
</tr>
<tr>
<td>.WORD arg1</td>
<td>subroutine argument address</td>
</tr>
<tr>
<td>.WORD arg2</td>
<td></td>
</tr>
<tr>
<td>XX:</td>
<td></td>
</tr>
<tr>
<td>.WORD argn</td>
<td>last argument</td>
</tr>
<tr>
<td>.</td>
<td>return point</td>
</tr>
</tbody>
</table>


H-1
SUMMARY OF FLOATING POINT MATH PACKAGE, FPMP-11

JPC:
.
.
push args onto stack
JMP PC, subr
.
.
H.1 OTS ROUTINES

These are the routines taken from the FORTRAN operating time system. The codes used in the following table are:

S = Routine is included in the standard single precision (2-word) package.
D = Routine is included in the standard double precision (4-word) package.
SD = Routine is included in both standard packages.

Octal codes shown in parentheses are not yet implemented.

<table>
<thead>
<tr>
<th>NAME</th>
<th>OCTAL CODE</th>
<th>PKG</th>
<th># OF ARGU</th>
<th>MODE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ADD</td>
<td>14</td>
<td>D</td>
<td>2</td>
<td>Polish</td>
<td>The double precision add routine. Adds the top stack item (4-words) to the second item (4-words) and leaves the four word sum in their place.</td>
</tr>
<tr>
<td>$ADR</td>
<td>12</td>
<td>S</td>
<td>2</td>
<td>Polish</td>
<td>The single precision add routine. Same as $ADD except it uses 2 word numbers.</td>
</tr>
<tr>
<td>AINT</td>
<td>26</td>
<td>S</td>
<td>1</td>
<td>J5RR</td>
<td>Returns sign of argument * greatest real integer = absolute value of the argument in R0,R1.</td>
</tr>
<tr>
<td>ALOG</td>
<td>53</td>
<td>S</td>
<td>1</td>
<td>J5RR</td>
<td>Calculates natural logarithm of its single argument and returns a two word result in R0,R1.</td>
</tr>
<tr>
<td>ALOG10</td>
<td>54</td>
<td>S</td>
<td>1</td>
<td>J5RR</td>
<td>Same as ALOG, except calculates base-10 logarithm.</td>
</tr>
<tr>
<td>ATAN</td>
<td>42</td>
<td>S</td>
<td>1</td>
<td>J5RR</td>
<td>Returns the arctangent of its argument in R0,R1.</td>
</tr>
<tr>
<td>ATAN2</td>
<td>(43)</td>
<td>S</td>
<td>2</td>
<td>J5RR</td>
<td>Returns ARCTAN (ARGL/ARG2) in R0,R1.</td>
</tr>
<tr>
<td>$CMD</td>
<td>16</td>
<td>D</td>
<td>2</td>
<td>Polish</td>
<td>Compares top 4 word items on the stack, flushes the two items, and returns the following condition codes: 4(SP) $SP N=1,Z=0 4(SP) = $SP N=0,Z=1 4(SP) $SP N=0,Z=0</td>
</tr>
</tbody>
</table>
SUMMARY OF FLOATING POINT MATH PACKAGE, FPMP-11

<table>
<thead>
<tr>
<th>Command</th>
<th>Code</th>
<th>Type</th>
<th>Precision</th>
<th>Mode</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$CMR</td>
<td>17</td>
<td>S</td>
<td>2</td>
<td>Polish</td>
<td>Same as $CMD except it is for 2 word arguments.</td>
</tr>
<tr>
<td>COS</td>
<td>37</td>
<td>S</td>
<td>1</td>
<td>J5RR</td>
<td>Single precision version of DCOS.</td>
</tr>
<tr>
<td>DATAN</td>
<td>44</td>
<td>D</td>
<td>1</td>
<td>J5RR</td>
<td>Double precision version of ATAN.</td>
</tr>
<tr>
<td>DATAN2</td>
<td>(45)</td>
<td>D</td>
<td>2</td>
<td>J5RR</td>
<td>Double precision version of ATAN2.</td>
</tr>
<tr>
<td>DBLE</td>
<td>(34)</td>
<td></td>
<td></td>
<td>J5RR</td>
<td>Returns in R0-R3 the double precision equivalent of the single precision (two word) argument.</td>
</tr>
<tr>
<td>$DCI</td>
<td>(57)</td>
<td>SD</td>
<td>4</td>
<td>JPC</td>
<td>ASCII to double conversion. Calling sequence: Push address of start of ASCII field. Push length of ASCII field in bytes. Push format scale D (from W.D) position of assumed decimal point (see FORTRAN manual). Push P format scale (see FORTRAN manual). JSR PC,$DCI. Returns 4 word result on top of stack.</td>
</tr>
<tr>
<td>$DCO</td>
<td>(61)</td>
<td>SD</td>
<td>5</td>
<td>JPC</td>
<td>Double precision to ASCII conversion. Calling sequence: Push address of start of ASCII field. Push length in bytes of ASCII field (W part of W.D) Push D part of W.D position of decimal point). Push P scale. Push 4 word value to be converted, lowest order word first. JSR PC,$DCO.</td>
</tr>
<tr>
<td>DCOS</td>
<td>41</td>
<td>D</td>
<td>1</td>
<td>J5RR</td>
<td>Calculates the cosine of its double precision argument and returns the double precision result in R0-R3.</td>
</tr>
<tr>
<td>DEXP</td>
<td>52</td>
<td>D</td>
<td>1</td>
<td>J5RR</td>
<td>Calculates the exponential of its double precision argument, and returns the double precision result in R0-R3.</td>
</tr>
<tr>
<td>$DI</td>
<td>(11)</td>
<td>SD</td>
<td></td>
<td>Polish</td>
<td>Converts double precision number on the top of the stack to integer. Leaves result on stack.</td>
</tr>
<tr>
<td>$DINT</td>
<td>(76)</td>
<td>D</td>
<td>1</td>
<td>Polish</td>
<td>OTS internal function to find the integer part of a double precision number.</td>
</tr>
</tbody>
</table>

H-3
<table>
<thead>
<tr>
<th>Command</th>
<th>OPN</th>
<th>Operands</th>
<th>Flags</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLOG</td>
<td>55</td>
<td>D 1</td>
<td>J5RR</td>
<td>Double precision (4 word) version of ALOG.</td>
</tr>
<tr>
<td>DLOG10</td>
<td>56</td>
<td>D 1</td>
<td>J5RR</td>
<td>Double precision (4 word) version of ALOG10.</td>
</tr>
<tr>
<td>$DR</td>
<td>(6)</td>
<td></td>
<td>Polish</td>
<td>Replaces the double precision item at the top of the stack with its two word, rounded form.</td>
</tr>
<tr>
<td>DSIN</td>
<td>40</td>
<td>D 1</td>
<td>J5RR</td>
<td>Calculates the sine of its double precision arg. and returns the double precision result in R0-R3.</td>
</tr>
<tr>
<td>DSQRT</td>
<td>47</td>
<td>D 1</td>
<td>J5RR</td>
<td>Calculates the square root of its double precision arg. and returns the double precision result in R0-R3.</td>
</tr>
<tr>
<td>$DVD</td>
<td>23</td>
<td>D 2</td>
<td>Polish</td>
<td>The double precision division routine. Divides the second 4-word item on the stack by the top item and leaves the quotient in their place.</td>
</tr>
<tr>
<td>$DVI</td>
<td>(24)</td>
<td></td>
<td>Polish</td>
<td>The integer division routine. Calculates ( \frac{2(\text{SP})}{\text{SP}} ) and returns the integer quotient on the top of the stack.</td>
</tr>
<tr>
<td>$DVR</td>
<td>25</td>
<td>S 2</td>
<td>Polish</td>
<td>The single precision division routine. Same as $DVD, but for 2 word floating point numbers.</td>
</tr>
<tr>
<td>$ECO</td>
<td>(62)</td>
<td>SD 5</td>
<td>JPC</td>
<td>Single precision to ASCII conversion according to E format. Same calling sequence as $DCO except that a 2-word value is to be converted.</td>
</tr>
<tr>
<td>EXP</td>
<td>51</td>
<td>S 1</td>
<td>J5RR</td>
<td>Single precision version of DEXP. Returns result in R0,R1.</td>
</tr>
<tr>
<td>$FCALL</td>
<td>-</td>
<td>S</td>
<td></td>
<td>Internal OTS routine.</td>
</tr>
<tr>
<td>$FCO</td>
<td>(64)</td>
<td>SD 5</td>
<td>JPC</td>
<td>Same as $ECO except uses F format conversion.</td>
</tr>
<tr>
<td>FLOAT</td>
<td>(32)</td>
<td></td>
<td>J5RR</td>
<td>Returns in R0-R1, the real equivalent of its integer argument.</td>
</tr>
<tr>
<td>$GCO</td>
<td>(63)</td>
<td>SD 5</td>
<td>JPC</td>
<td>Same as $ECO except uses G format conversion.</td>
</tr>
</tbody>
</table>
### SUMMARY OF FLOATING POINT MATH PACKAGE, FPMP-11

<table>
<thead>
<tr>
<th>INSTRUCTION</th>
<th>ARGUMENTS</th>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>$ICI</strong></td>
<td>(65)</td>
<td>2 JPC</td>
</tr>
<tr>
<td><strong>$ICO</strong></td>
<td>(67)</td>
<td>3 JPC</td>
</tr>
<tr>
<td><strong>IDINT</strong></td>
<td>(31)</td>
<td>1 J5RR</td>
</tr>
<tr>
<td><strong>$ID</strong></td>
<td>(5)</td>
<td>SD 1 Polish</td>
</tr>
<tr>
<td><strong>IFIX</strong></td>
<td>(35)</td>
<td>1 J5RR</td>
</tr>
<tr>
<td><strong>INT</strong></td>
<td>(30)</td>
<td>1 J5RR</td>
</tr>
<tr>
<td><strong>$INTR</strong></td>
<td>(27)</td>
<td>S 1 Polish</td>
</tr>
<tr>
<td><strong>$IR</strong></td>
<td>(4)</td>
<td>SD 1 Polish</td>
</tr>
<tr>
<td><strong>$MLD</strong></td>
<td>22</td>
<td>D 2 Polish</td>
</tr>
<tr>
<td><strong>$MLI</strong></td>
<td>(20)</td>
<td>2 Polish</td>
</tr>
<tr>
<td><strong>$MLR</strong></td>
<td>21</td>
<td>S 2 Polish</td>
</tr>
<tr>
<td><strong>$NGNU</strong></td>
<td>(3)</td>
<td>SD 2 Polish</td>
</tr>
<tr>
<td>Command</td>
<td>Call</td>
<td>Type</td>
</tr>
<tr>
<td>---------</td>
<td>------</td>
<td>------</td>
</tr>
<tr>
<td>$NGI</td>
<td>(1)</td>
<td>SD</td>
</tr>
<tr>
<td>$NGR</td>
<td>(2)</td>
<td>SD</td>
</tr>
<tr>
<td>$OCI</td>
<td>(66)</td>
<td></td>
</tr>
<tr>
<td>$OCO</td>
<td>(70)</td>
<td></td>
</tr>
<tr>
<td>$POLISH</td>
<td>-</td>
<td>SD</td>
</tr>
<tr>
<td>$POPR3</td>
<td>-</td>
<td>D</td>
</tr>
<tr>
<td>$POPR4</td>
<td>-</td>
<td>D</td>
</tr>
<tr>
<td>$POPR5</td>
<td>-</td>
<td>D</td>
</tr>
<tr>
<td>$PSHR1</td>
<td>-</td>
<td>SD</td>
</tr>
<tr>
<td>$PSHR2</td>
<td>-</td>
<td>SD</td>
</tr>
<tr>
<td>$PSHR3</td>
<td>-</td>
<td>SD</td>
</tr>
<tr>
<td>$PSHR4</td>
<td>-</td>
<td>SD</td>
</tr>
<tr>
<td>$PSHR5</td>
<td>-</td>
<td>SD</td>
</tr>
<tr>
<td>$RCI</td>
<td>(60)</td>
<td>SD</td>
</tr>
<tr>
<td>$RD</td>
<td>(7)</td>
<td>Polish</td>
</tr>
<tr>
<td>$RI</td>
<td>(10)</td>
<td>SD</td>
</tr>
<tr>
<td>$SBD</td>
<td>15</td>
<td>D</td>
</tr>
</tbody>
</table>
SUMMARY OF FLOATING POINT MATH PACKAGE, FPMP-11

<table>
<thead>
<tr>
<th>NAME</th>
<th>OCTAL CODE</th>
<th>PKG</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$SBR</td>
<td>13</td>
<td>S</td>
<td>Polish Same as $SBD but for single precision.</td>
</tr>
<tr>
<td>SIN</td>
<td>36</td>
<td>S</td>
<td>1 J5RR Single precision version of DSIN.</td>
</tr>
<tr>
<td>SNGL</td>
<td>(33)</td>
<td></td>
<td>1 J5RR Rounds double precision argument to single precision. Returns result in R0,R1.</td>
</tr>
<tr>
<td>SQRT</td>
<td>46</td>
<td>S</td>
<td>1 J5RR Single precision version of DSQRT.</td>
</tr>
<tr>
<td>TANH</td>
<td>50</td>
<td>S</td>
<td>1 J5RR Single precision hyperbolic tangent function. Returns (\frac{\exp(2*\text{ARG})-1}{\exp(2*\text{ARG})+1}) in R0,R1.</td>
</tr>
</tbody>
</table>

H.2 NON-OTS ROUTINES

These routines are written especially for FPMP-11 and should not be called directly by the user.

<table>
<thead>
<tr>
<th>NAME</th>
<th>OCTAL CODE</th>
<th>PKG</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>$ERR</td>
<td>-</td>
<td>SD</td>
<td>Internal error handler.</td>
</tr>
<tr>
<td>$ERRA</td>
<td>-</td>
<td>SD</td>
<td>Similar to $ERR.</td>
</tr>
<tr>
<td>$LDR</td>
<td>71</td>
<td>S</td>
<td>Load FLAC, single precision.</td>
</tr>
<tr>
<td>$LDD</td>
<td>72</td>
<td>D</td>
<td>Load FLAC, double precision.</td>
</tr>
<tr>
<td>$STR</td>
<td>73</td>
<td>S</td>
<td>Store FLAC, single precision.</td>
</tr>
<tr>
<td>$STD</td>
<td>74</td>
<td>D</td>
<td>Store FLAC, double precision.</td>
</tr>
<tr>
<td>TRAPH</td>
<td>-</td>
<td>SD</td>
<td>The TRAP handler routines and tables.</td>
</tr>
</tbody>
</table>

H.3 ROUTINES ACCESSED VIA TRAP HANDLER

The following is a table of the FPMP-11 routines which can be accessed via TRAPH, the trap handler. Each routine name (entry point) is preceded by its TRAP code number to be used to access it, and followed by a brief description of its operation when called via the TRAP handler. Those entries which are preceded by an asterisk (*) perform operations only on the FLAC, and address no operands. For example, a TRAP call to the single precision square root routine can be coded as follows:

```
.
.
.
TRAP 46
.
.
```
The net effect of the above TRAP instruction is to replace the contents of the FLAC with its square root and then set the condition codes to reflect the result. Note that since the FLAC is implicitly addressed in this instruction, the TRAP call supplies no other address. For such a TRAP call, the addressing mode bits (bits 6 and 7 of the TRAP instruction) are ignored.

All entries not marked by an asterisk require an operand when called. The operand is addressed in one of the four addressing modes explained in section 3.1.1. of the FPMP-11 User's Manual. The addressing mode is specified in bit 6-7 of the TRAP instruction.

("Operand" is the contents of the location addressed in the TRAP call.)

<table>
<thead>
<tr>
<th>OCTAL CODE</th>
<th>NAME</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>14</td>
<td>$ADD</td>
<td>Double precision addition routine. Adds operand to the FLAC. Assumes 4-word operand.</td>
</tr>
<tr>
<td>12</td>
<td>$ADR</td>
<td>Single precision addition routine. Adds operand to the FLAC. Assumes 2-word operand.</td>
</tr>
<tr>
<td>*</td>
<td>26</td>
<td>AINT</td>
</tr>
<tr>
<td>*</td>
<td>53</td>
<td>ALOG</td>
</tr>
<tr>
<td>*</td>
<td>54</td>
<td>ALOG10</td>
</tr>
<tr>
<td>*</td>
<td>42</td>
<td>ATAN</td>
</tr>
<tr>
<td>16</td>
<td>$CMD</td>
<td>Compares operand to the contents of the FLAC, and returns the following condition codes.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{FLAC&lt;operand, N=1, Z=0} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{FLAC=operand, N=0, Z=1} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(\text{FLAC&gt;operand, N=0, Z=0} )</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Assumes 4-word operands.</td>
</tr>
<tr>
<td>17</td>
<td>$CMR</td>
<td>Same as $CMD, but for 2-word operands.</td>
</tr>
<tr>
<td>*</td>
<td>37</td>
<td>COS</td>
</tr>
<tr>
<td>*</td>
<td>44</td>
<td>DATAN</td>
</tr>
<tr>
<td>*</td>
<td>52</td>
<td>DEXP</td>
</tr>
<tr>
<td>No.</td>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>-----</td>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td>55</td>
<td>DLOG</td>
<td>Same as ALOG, but for 4-word argument.</td>
</tr>
<tr>
<td>56</td>
<td>DLOG10</td>
<td>Same as ALOG10, but for 4-word argument.</td>
</tr>
<tr>
<td>41</td>
<td>DCOS</td>
<td>Replaces the contents of the FLAC by its cosine. Assumes 4-word argument in the FLAC.</td>
</tr>
<tr>
<td>40</td>
<td>DSIN</td>
<td>Same as DCOS, but calculates sine instead of cosine.</td>
</tr>
<tr>
<td>47</td>
<td>DSQRT</td>
<td>Replaces the contents of the FLAC by its square root. Assumes 4-word argument in the FLAC.</td>
</tr>
<tr>
<td>23</td>
<td>$DVD</td>
<td>Double precision division routine. Divides the FLAC by the operand and stores the result in the FLAC. Assumes 4-word operands.</td>
</tr>
<tr>
<td>25</td>
<td>$DVR</td>
<td>Same as $DVD, but for 2-word operands.</td>
</tr>
<tr>
<td>51</td>
<td>EXP</td>
<td>Same as DEXP, but for 2-word argument.</td>
</tr>
<tr>
<td>72</td>
<td>$LDD</td>
<td>Same as $LDR, but assumes 4-word operand.</td>
</tr>
<tr>
<td>71</td>
<td>$LDR</td>
<td>Replaces the contents of the FLAC by the operand. Assumes 2-word operand.</td>
</tr>
<tr>
<td>22</td>
<td>MLD</td>
<td>Double precision multiplication routine. Multiplies the contents of the FLAC by the operand and stores the result in the FLAC. Assumes 4-word operands.</td>
</tr>
<tr>
<td>21</td>
<td>$MLR</td>
<td>Same as $MLD, but for 2-word operands.</td>
</tr>
<tr>
<td>15</td>
<td>$SBD</td>
<td>The double precision subtraction routine. Subtracts the operand from the contents of the FLAC. Assumes a 4-word operand.</td>
</tr>
<tr>
<td>13</td>
<td>$SBR</td>
<td>Same as $SBD, but for 2-word operand.</td>
</tr>
<tr>
<td>36</td>
<td>SIN</td>
<td>Same as DSIN, but for 2-word argument.</td>
</tr>
<tr>
<td>46</td>
<td>SQRT</td>
<td>Same as DSQRT, but for 2-word argument.</td>
</tr>
<tr>
<td>73</td>
<td>$STR</td>
<td>Stores the contents of the FLAC into the operand location. The contents of the FLAC are unchanged.</td>
</tr>
<tr>
<td>74</td>
<td>$STD</td>
<td>Same as $STR, but assumes 4-word operand location.</td>
</tr>
<tr>
<td>50</td>
<td>TANH</td>
<td>Replaces the contents of the FLAC by its hyperbolic tangent. Assumes 2-word argument.</td>
</tr>
</tbody>
</table>
APPENDIX I

TAPE DUPLICATION

Duplication of paper tapes can be accomplished via low- or high-speed I/O devices by toggling (as with the Bootstrap Loader) the following program directly into memory through the Switch Register. (Refer to Section 6.1.1 in Chapter 6 if necessary, for toggling procedure.)

1. Turn on appropriate device switches and place tape in desired reader.

2. Set ENABLE/HALT switch to HALT.

3. Set Switch Register to the desired starting address and press LOAD ADDR.

4. Set Switch Register to each value listed in the CONTENTS column below, lifting the DEP switch after each setting. (Addresses are automatically incremented.) The desired input device (either Low- or High-Speed Reader) and output device (Low- or High-Speed Punch) are specified in the last two words.

<table>
<thead>
<tr>
<th>ADDRESS</th>
<th>CONTENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>016700</td>
</tr>
<tr>
<td>2</td>
<td>000024</td>
</tr>
<tr>
<td>4</td>
<td>016701</td>
</tr>
<tr>
<td>6</td>
<td>000022</td>
</tr>
<tr>
<td>10</td>
<td>005210</td>
</tr>
<tr>
<td>12</td>
<td>105710</td>
</tr>
<tr>
<td>14</td>
<td>100376</td>
</tr>
<tr>
<td>16</td>
<td>105711</td>
</tr>
<tr>
<td>20</td>
<td>100376</td>
</tr>
<tr>
<td>22</td>
<td>022021</td>
</tr>
<tr>
<td>24</td>
<td>111011</td>
</tr>
<tr>
<td>26</td>
<td>000764</td>
</tr>
<tr>
<td>30</td>
<td>177560 (LSR) or 177550 (HSR)</td>
</tr>
<tr>
<td>32</td>
<td>177564 (LSP) or 177554 (HSP)</td>
</tr>
</tbody>
</table>

5. Set Switch Register to starting address specified in 3 above and press LOAD ADDR.

6. Set ENABLE/HALT switch to ENABLE.

7. Press START switch.
NOTE

This program is recommended as a simple way of duplicating the system tapes. However, for extensive tape duplication, the program shown in section 7.8 is recommended.
APPENDIX J

ASSEMBLY AND LINKING INSTRUCTIONS

J.1 SYSTEMS WITHOUT SWITCH REGISTERS

J.1.1 IOX/IOXLPT

IOX/IOXLPT is provided in both source and relocatable object form. Unless modifications are made to the source it is not necessary to assemble the source tapes. The object tape may be linked with the user's object tapes to produce an absolute tape (.LDA).

J.1.1.1 Assembling IOX - IOX consists of three source tapes (-PAL to -PA3). These tapes are assembled together in sequence with PAL-11S.

J.1.1.2 Assembling IOXLPT - IOXLPT consists of two source tapes (-PAL to PA2). These tapes are assembled together in sequence with PAL-11S.

J.1.1.3 Linking IOX and IOXLPT - IOX and IOXLPT are linked by LINK-11S with the user's object tapes to produce an absolute tape.

J.1.2 ODT11X

ODT11X is provided in both source and relocatable object form. Unless modifications are made to the source, it is not necessary to assemble the source tape. The object tape may be linked with the user's object tapes to produce an absolute tape.

J.1.2.1 Assembling ODT11X - ODT11X consists of one source tape (-PAL) which is terminated with the following:

```
.EOT
form feed
.END O.ODT
```

When PAL-11S indicates that it has encountered the .EOT, type return so that it will process the .END statement.
ASSEMBLY AND LINKING INSTRUCTIONS

J.1.2.2 Linking ODT11X - ODT11X is linked with user object tapes. It is self starting and should be the first object tape input to LINK-11S so that it will be the program started by the Absolute Loader when the program is loaded.

J.1.3 ED-11

The ED-11 source file is available only in RT-11 format on a flexible diskette. The RT-11 MACRO assembler is required to assemble ED-11. The RT-11 linker (LINK) is used to produce the absolute tape.

J.1.3.1 Assembling ED-11 - The RT-11 commands to assemble ED-11 are as follows:

```
.R MACRO
*EDIT11=DX1:EDIT11
```

J.1.3.2 Linking ED-11 - The RT-11 commands to link ED-11 are as follows:

```
.R LINK
*PP:EDIT11/L=EDIT11
```

J.1.4 PAL-11S

The PAL-11S source file is available only in RT-11 format on a flexible diskette. The RT-11 MACRO assembler is required to assemble PAL-11S. The RT11 linker (LINK) or LINK-11S may be used to produce the absolute tape.

J.1.4.1 Assembling PAL-11S - There are three sources which are assembled separately for PAL-11S. One of these, the symbol table source, is available in three versions: 8K, 12K, and 16K. The RT-11 commands to assemble PAL-11S source files are as follows:

```
.R MACRO
*RELMEM=DX1:RELMEM.PAL Clear Memory Program
*PSYM08=DX1:PSYM08.PAL 8K Symbol Table
*PSYM12=DX1:PSYM12.PAL 12K Symbol Table
*PSYM16=DX1:PSYM16.PAL 16K Symbol Table
*PAL11S=DX1:PAL11S.PAL Assembler
```

In addition to the above, IOXLPT is used by PAL-11S. The IOXLPT source is also available in RT-11 format on a flexible diskette. The commands to assemble IOXLPT are:

```
.R MACRO
*IOXLPT=DX1:IOXLPT.PAL
```
ASSEMBLY AND LINKING INSTRUCTIONS

J.1.4.2  Linking PAL-11S - PAL-11S may be linked with LINK-11S or the RT-11 linker, LINK. The PAL-11S tape actually contains two programs: RELMEM and PAL-11S. RELMEM precedes PAL-11S on the tape.

Using LINK-11S, link PAL-11S as follows:

1. Link RELMEM as a separate program and do not remove the tape from the punch when finished.

2. Link PAL11S.OBJ, IOXLPT.OBJ, and one of the symbol table object tapes (PSYM08.OBJ, PSYM12.OBJ, or PSYM16.OBJ) in that order. The symbol table tape is selected depending on the size of the memory of the computer on which the program is to be executed. If the target computer has 8K words of memory then PSYM08.OBJ, if 12K then PSYM12.OBJ, and if 16K then PSYM16.OBJ. Specify a top address of 57460 for 12K and 77460 for 16K.

Do not link PAL-11S to run above 16K. The size of the symbol table is fixed, and there is no need to re-link at a higher address even on large systems.

Using RT-11 LINK, link PAL-11S as follows:

1. Link RELMEM as a separate program as shown

   .R LINK
   *RELMEM/L=RELMEM

2. Link 8K, 12K, and 16K versions of PAL-11S

   .R LINK
   *PAL08/L/B:204=PAL11S,IOXLPT,PSYM08
   *PAL12/L/B:204=PAL11S,IOXLPT,PSYM12
   *PAL16/L/B:204=PAL11S,IOXLPT,PSYM16

3. Use RT-11 PIP to punch the tapes. Remember not to remove the tape from the punch after punching RELMEM.

   .R PIP
   *PP:=RELMEM.LDA/B
   *PP:=PAL08.LDA/B

   remove 8K PAL11S.LDA from punch

   *PP:=RELMEM.LDA/B
   *PP:=PAL12.LDA/B

   remove 12K PAL11S.LDA from punch

   *PP:=RELMEM.LDA/B
   *PP:=PAL16.LDA/B

J.1.5  LINK-11S

The LINK-11S source file is available only in RT-11 format on a flexible diskette. The RT-11 MACRO assembler is required to assemble LINK-11S. LINK-11S is composed of two components: LINK-11S proper and IOXLPT. See Section N.1.4.1 for instructions on how to assemble IOXLPT using RT-11.
ASSEMBLY AND LINKING INSTRUCTIONS

J.1.5.1  Assembling LINK-11S - The RT-11 commands to assemble LINK-11S follow:

    .R MACRO
    *LINK11=DX1:LINC11

J.1.5.2  Linking LINK-11S - LINK-11S may be linked with LINK-11S or the RT-11 linker, LINK. There are two object tapes which are linked together to produce LINK-11S: LINK11.OBJ and IOXLPT.OBJ.

Using LINK-11S to link LINK-11S, link the following two tapes in order: LINK11.OBJ and IOXLPT. If versions are desired for systems with more than 8K, specify a top address of 57460 for 12K and 77460 for 16K.

Using RT-11 LINK to link LINK-11S is a two step process because of a difference in philosophy. An initial link is required which produces a link map so that the size of LINK11S can be determined. A final link is then made with the information obtained in the initial link used to compute the bottom address.

The initial link is executed as follows:

    .R LINK
    */TT:=LINK11,IOXLPT

The value displayed for "HIGH LIMIT" is used to compute the bottom address for the final link. Assume for an example that the following was displayed:

    HIGH LIMIT = 015572

Select 37460, 57460, or 77460 depending on whether an 8K, 12K, or 16K top address is desired. The bottom address is computed as follows:

    B = T - 4 + 1000

Where:
B = bottom address
T = top address
H = high limit

Example:  B = 37460-15572+1000
          B = 22666

Using the figures in the example above, the final link for an 8K system would be executed as follows:

    .R LINK
    */PP:/B=22666/L,TT:=LINK11,IOXLPT

As a check, examine the link map produced and verify that the high limit matches the one used in the calculations above. In the example, the high limit value must be 37460.
J.2 SYSTEMS WITH SWITCH REGISTERS

J.2.1 Assembling PAL-11A

The following procedures are for assembling the PAL-11 Assembler source tapes. An 8K version of the PAL-11A (V007A) Assembler is required, thus also requiring at least an 8K PDP-11 system.

The Assembler consists of two programs. The first program, on tape 1, is a memory clear program and is very short (DEC-11-UPLA-A-PA1). The second program is the Assembler proper, and consists of eleven ASCII tapes (DEC-11-UPLA-A-PA2-PA12). They are assembled as follows:

1. Generate a sufficient amount of blank leader tape.

2. Assemble the memory clear program source tape (DEC-11-UPLA-A-PA1) and assign the binary output to the high-speed punch. For example, PAL-11A's initial dialogue to specify the 2-pass assembly would be:

   *S H
   *R H/E
   *L
   \*

   END?

   (PAL assembly - 1st pass)

   00000000 ERRORS
   C

   (PAL assembly - 2nd pass)

   (No errors - Do not remove the binary tape from the punch.)

3. Assemble the rest of the Assembler's source tapes (PA2 - PA12) in numerical sequence.

   Assign the binary output to the high-speed punch. For example, the initial dialogue should be answered as follows:

   *S H
   *R H/E
   *L
   \*

   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?

   MAXCL3 = ***** SIMBC = *****

   (End of first pass)

   END ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?
   EOF ?

   (Enter tape PA2 for 2nd pass)
ASSEMBLY AND LINKING INSTRUCTIONS

000000 ERRORS (End of 2nd pass)

Note that at the end of the first pass there are two undefined symbols: MAXCL3 and SIMBC. These undefined symbols are resolved so that there are no errors reported during the second pass.

Be sure that there is sufficient blank trailer tape on the binary output tape before removing the assembled tape from the punch.

Normally, using high-speed paper tape input and output, this process requires about 45 minutes. If a symbol table and listing are requested, there will be about 750 symbols and about 4500 lines of listing.

J.2.2 Assembling ED-11

ED-11 consists of five source tapes (PA1 to PA5) which are assembled together in sequence with 8K PAL-11A.

J.2.3 ODT-11/ODT-11X

In subsequent discussion, reference to ODT applies to both versions. ODT is supplied on both source and absolute binary tapes.

If the program being debugged requires storage where the version of ODT being used is normally loaded, it is necessary to reassemble ODT after changing the starting location.

The source tape of ODT is in three segments, each separated from the next by blank tape. The first segment contains:

```
.-n (standard location setting statement)
.EOT
```

where n=13026 for ODT-11 or n=12054 for ODT-11X. This statement tells the Assembler to start assembling at address n. To relocate ODT to another starting address, substitute for segment one a source tape consisting of:

```
.-n (n is the new load address for ODT)
.EOT
```

The .EOT statement tells the Assembler that this is the end of the segment but not the end of the program -- the Assembler will stop and wait for another tape to be placed in the reader.

The second segment of tape contains the ODT source program. This segment is also terminated with .EOT.

The third segment of the tape consists of the statement:

```
.END O.ODT
```

where .END means "end of program" and 0.ODT represents the starting address of the program (see Section 6.2.3).

When relocating ODT, the first segment of the source tape must be changed to reflect the desired load address. The third segment may be changed to .END without a start address. The latter will cause the Loader to halt upon completion of loading.
ASSEMBLY AND LINKING INSTRUCTIONS

The segmentation allows the following assembly forms:

1. Assemble alone but at a new address. A new segment one must be generated and assembled with segments two and three.

2. Assemble immediately after the user's program to be debugged. Assemble the tape of the user's program (ending with .EOT) followed by ODT's segment two and either segment three or a new segment three.

3. Assemble inside the program to be debugged. Assemble the first part of the user program (ending with .EOT) followed by ODT's second segment followed by the second part of the user program.

When setting locations before assembling, it must be noted that immediately preceding ODT a minimum internal stack of 408 bytes is required for the ODT-11 and 1168 bytes is required for ODT-11X. Additional room must be allocated for subroutine calls and possible interrupts while ODT is in control. Twelve bytes maximum will be used by ODT proper for subroutine calls and interrupts, giving a minimum safe stack space of 528 bytes for ODT-11 or 1308 bytes for ODT-11X.

Once a new binary tape of ODT has been assembled, load it using the Absolute Loader as explained in Section 6.2.2. Normally, the program to be debugged is loaded before ODT, since ODT will automatically be in control immediately after loading, unless the third segment of ODT's source tape was altered before assembly. As soon as the tape is read in, ODT will print an * on the Teletype to indicate that it is ready for a command.

J.2.4 Assembling IOX/IOXLPT

In subsequent discussion, reference to IOX applies to both versions. IOX is supplied on both source and absolute binary tapes.

If there is more than 4K of core available and it is desired to load IOX (or IOXLPT) in other than the normal location, IOX must be reassembled.

The code

```
  .=15100
  .EOT
```

appears at the beginning of the first IOX tape (PA1) and contains the starting address. Create a new tape containing the new starting address desired; be sure to allow enough room for 63410 words for IOX, 72510 for IOXLPT. For example,

```
  .=25100
  .EOT
```

Use PAL-11A to assemble IOX and substitute the new section of tape for the first part of the old tape (PA1). After the new section is read, insert the IOX tape in the reader so the read head is past the old starting address and .EOT and type the RETURN key to read in the rest of the tape.

Now read in the second tape (PA2). An EOF? message is output at the end of the second tape. Type the RETURN key and the END? message is printed. Put the tapes through for the second pass of the assembler.
ASSEMBLY AND LINKING INSTRUCTIONS

IOX (IOXLPT) can also be assembled with a user program if desired. The .E = 15100 and .EOT lines must be deleted before IOX is assembled with a user program.

IOX can be assembled into the program wherever desired but if it is the first tape read by the assembler, remove it from the reader before typing the RETURN key (after the EOF? message of the second tape. IOX and IOXLPT have a .END code which would cause the assembly pass to end when read). Assembling a user program and IOX together eliminates the need to read in IOX each time the program is run.

J.2.5 Assembling and Linking PAL-11S

PAL-11S consists of two independent programs. The first program is a memory clear program. The second is the assembler. All programs are available as ASCII source tapes, object modules and as a load module.

The memory clear program, MEMCLR, (DEC-11-UPLSA-A-PA1) consists of one ASCII tape. This program should never need to be assembled. The object module may be used when constructing a new load module of PAL-11S.

The assembler consists of three program modules which are assembled separately and then linked together. The first is the main program called PAL-11S. It consists of 13 ASCII tapes (DEC-UPLSA-A-PA2-PA14). The second module is the symbol table, PALSYM, which consists of 2 ASCII tapes (DEC-11-UPLSA-A-PA15-PA16). The third is ICXLPT consisting of 2 ASCII tapes (DEC-11-UPLSA-A-PA17-PA18). Also included is PALSYM, specially created for 12K and 16K, consisting of one tape each (DEC-11-UPLSA-A-PA19-PA20).

If changes are made in any of these modules, that module must be assembled by PAL-11S (V003A) and the new object module can be linked with the other object modules. It should be noted that assembly of these programs will result in:

<table>
<thead>
<tr>
<th>Program</th>
<th>Pages of Listing (Decimal)</th>
<th>Number of Symbols (Decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PAL-11S</td>
<td>160</td>
<td>756</td>
</tr>
<tr>
<td>PALSYM</td>
<td>11</td>
<td>32</td>
</tr>
<tr>
<td>IOXLPT</td>
<td>29</td>
<td>191</td>
</tr>
</tbody>
</table>

Also note that there will be two undefined symbols listed at the end of pass 1. These are forward references on direct assignments which get defined properly in pass 2.

An example of the PAL-11S assembly follows:

PAL-11S V003A
*S H
*B H
*L P
*T P/2
END ?
00000000 ERRORS

(first pass on PAL)
(2nd pass on PAL)
(End of tape #1 assembly)
(Remove tape from punch)
ASSEMBLY AND LINKING INSTRUCTIONS

PAL-11S V003A
*S H
*B H
*L P
*T P/2
EOF ?
EOF ?
EOF ?
EOF ?
EOF ?
EOF ?
EOF ?
EOF ?
EOF ?
EOF ?
BINCNT=***** SIMBC=*****
END ?
EOF ?
EOF ?
EOF ?
EOF ?
EOF ?
EOF ?
EOF ?
EOF ?
EOF ?
EOF ?
000000 ERRORS

PAL-11S V003A
*S H
*B H
*L P
*T P/2
EOF ?
END ?
EOF ?
000000 ERRORS

PAL-11S V003A
*S H
*B H
*L P
*T P/2
EOF ?
END ?
EOF ?
000000 ERRORS

PAL-11S V003A
*S H
*B H
*L P
*T P/2
END ?
000000 ERRORS

(Insert PA2 for 1st pass)
(End of PA2, insert PA3)
(End of PA3, insert PA4)
(End of PA4, insert PA5)
(End of PA5, insert PA6)
(End of PA6, insert PA7)
(End of PA7, insert PA8)
(End of PA8, insert PA9)
(End of PA9, insert PA10)
(End of PA10, insert PA11)
(End of PA11, insert PA12)
(End of PA12, insert PA13)
(End of PA13, insert PA14)
(End of PA14 and 1st pass)
(Insert PA2 for 2nd pass)
(End of PA2, insert PA3)
(End of PA3, insert PA4)
(End of PA4, insert PA5)
(End of PA5, insert PA6)
(End of PA6, insert PA7)
(End of PA7, insert PA8)
(End of PA8, insert PA9)
(End of PA9, insert PA10)
(End of PA10, insert PA11)
(End of PA11, insert PA12)
(End of PA12, insert PA13)
(End of PA13, insert PA14)
(End of PA14 and 2nd pass)
(Remove tape from punch)

(1st pass on PA15)
(End of PA15, insert PA16)
(End of PA16, insert PA15 for 2nd pass)
(End of PA15, insert PA16)
(End of 2nd pass)
(Remove tape from punch)

(1st pass on PA17)
(End of PA17, insert PA18)
(End of PA18, insert PA17 for 2nd pass)
(End of PA17, insert PA18)
(End of 2nd pass)
(Remove tape from punch)

(Pass 1 on PA20)
(Pass 2 on PA20)
(End of pass 2)
(Remove tape from punch)
ASSEMBLY AND LINKING INSTRUCTIONS

The final load module is constructed by LINK-11S. First the memory clear program object module is processed by the linker and the resulting load module is left in the punch while the PAL-11S, PALSYM, and IOXLPT object modules are linked to create a second load module. The resulting tape contains two load modules. The first clears memory and then jumps to the absolute loader to load the second.

In order to take advantage of core sizes larger than 8K, PALSYM, the symbol table, specially created for 12K core and 16K core, and the object modules are included with the assembler. To link for 12K (or 16K), simply substitute the appropriate object tape for PALSYM (use DEC-11-UPLSA-A-PR5 for 12K or DEC-11-UPLSA-A-PR6 for 16K) specify a top address to LINK-11S of 57460 for 12K (77460 for 16K) and link as described in the preceding paragraph.

Do not relink PAL-11S to run above 16K. The size of the symbol table is fixed, and there is no need to re-link at a higher address even on large systems.

The supplied tapes are identified as follows:

<table>
<thead>
<tr>
<th>Library Code</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC-11-UPLSA-A-PA1</td>
<td>Tape 1 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA2</td>
<td>Tape 2 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA3</td>
<td>Tape 3 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA4</td>
<td>Tape 4 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA5</td>
<td>Tape 5 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA6</td>
<td>Tape 6 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA7</td>
<td>Tape 7 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA8</td>
<td>Tape 8 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA9</td>
<td>Tape 9 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA10</td>
<td>Tape 10 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA11</td>
<td>Tape 11 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA12</td>
<td>Tape 12 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA13</td>
<td>Tape 13 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA14</td>
<td>Tape 14 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA15</td>
<td>Tape 15 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA16</td>
<td>Tape 16 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA17</td>
<td>Tape 17 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA18</td>
<td>Tape 18 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA19</td>
<td>Tape 19 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PA20</td>
<td>Tape 20 of 20</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PR1</td>
<td>Tape 1 of 6</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PR2</td>
<td>Tape 2 of 6</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PR3</td>
<td>Tape 3 of 6</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PR4</td>
<td>Tape 4 of 6</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PR5</td>
<td>Tape 5 of 6</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PR6</td>
<td>Tape 6 of 6</td>
</tr>
<tr>
<td>DEC-11-UPLSA-A-PL</td>
<td></td>
</tr>
</tbody>
</table>

1This tape is the concatenation of a link of the RELMEM object module followed by a link of the PAL-11S, PALSYM for 8K, and IOXLPT object modules.
### ASSEMBLY AND LINKING INSTRUCTIONS

#### J.2.6 Assembling And Linking LINK-11S

LINK-11S is available as an absolute load module (for an 8K machine), as two object modules (for relinking) and as several ASCII source tapes. There is one object module for the Linker and one for IOXLPT. The supplied object modules may be relinked (using the supplied load module) to load into any size machine larger than 8K. However, the resulting Linker will still assume a top of memory corresponding to an 8K machine (this can be overridden in the command string options). The assumed top of memory and reserved Absolute Loader space may be changed by editing the first linker ASCII tape with ED-11. The parameters to be changed are HGHMEM (high memory address +1 (always even)) and ALODSZ (Absolute Loader size (always even)). The source tapes for the Linker may then be assembled with PAL-11S and the new object module can then replace the supplied Linker object module.

The tapes are identified as follows:

**Library Code**

<table>
<thead>
<tr>
<th>Code</th>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEC-11-ULKSA-A-PA1</td>
<td>Tape 1 of 6</td>
<td>LINK-11S (Main Program)</td>
</tr>
<tr>
<td>DEC-11-ULKSA-A-PA2</td>
<td>Tape 2 of 6</td>
<td>One</td>
</tr>
<tr>
<td>DEC-11-ULKSA-A-PA3</td>
<td>Tape 3 of 6</td>
<td>Assembly</td>
</tr>
<tr>
<td>DEC-11-ULKSA-A-PA4</td>
<td>Tape 4 of 6</td>
<td>Assembly</td>
</tr>
<tr>
<td>DEC-11-ULKSA-A-PA5</td>
<td>Tape 5 of 6</td>
<td>One</td>
</tr>
<tr>
<td>DEC-11-ULKSA-A-PA6</td>
<td>Tape 6 of 6</td>
<td>IOXLPT</td>
</tr>
<tr>
<td>DEC-11-ULKSA-A-FR1</td>
<td>Tape 1 of 2</td>
<td>LINK-11S Object Module</td>
</tr>
<tr>
<td>DEC-11-ULKSA-A-FR2</td>
<td>Tape 2 of 2</td>
<td>IOXLPT Object Module</td>
</tr>
<tr>
<td>DEC-11-ULKSA-A-PL</td>
<td></td>
<td>LINK-11S Load Module</td>
</tr>
</tbody>
</table>

J-11
APPENDIX K

STANDARD PDP-11 ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS</td>
<td>absolute</td>
</tr>
<tr>
<td>A/D</td>
<td>analog-to-digital</td>
</tr>
<tr>
<td>ADC</td>
<td>add carry</td>
</tr>
<tr>
<td>ADRS</td>
<td>address</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>ASL</td>
<td>arithmetic shift left</td>
</tr>
<tr>
<td>ASR</td>
<td>arithmetic shift right</td>
</tr>
<tr>
<td>automatic send/receive</td>
<td></td>
</tr>
<tr>
<td>B</td>
<td>byte</td>
</tr>
<tr>
<td>BAR</td>
<td>bus address register</td>
</tr>
<tr>
<td>BBSY</td>
<td>bus busy</td>
</tr>
<tr>
<td>BCC</td>
<td>branch if carry clear</td>
</tr>
<tr>
<td>BCS</td>
<td>branch if carry set</td>
</tr>
<tr>
<td>BEQ</td>
<td>branch if equal</td>
</tr>
<tr>
<td>BG</td>
<td>bus grant</td>
</tr>
<tr>
<td>BGE</td>
<td>branch if greater or equal</td>
</tr>
<tr>
<td>BGT</td>
<td>branch if greater than</td>
</tr>
<tr>
<td>BHI</td>
<td>branch if higher</td>
</tr>
<tr>
<td>BHIS</td>
<td>branch if higher or same</td>
</tr>
<tr>
<td>BIC</td>
<td>bit clear</td>
</tr>
<tr>
<td>BIS</td>
<td>bit set</td>
</tr>
<tr>
<td>BIT</td>
<td>bit test</td>
</tr>
<tr>
<td>BLE</td>
<td>branch if less or equal</td>
</tr>
<tr>
<td>BLOS</td>
<td>branch if lower or same</td>
</tr>
<tr>
<td>BLT</td>
<td>branch if less than</td>
</tr>
<tr>
<td>BMI</td>
<td>branch if minus</td>
</tr>
<tr>
<td>BNE</td>
<td>branch if not equal</td>
</tr>
<tr>
<td>BPL</td>
<td>branch if plus</td>
</tr>
<tr>
<td>BR</td>
<td>branch</td>
</tr>
<tr>
<td>BRD</td>
<td>bus register data</td>
</tr>
<tr>
<td>BRX</td>
<td>bus request</td>
</tr>
<tr>
<td>BSP</td>
<td>back space</td>
</tr>
<tr>
<td>BSR</td>
<td>bus shift register</td>
</tr>
<tr>
<td>BSY</td>
<td>back space record</td>
</tr>
<tr>
<td>BVC</td>
<td>busy</td>
</tr>
<tr>
<td>BVS</td>
<td>branch if overflow clear</td>
</tr>
<tr>
<td>branch if overflow set</td>
<td></td>
</tr>
<tr>
<td>CBR</td>
<td>console bus request</td>
</tr>
<tr>
<td>CLC</td>
<td>clear carry</td>
</tr>
<tr>
<td>CLK</td>
<td>clock</td>
</tr>
<tr>
<td>CLN</td>
<td>clear negative</td>
</tr>
<tr>
<td>CLR</td>
<td>clear</td>
</tr>
<tr>
<td>CLV</td>
<td>clear overflow</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Definition</td>
</tr>
<tr>
<td>-------------</td>
<td>------------</td>
</tr>
<tr>
<td>CLZ</td>
<td>clear zero</td>
</tr>
<tr>
<td>CMP</td>
<td>compare</td>
</tr>
<tr>
<td>CNPR</td>
<td>console nonprocessor request</td>
</tr>
<tr>
<td>CNTL</td>
<td>control</td>
</tr>
<tr>
<td>COM</td>
<td>complement</td>
</tr>
<tr>
<td>COND</td>
<td>condition</td>
</tr>
<tr>
<td>CONS</td>
<td>console</td>
</tr>
<tr>
<td>CONT</td>
<td>contents</td>
</tr>
<tr>
<td>CP</td>
<td>continue</td>
</tr>
<tr>
<td>CSR</td>
<td>central processor</td>
</tr>
<tr>
<td>D</td>
<td>data</td>
</tr>
<tr>
<td>D/A</td>
<td>digital-to-analog</td>
</tr>
<tr>
<td>DAR</td>
<td>device address register</td>
</tr>
<tr>
<td>DATI</td>
<td>data in</td>
</tr>
<tr>
<td>DATIP</td>
<td>data in, pause</td>
</tr>
<tr>
<td>DATO</td>
<td>data out</td>
</tr>
<tr>
<td>DATOB</td>
<td>data out, byte</td>
</tr>
<tr>
<td>DBR</td>
<td>data buffer register</td>
</tr>
<tr>
<td>DCDR</td>
<td>decoder</td>
</tr>
<tr>
<td>DE</td>
<td>destination effective address</td>
</tr>
<tr>
<td>DEC</td>
<td>decrement</td>
</tr>
<tr>
<td>DEL</td>
<td>Digital Equipment Corp.</td>
</tr>
<tr>
<td>DEP</td>
<td>delay</td>
</tr>
<tr>
<td>DEPF</td>
<td>deposit</td>
</tr>
<tr>
<td>DIV</td>
<td>deposit flag</td>
</tr>
<tr>
<td>DMA</td>
<td>divide</td>
</tr>
<tr>
<td>DMA</td>
<td>direct memory access</td>
</tr>
<tr>
<td>DSEL</td>
<td>device select</td>
</tr>
<tr>
<td>DST</td>
<td>destination</td>
</tr>
<tr>
<td>DSX</td>
<td>display, X-deflection register</td>
</tr>
<tr>
<td>EMT</td>
<td>emulator trap</td>
</tr>
<tr>
<td>ENB</td>
<td>enable</td>
</tr>
<tr>
<td>EOF</td>
<td>end-of-file</td>
</tr>
<tr>
<td>EOM</td>
<td>end-of-medium</td>
</tr>
<tr>
<td>ERR</td>
<td>error</td>
</tr>
<tr>
<td>EX</td>
<td>external</td>
</tr>
<tr>
<td>EXAM</td>
<td>examine</td>
</tr>
<tr>
<td>EXAMF</td>
<td>examine flag</td>
</tr>
<tr>
<td>EXEC</td>
<td>execute</td>
</tr>
<tr>
<td>EXR</td>
<td>external reset</td>
</tr>
<tr>
<td>F</td>
<td>flag (part of signal name)</td>
</tr>
<tr>
<td>FCTN</td>
<td>function</td>
</tr>
<tr>
<td>FILO</td>
<td>first in, last out</td>
</tr>
<tr>
<td>FLG</td>
<td>flag</td>
</tr>
<tr>
<td>GEN</td>
<td>generator</td>
</tr>
<tr>
<td>INDIVR</td>
<td>integer divide routine</td>
</tr>
<tr>
<td>INC</td>
<td>increment</td>
</tr>
<tr>
<td>INCF</td>
<td>increase</td>
</tr>
<tr>
<td>INCF</td>
<td>increment flag</td>
</tr>
<tr>
<td>IND</td>
<td>indicator</td>
</tr>
<tr>
<td>INH</td>
<td>inhibit</td>
</tr>
<tr>
<td>INIT</td>
<td>initialize</td>
</tr>
<tr>
<td>INST</td>
<td>instruction</td>
</tr>
<tr>
<td>INTR</td>
<td>interrupt</td>
</tr>
<tr>
<td>INTRF</td>
<td>interrupt flag</td>
</tr>
<tr>
<td>I/O</td>
<td>input/output</td>
</tr>
<tr>
<td>IOT</td>
<td>input/output trap</td>
</tr>
<tr>
<td>IOX</td>
<td>input/output executive routine</td>
</tr>
<tr>
<td>Abbreviation</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>-------------</td>
</tr>
<tr>
<td>IR</td>
<td>instruction register</td>
</tr>
<tr>
<td>IRD</td>
<td>instruction register decoder</td>
</tr>
<tr>
<td>ISR</td>
<td>instruction shift register</td>
</tr>
<tr>
<td>JMP</td>
<td>jump</td>
</tr>
<tr>
<td>JSR</td>
<td>jump to subroutine</td>
</tr>
<tr>
<td>LIFO</td>
<td>last in, first out</td>
</tr>
<tr>
<td>LKS</td>
<td>line time clock status register</td>
</tr>
<tr>
<td>LOC</td>
<td>location</td>
</tr>
<tr>
<td>LP</td>
<td>line printer</td>
</tr>
<tr>
<td>LSB</td>
<td>least significant bit</td>
</tr>
<tr>
<td>LSBY</td>
<td>least significant byte</td>
</tr>
<tr>
<td>LSD</td>
<td>least significant digit</td>
</tr>
<tr>
<td>MA</td>
<td>memory address</td>
</tr>
<tr>
<td>MAR</td>
<td>memory address register</td>
</tr>
<tr>
<td>MBR</td>
<td>memory buffer register</td>
</tr>
<tr>
<td>MEM</td>
<td>memory</td>
</tr>
<tr>
<td>ML</td>
<td>memory location</td>
</tr>
<tr>
<td>MOV</td>
<td>move</td>
</tr>
<tr>
<td>MSB</td>
<td>most significant bit</td>
</tr>
<tr>
<td>MSBY</td>
<td>most significant byte</td>
</tr>
<tr>
<td>MSD</td>
<td>most significant digit</td>
</tr>
<tr>
<td>MSEL</td>
<td>memory select</td>
</tr>
<tr>
<td>MSYN</td>
<td>master sync</td>
</tr>
<tr>
<td>ND</td>
<td>negative driver</td>
</tr>
<tr>
<td>NEG</td>
<td>negate</td>
</tr>
<tr>
<td>NOR</td>
<td>normalize</td>
</tr>
<tr>
<td>NPG</td>
<td>nonprocessor grant</td>
</tr>
<tr>
<td>NPR</td>
<td>nonprocessor request</td>
</tr>
<tr>
<td>NPRF</td>
<td>nonprocessor request flag</td>
</tr>
<tr>
<td>NS</td>
<td>negative switch</td>
</tr>
<tr>
<td>ODT</td>
<td>octal debugging technique</td>
</tr>
<tr>
<td>OP</td>
<td>operate</td>
</tr>
<tr>
<td>OPR</td>
<td>operation</td>
</tr>
<tr>
<td>PA</td>
<td>operator</td>
</tr>
<tr>
<td>PR</td>
<td>operand</td>
</tr>
<tr>
<td>PA</td>
<td>parity available</td>
</tr>
<tr>
<td>PAL</td>
<td>program assembly language</td>
</tr>
<tr>
<td>PB</td>
<td>parity bit</td>
</tr>
<tr>
<td>PC</td>
<td>program counter</td>
</tr>
<tr>
<td>PD</td>
<td>positive driver</td>
</tr>
<tr>
<td>PDP</td>
<td>programmed data processor</td>
</tr>
<tr>
<td>PERIF</td>
<td>peripheral</td>
</tr>
<tr>
<td>PGM</td>
<td>program</td>
</tr>
<tr>
<td>PP</td>
<td>paper tape punch</td>
</tr>
<tr>
<td>PPB</td>
<td>paper tape punch buffer register</td>
</tr>
<tr>
<td>PPS</td>
<td>paper tape punch status register</td>
</tr>
<tr>
<td>PR</td>
<td>paper tape reader</td>
</tr>
<tr>
<td>PRB</td>
<td>paper tape reader buffer register</td>
</tr>
<tr>
<td>PROC</td>
<td>processor</td>
</tr>
<tr>
<td>PRS</td>
<td>paper tape reader status register</td>
</tr>
<tr>
<td>PS</td>
<td>processor status</td>
</tr>
<tr>
<td>PTR</td>
<td>positive switch</td>
</tr>
<tr>
<td>PTS</td>
<td>priority transfer</td>
</tr>
<tr>
<td>PUN</td>
<td>paper tape software system punch</td>
</tr>
</tbody>
</table>
STANDARD PDP-11 ABBREVIATIONS

RD  read
RDR  reader
REG  register
REL  release
RES  reset
ROL  rotate left
ROM  read-only memory
ROR  rotate right
R/S  rotate shift
RTI  return from interrupt
RTS  return from subroutine
R/W  read/write
R/WSR read/write shift register
S  single
SACK  selection acknowledge
SBC  SUBTRACT CARRY
SC  single cycle
SE  source effective address
SEC  set carry
SEL  select
SEN  set negative
SEV  set overflow
SEX  sign extend
SEZ  set zero
SI  single instruction
SP  stack pointer
spare
SR  switch register
SRC  source
SSYN  slave sync
ST  start
STPM  set trap marker
STR  strobe
SUB  subtract
SVC  service
SWAB  swap byte
TA  trap address
track address
temporary
TEMP
TK  teletype keyboard
TKB  teletype keyboard buffer register
TKS  teletype keyboard status register
TP  teletype printer
TPS  teletype printer status register
TRT  trace trap
timing state control
TST  test

UTR  user trap
VEC  vector

WC  word count
WCR  word count register

XDR  X-line driver
XR CG  X-line read control group
XWC G  X-line write control group

YDR  Y-line driver
YRC G  Y-line read control group
YWCG  Y-line write control group
## APPENDIX L

### CONVERSION TABLES

#### L.1 OCTAL-DECIMAL INTEGER CONVERSIONS

<table>
<thead>
<tr>
<th>Octal (Decim)</th>
<th>0000</th>
<th>0001</th>
<th>0010</th>
<th>0011</th>
<th>0100</th>
<th>0101</th>
<th>0110</th>
<th>0111</th>
<th>1000</th>
<th>1001</th>
<th>1010</th>
<th>1011</th>
<th>1100</th>
<th>1101</th>
<th>1110</th>
<th>1111</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
</tr>
<tr>
<td>0010</td>
<td>0010</td>
<td>0011</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
</tr>
<tr>
<td>0011</td>
<td>0011</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td></td>
</tr>
<tr>
<td>0100</td>
<td>0100</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
</tr>
<tr>
<td>0101</td>
<td>0101</td>
<td>0110</td>
<td>0111</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td></td>
</tr>
<tr>
<td>0110</td>
<td>0110</td>
<td>0111</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0111</td>
<td>0111</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1000</td>
<td>1000</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1001</td>
<td>1001</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1010</td>
<td>1010</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1011</td>
<td>1011</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1100</td>
<td>1100</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1101</td>
<td>1101</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1110</td>
<td>1110</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1111</td>
<td>1111</td>
<td>0000</td>
<td>0001</td>
<td>0010</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---

**L-1**
## OCTAL-DECIMAL CONVERSIONS (Continued)

<table>
<thead>
<tr>
<th>Octal to Decimal</th>
<th>Conversion Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>1024</td>
</tr>
<tr>
<td>2777</td>
<td>1535</td>
</tr>
<tr>
<td>3000</td>
<td>1536</td>
</tr>
<tr>
<td>3777</td>
<td>2047</td>
</tr>
<tr>
<td>200</td>
<td>101</td>
</tr>
<tr>
<td>202</td>
<td>103</td>
</tr>
<tr>
<td>203</td>
<td>104</td>
</tr>
<tr>
<td>204</td>
<td>105</td>
</tr>
<tr>
<td>205</td>
<td>106</td>
</tr>
<tr>
<td>206</td>
<td>107</td>
</tr>
<tr>
<td>207</td>
<td>108</td>
</tr>
<tr>
<td>210</td>
<td>111</td>
</tr>
<tr>
<td>211</td>
<td>112</td>
</tr>
<tr>
<td>212</td>
<td>113</td>
</tr>
<tr>
<td>213</td>
<td>114</td>
</tr>
<tr>
<td>214</td>
<td>115</td>
</tr>
<tr>
<td>215</td>
<td>116</td>
</tr>
<tr>
<td>216</td>
<td>117</td>
</tr>
<tr>
<td>217</td>
<td>118</td>
</tr>
<tr>
<td>218</td>
<td>119</td>
</tr>
<tr>
<td>220</td>
<td>121</td>
</tr>
<tr>
<td>221</td>
<td>122</td>
</tr>
<tr>
<td>222</td>
<td>123</td>
</tr>
<tr>
<td>223</td>
<td>124</td>
</tr>
<tr>
<td>224</td>
<td>125</td>
</tr>
<tr>
<td>225</td>
<td>126</td>
</tr>
<tr>
<td>226</td>
<td>127</td>
</tr>
<tr>
<td>227</td>
<td>128</td>
</tr>
<tr>
<td>230</td>
<td>131</td>
</tr>
<tr>
<td>231</td>
<td>132</td>
</tr>
<tr>
<td>232</td>
<td>133</td>
</tr>
<tr>
<td>233</td>
<td>134</td>
</tr>
<tr>
<td>234</td>
<td>135</td>
</tr>
<tr>
<td>235</td>
<td>136</td>
</tr>
<tr>
<td>236</td>
<td>137</td>
</tr>
<tr>
<td>237</td>
<td>138</td>
</tr>
<tr>
<td>240</td>
<td>140</td>
</tr>
<tr>
<td>241</td>
<td>141</td>
</tr>
<tr>
<td>242</td>
<td>142</td>
</tr>
<tr>
<td>243</td>
<td>143</td>
</tr>
<tr>
<td>244</td>
<td>144</td>
</tr>
<tr>
<td>245</td>
<td>145</td>
</tr>
<tr>
<td>246</td>
<td>146</td>
</tr>
<tr>
<td>247</td>
<td>147</td>
</tr>
<tr>
<td>250</td>
<td>150</td>
</tr>
<tr>
<td>251</td>
<td>151</td>
</tr>
<tr>
<td>252</td>
<td>152</td>
</tr>
<tr>
<td>253</td>
<td>153</td>
</tr>
<tr>
<td>254</td>
<td>154</td>
</tr>
<tr>
<td>255</td>
<td>155</td>
</tr>
<tr>
<td>256</td>
<td>156</td>
</tr>
<tr>
<td>257</td>
<td>157</td>
</tr>
<tr>
<td>258</td>
<td>158</td>
</tr>
<tr>
<td>259</td>
<td>159</td>
</tr>
<tr>
<td>260</td>
<td>160</td>
</tr>
<tr>
<td>261</td>
<td>161</td>
</tr>
<tr>
<td>262</td>
<td>162</td>
</tr>
<tr>
<td>263</td>
<td>163</td>
</tr>
<tr>
<td>264</td>
<td>164</td>
</tr>
<tr>
<td>265</td>
<td>165</td>
</tr>
<tr>
<td>266</td>
<td>166</td>
</tr>
<tr>
<td>267</td>
<td>167</td>
</tr>
<tr>
<td>270</td>
<td>170</td>
</tr>
<tr>
<td>271</td>
<td>171</td>
</tr>
<tr>
<td>272</td>
<td>172</td>
</tr>
<tr>
<td>273</td>
<td>173</td>
</tr>
<tr>
<td>274</td>
<td>174</td>
</tr>
<tr>
<td>275</td>
<td>175</td>
</tr>
<tr>
<td>276</td>
<td>176</td>
</tr>
<tr>
<td>277</td>
<td>177</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Decimal to Octal</th>
<th>Conversion Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>200</td>
<td>101</td>
</tr>
<tr>
<td>202</td>
<td>103</td>
</tr>
<tr>
<td>203</td>
<td>104</td>
</tr>
<tr>
<td>204</td>
<td>105</td>
</tr>
<tr>
<td>205</td>
<td>106</td>
</tr>
<tr>
<td>206</td>
<td>107</td>
</tr>
<tr>
<td>207</td>
<td>108</td>
</tr>
<tr>
<td>210</td>
<td>111</td>
</tr>
<tr>
<td>211</td>
<td>112</td>
</tr>
<tr>
<td>212</td>
<td>113</td>
</tr>
<tr>
<td>213</td>
<td>114</td>
</tr>
<tr>
<td>214</td>
<td>115</td>
</tr>
<tr>
<td>215</td>
<td>116</td>
</tr>
<tr>
<td>216</td>
<td>117</td>
</tr>
<tr>
<td>217</td>
<td>118</td>
</tr>
<tr>
<td>218</td>
<td>119</td>
</tr>
<tr>
<td>220</td>
<td>121</td>
</tr>
<tr>
<td>221</td>
<td>122</td>
</tr>
<tr>
<td>222</td>
<td>123</td>
</tr>
<tr>
<td>223</td>
<td>124</td>
</tr>
<tr>
<td>224</td>
<td>125</td>
</tr>
<tr>
<td>225</td>
<td>126</td>
</tr>
<tr>
<td>226</td>
<td>127</td>
</tr>
<tr>
<td>227</td>
<td>128</td>
</tr>
<tr>
<td>230</td>
<td>131</td>
</tr>
<tr>
<td>231</td>
<td>132</td>
</tr>
<tr>
<td>232</td>
<td>133</td>
</tr>
<tr>
<td>233</td>
<td>134</td>
</tr>
<tr>
<td>234</td>
<td>135</td>
</tr>
<tr>
<td>235</td>
<td>136</td>
</tr>
<tr>
<td>236</td>
<td>137</td>
</tr>
<tr>
<td>237</td>
<td>138</td>
</tr>
<tr>
<td>240</td>
<td>140</td>
</tr>
<tr>
<td>241</td>
<td>141</td>
</tr>
<tr>
<td>242</td>
<td>142</td>
</tr>
<tr>
<td>243</td>
<td>143</td>
</tr>
<tr>
<td>244</td>
<td>144</td>
</tr>
<tr>
<td>245</td>
<td>145</td>
</tr>
<tr>
<td>246</td>
<td>146</td>
</tr>
<tr>
<td>247</td>
<td>147</td>
</tr>
<tr>
<td>250</td>
<td>150</td>
</tr>
<tr>
<td>251</td>
<td>151</td>
</tr>
<tr>
<td>252</td>
<td>152</td>
</tr>
<tr>
<td>253</td>
<td>153</td>
</tr>
<tr>
<td>254</td>
<td>154</td>
</tr>
<tr>
<td>255</td>
<td>155</td>
</tr>
<tr>
<td>256</td>
<td>156</td>
</tr>
<tr>
<td>257</td>
<td>157</td>
</tr>
<tr>
<td>258</td>
<td>158</td>
</tr>
<tr>
<td>259</td>
<td>159</td>
</tr>
<tr>
<td>260</td>
<td>160</td>
</tr>
<tr>
<td>261</td>
<td>161</td>
</tr>
<tr>
<td>262</td>
<td>162</td>
</tr>
<tr>
<td>263</td>
<td>163</td>
</tr>
<tr>
<td>264</td>
<td>164</td>
</tr>
<tr>
<td>265</td>
<td>165</td>
</tr>
<tr>
<td>266</td>
<td>166</td>
</tr>
<tr>
<td>267</td>
<td>167</td>
</tr>
<tr>
<td>270</td>
<td>170</td>
</tr>
<tr>
<td>271</td>
<td>171</td>
</tr>
<tr>
<td>272</td>
<td>172</td>
</tr>
<tr>
<td>273</td>
<td>173</td>
</tr>
<tr>
<td>274</td>
<td>174</td>
</tr>
<tr>
<td>275</td>
<td>175</td>
</tr>
<tr>
<td>276</td>
<td>176</td>
</tr>
<tr>
<td>277</td>
<td>177</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Conversion Table</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
</tr>
<tr>
<td>2047</td>
</tr>
</tbody>
</table>

L-2
### OCTAL-DEcimal INTEGER CONVERSIONS (Continued)

<table>
<thead>
<tr>
<th>Octal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
</tr>
</tbody>
</table>

#### Conversions Table

<table>
<thead>
<tr>
<th>Octal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0002</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

#### Octal to Decimal Conversion

<table>
<thead>
<tr>
<th>Octal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>0</td>
</tr>
<tr>
<td>0001</td>
<td>1</td>
</tr>
<tr>
<td>0002</td>
<td>2</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

#### Decimal to Octal Conversion

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Octal</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0000</td>
</tr>
<tr>
<td>1</td>
<td>0001</td>
</tr>
<tr>
<td>2</td>
<td>0002</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
### OCTAL-DECIMAL INTEGER CONVERSIONS (Concluded)

<table>
<thead>
<tr>
<th>Octal</th>
<th>Decimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>4096</td>
</tr>
<tr>
<td>20000</td>
<td>8192</td>
</tr>
<tr>
<td>30000</td>
<td>12288</td>
</tr>
<tr>
<td>40000</td>
<td>16384</td>
</tr>
<tr>
<td>50000</td>
<td>20480</td>
</tr>
<tr>
<td>60000</td>
<td>24576</td>
</tr>
<tr>
<td>70000</td>
<td>28672</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Octal</th>
<th>3072</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000</td>
<td>6000</td>
</tr>
<tr>
<td>0010</td>
<td>3008</td>
</tr>
<tr>
<td>0020</td>
<td>6008</td>
</tr>
<tr>
<td>0030</td>
<td>9008</td>
</tr>
<tr>
<td>0040</td>
<td>12008</td>
</tr>
<tr>
<td>0050</td>
<td>15008</td>
</tr>
<tr>
<td>0060</td>
<td>18008</td>
</tr>
<tr>
<td>0070</td>
<td>21008</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Octal</th>
<th>3583</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100</td>
<td>10000</td>
</tr>
<tr>
<td>0110</td>
<td>13000</td>
</tr>
<tr>
<td>0120</td>
<td>16000</td>
</tr>
<tr>
<td>0130</td>
<td>19000</td>
</tr>
<tr>
<td>0140</td>
<td>22000</td>
</tr>
<tr>
<td>0150</td>
<td>25000</td>
</tr>
<tr>
<td>0160</td>
<td>28000</td>
</tr>
<tr>
<td>0170</td>
<td>31000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Octal</th>
<th>3584</th>
</tr>
</thead>
<tbody>
<tr>
<td>0700</td>
<td>3584</td>
</tr>
<tr>
<td>0800</td>
<td>3585</td>
</tr>
<tr>
<td>0900</td>
<td>3586</td>
</tr>
<tr>
<td>1000</td>
<td>3587</td>
</tr>
<tr>
<td>1100</td>
<td>3588</td>
</tr>
<tr>
<td>1200</td>
<td>3589</td>
</tr>
<tr>
<td>1300</td>
<td>3590</td>
</tr>
<tr>
<td>1400</td>
<td>3591</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Octal</th>
<th>4095</th>
</tr>
</thead>
<tbody>
<tr>
<td>7000</td>
<td>3584</td>
</tr>
<tr>
<td>7100</td>
<td>3585</td>
</tr>
<tr>
<td>7200</td>
<td>3586</td>
</tr>
<tr>
<td>7300</td>
<td>3587</td>
</tr>
<tr>
<td>7400</td>
<td>3588</td>
</tr>
<tr>
<td>7500</td>
<td>3589</td>
</tr>
<tr>
<td>7600</td>
<td>3590</td>
</tr>
<tr>
<td>7700</td>
<td>3591</td>
</tr>
</tbody>
</table>

L-4
## L.2 POWERS OF TWO

<table>
<thead>
<tr>
<th>n</th>
<th>2^n</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
</tr>
<tr>
<td>6</td>
<td>64</td>
</tr>
<tr>
<td>7</td>
<td>128</td>
</tr>
<tr>
<td>8</td>
<td>256</td>
</tr>
<tr>
<td>9</td>
<td>512</td>
</tr>
<tr>
<td>10</td>
<td>1024</td>
</tr>
<tr>
<td>11</td>
<td>2048</td>
</tr>
<tr>
<td>12</td>
<td>4096</td>
</tr>
<tr>
<td>13</td>
<td>8192</td>
</tr>
<tr>
<td>14</td>
<td>16384</td>
</tr>
<tr>
<td>15</td>
<td>32768</td>
</tr>
<tr>
<td>16</td>
<td>65536</td>
</tr>
<tr>
<td>17</td>
<td>131072</td>
</tr>
<tr>
<td>18</td>
<td>262144</td>
</tr>
<tr>
<td>19</td>
<td>524288</td>
</tr>
<tr>
<td>20</td>
<td>1048576</td>
</tr>
<tr>
<td>21</td>
<td>2097152</td>
</tr>
<tr>
<td>22</td>
<td>4194304</td>
</tr>
<tr>
<td>23</td>
<td>8388608</td>
</tr>
<tr>
<td>24</td>
<td>16777216</td>
</tr>
<tr>
<td>25</td>
<td>33554432</td>
</tr>
<tr>
<td>26</td>
<td>67108864</td>
</tr>
<tr>
<td>27</td>
<td>134217728</td>
</tr>
<tr>
<td>28</td>
<td>268435456</td>
</tr>
<tr>
<td>29</td>
<td>536870912</td>
</tr>
<tr>
<td>30</td>
<td>1073741824</td>
</tr>
<tr>
<td>31</td>
<td>2147483648</td>
</tr>
<tr>
<td>32</td>
<td>4294967296</td>
</tr>
<tr>
<td>33</td>
<td>8589934592</td>
</tr>
<tr>
<td>34</td>
<td>17179869184</td>
</tr>
<tr>
<td>35</td>
<td>34359738368</td>
</tr>
<tr>
<td>36</td>
<td>68719476736</td>
</tr>
<tr>
<td>37</td>
<td>137438953472</td>
</tr>
<tr>
<td>38</td>
<td>274877896894</td>
</tr>
<tr>
<td>39</td>
<td>549755813888</td>
</tr>
<tr>
<td>40</td>
<td>109951162776</td>
</tr>
<tr>
<td>41</td>
<td>219902335552</td>
</tr>
<tr>
<td>42</td>
<td>439804671104</td>
</tr>
<tr>
<td>43</td>
<td>879609342208</td>
</tr>
<tr>
<td>44</td>
<td>1759218684416</td>
</tr>
<tr>
<td>45</td>
<td>3518437208832</td>
</tr>
<tr>
<td>46</td>
<td>7036874417764</td>
</tr>
<tr>
<td>47</td>
<td>14073748835528</td>
</tr>
<tr>
<td>48</td>
<td>28147497671056</td>
</tr>
<tr>
<td>49</td>
<td>56294950988800</td>
</tr>
<tr>
<td>50</td>
<td>1125899019776000</td>
</tr>
<tr>
<td>51</td>
<td>2251798039552000</td>
</tr>
<tr>
<td>52</td>
<td>4503596079104000</td>
</tr>
<tr>
<td>53</td>
<td>9007192038400000</td>
</tr>
<tr>
<td>54</td>
<td>18014384076800000</td>
</tr>
<tr>
<td>55</td>
<td>36028768153600000</td>
</tr>
<tr>
<td>56</td>
<td>72057536307200000</td>
</tr>
<tr>
<td>57</td>
<td>144115072614400000</td>
</tr>
<tr>
<td>58</td>
<td>288230145228800000</td>
</tr>
<tr>
<td>59</td>
<td>576460290457600000</td>
</tr>
</tbody>
</table>

---

L-5
CONVERSION TABLES

L.3 SCALES OF NOTATION

L.3.1 \(2^n\) In Decimal

<table>
<thead>
<tr>
<th>x</th>
<th>(2^x)</th>
<th>x</th>
<th>(2^x)</th>
<th>x</th>
<th>(2^x)</th>
<th>x</th>
<th>(2^x)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.001</td>
<td>1.0000</td>
<td>0.01</td>
<td>1.0099</td>
<td>0.02</td>
<td>1.0195</td>
<td>0.03</td>
<td>1.0291</td>
</tr>
<tr>
<td>0.002</td>
<td>1.0018</td>
<td>0.02</td>
<td>1.0195</td>
<td>0.03</td>
<td>1.0291</td>
<td>0.04</td>
<td>1.0388</td>
</tr>
<tr>
<td>0.003</td>
<td>1.0036</td>
<td>0.03</td>
<td>1.0388</td>
<td>0.04</td>
<td>1.0485</td>
<td>0.05</td>
<td>1.0582</td>
</tr>
<tr>
<td>0.004</td>
<td>1.0056</td>
<td>0.04</td>
<td>1.0485</td>
<td>0.05</td>
<td>1.0582</td>
<td>0.06</td>
<td>1.0678</td>
</tr>
<tr>
<td>0.005</td>
<td>1.0078</td>
<td>0.05</td>
<td>1.0575</td>
<td>0.06</td>
<td>1.0678</td>
<td>0.07</td>
<td>1.0775</td>
</tr>
<tr>
<td>0.006</td>
<td>1.0099</td>
<td>0.06</td>
<td>1.0678</td>
<td>0.07</td>
<td>1.0775</td>
<td>0.08</td>
<td>1.0874</td>
</tr>
<tr>
<td>0.007</td>
<td>1.0122</td>
<td>0.07</td>
<td>1.0874</td>
<td>0.08</td>
<td>1.0971</td>
<td>0.09</td>
<td>1.1070</td>
</tr>
<tr>
<td>0.008</td>
<td>1.0145</td>
<td>0.08</td>
<td>1.1070</td>
<td>0.09</td>
<td>1.1169</td>
<td>0.10</td>
<td>1.1269</td>
</tr>
</tbody>
</table>

L.3.2 \(10^n\) In Octal

<table>
<thead>
<tr>
<th>(10^n)</th>
<th>n</th>
<th>(10^n)</th>
<th>n</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 0</td>
<td>0</td>
<td>1 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0</td>
<td>0</td>
<td>1 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0 0</td>
<td>0</td>
<td>1 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0 0 0</td>
<td>0</td>
<td>1 0 0 0 0</td>
<td>0</td>
</tr>
<tr>
<td>1 0 0 0 0 0</td>
<td>0</td>
<td>1 0 0 0 0 0</td>
<td>0</td>
</tr>
</tbody>
</table>

L.3.3 \(n\) Log 2 and 10 In Decimal

<table>
<thead>
<tr>
<th>n</th>
<th>(n \log_{10} 2)</th>
<th>(n \log_{10} 10)</th>
<th>n</th>
<th>(n \log_{10} 2)</th>
<th>(n \log_{10} 10)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.30102</td>
<td>0.30102</td>
<td>6</td>
<td>1.76096</td>
<td>1.76096</td>
</tr>
<tr>
<td>2</td>
<td>0.60205</td>
<td>0.60205</td>
<td>7</td>
<td>2.04146</td>
<td>2.04146</td>
</tr>
<tr>
<td>3</td>
<td>0.90309</td>
<td>0.90309</td>
<td>8</td>
<td>2.32432</td>
<td>2.32432</td>
</tr>
<tr>
<td>4</td>
<td>1.20413</td>
<td>1.20413</td>
<td>9</td>
<td>2.60944</td>
<td>2.60944</td>
</tr>
<tr>
<td>5</td>
<td>1.50515</td>
<td>1.50515</td>
<td>10</td>
<td>2.89477</td>
<td>2.89477</td>
</tr>
</tbody>
</table>

L.3.4 Addition and Multiplication, Binary and Octal

### Addition

**Binary Scale**

<table>
<thead>
<tr>
<th>0 + 0 = 0</th>
<th>0 + 1 = 1</th>
<th>1 + 0 = 1</th>
<th>1 + 1 = 10</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Octal Scale</th>
<th>00 01 02 03 04 05 06 07</th>
<th>10 11 12 13 14 15 16</th>
</tr>
</thead>
</table>

### Multiplication

**Binary Scale**

<table>
<thead>
<tr>
<th>0 × 0 = 0</th>
<th>0 × 1 = 0</th>
<th>1 × 0 = 0</th>
<th>1 × 1 = 1</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Octal Scale</th>
<th>00 01 02 03 04 05 06 07</th>
<th>10 11 12 13 14 15 16</th>
</tr>
</thead>
</table>
### L.3.5 Mathematical Constants In Octal

<table>
<thead>
<tr>
<th></th>
<th>Octal Representation</th>
<th></th>
<th>Octal Representation</th>
<th></th>
<th>Octal Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \pi )</td>
<td>3.11037 552421,</td>
<td>( e )</td>
<td>2.55760 521305,</td>
<td>( \gamma )</td>
<td>0.44742 147707,</td>
</tr>
<tr>
<td>( \pi^{-1} )</td>
<td>0.24276 301556,</td>
<td>( e^{-1} )</td>
<td>0.27426 530661,</td>
<td>( \ln \gamma )</td>
<td>-0.43127 233602,</td>
</tr>
<tr>
<td>( \sqrt{\pi} )</td>
<td>1.61337 611067,</td>
<td>( \sqrt{e} )</td>
<td>1.51411 230704,</td>
<td>( \log_2 \gamma )</td>
<td>-0.62573 030645,</td>
</tr>
<tr>
<td>( \ln \pi )</td>
<td>1.11206 404435,</td>
<td>( \log_{10} e )</td>
<td>0.33626 754251,</td>
<td>( \sqrt{2} )</td>
<td>1.32404 746320,</td>
</tr>
<tr>
<td>( \log_{10} \pi )</td>
<td>1.51544 163223,</td>
<td>( \log_{10} e )</td>
<td>1.34252 166245,</td>
<td>( \ln 2 )</td>
<td>0.54271 027760,</td>
</tr>
<tr>
<td>( \sqrt{10} )</td>
<td>3.12305 407267,</td>
<td>( \log_{10} 10 )</td>
<td>3.24464 741136,</td>
<td>( \ln 10 )</td>
<td>2.23273 067355,</td>
</tr>
</tbody>
</table>
APPENDIX M

NOTE TO USERS OF SERIAL LA30 AND 600, 1200, AND 2400 BAUD VT05'S

The serial LA30 requires that filler characters follow each carriage return; the 600, 1200, and 2400 baud VT05's require that filler characters follow each line feed. The following table lists the filler characters needed. The byte at location 44 has been established as the filler count and the byte at location 45 contains the character to be filled. These locations are initially set to zero by PAL-11A and ED-11 to allow normal operation of the program.

Depending on the terminal, change the locations as follows:

<table>
<thead>
<tr>
<th>LOC 44</th>
<th>LOC 45</th>
<th>Resulting Word (binary)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA30</td>
<td>011</td>
<td>015</td>
</tr>
<tr>
<td></td>
<td></td>
<td>00001101000001001</td>
</tr>
<tr>
<td>VT05 600 Baud</td>
<td>001</td>
<td>012</td>
</tr>
<tr>
<td>VT05 1200 Baud</td>
<td>002</td>
<td>012</td>
</tr>
<tr>
<td>VT05 2400 Baud</td>
<td>004</td>
<td>012</td>
</tr>
</tbody>
</table>

The proper binary word can be stored at location 44 by using the console switches as described in section 2.1.2 of this manual. Furthermore, users with a 2400 baud VT05 should avoid the use of vertical tab characters in their programs. Vertical tabs will not be properly filled and may cause characters to be lost.

Once the changes have been made, the program may be dumped to paper tape by using the bootstrap version of DUMPAB (see section 6.3 in this manual). However, since programs change each time a new version is released, it is necessary to have a program listing to determine the exact memory limits to be dumped.

The above changes only affect output to the console teleprinter.

Users of IOX or IOXLPT source tapes will find the byte at location 44 tagged "I.44:" and the byte at location 45 tagged "I.45:". These locations are defined near the end of the second source tape and can be changed to appropriate values using ED-11.

ODT-11 uses the locations (44 and 45) but does not set them to zero initially.
APPENDIX N

USING THE ABSOLUTE LOADER ON PDP-11'S WITHOUT SWITCH REGISTERS

This appendix describes the procedures for loading and using the Absolute Loader on PDP-11's without switch registers. The procedures are divided into LSI-11, M9301-YB bootstrap loader, and M9301-YA bootstrap loader. Chapter 6 describes the procedures for machines with switch registers.

N.1 LSI-11

The following are instructions for loading and using the Absolute Loader on an LSI-11.

1. Press the BOOT/INIT switch on the LSI-11 front panel to enable the bootstrap loader. An @ prints at the terminal.

2. Place the Absolute Loader tape (DEC-11-UABL-B-A-PO) in the reader.

3. Type the status register address of the input device and L to load the Absolute Loader.

   For example, when loading from the console terminal paper tape reader, type:

   @177560L

   When the tape has been read, an @ followed by the start address of the Absolute Loader prints at the terminal.

   For example, on a machine with 8K memory, type:

   @177560L

   The Absolute Loader prints the address of the Absolute Loader:

   @37500

   4. Place the tape to be loaded via the Absolute Loader in the reader.

   5. Select the type of loading from the following:

      a. Normal Loading

         For normal loading, type the address of the Absolute Loader (printed at the terminal), followed by G, e.g.,

         @xxx500G
where xxx is the memory size of the system and is:

<table>
<thead>
<tr>
<th>xxx</th>
<th>Memory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>017</td>
<td>4K</td>
</tr>
<tr>
<td>037</td>
<td>8K</td>
</tr>
<tr>
<td>057</td>
<td>12K</td>
</tr>
<tr>
<td>077</td>
<td>16K</td>
</tr>
<tr>
<td>117</td>
<td>20K</td>
</tr>
<tr>
<td>137</td>
<td>24K</td>
</tr>
<tr>
<td>157</td>
<td>28K</td>
</tr>
</tbody>
</table>

For example, in an 8K system, type:

@37500G

Normal loading can also be achieved by typing the P command, e.g.,

@P

b. Relocated Loading

Type the software switch register value and deposit the relocation value as follows:

@xxx516/yyyyyy zzzzzz

@xxx500G

or type:

@xxx516/yyyyyy zzzzzz

@P

where xxx516 is dependent on memory size and is the address of the software switch register, yyyyy is the old content of the switch register, and zzzzz is the new relocation value.

The value of zzzzzz is explained in Section 6.2.2 for the value of the switch register for relocated loading. For example, in an 8K system, the dialogue would be:

@37516/yyyyyy zzzzzz

The following is an example of a normal load on an 8K machine.

;boot system and put Absolute Loader in reader
@177560L
@37500
@P

@37500
;Absolute Loader tape is read

@37516/000000 1001 ;put tape to be loaded in reader
@P ;tape is read

The following is an example of a relocated load on an 8K machine:

@177560L ;boot system
@37500 ;put Absolute Loader tape in reader
@37516/000000 1001 ;put tape in reader
@P ;tape is read
USING THE ABSOLUTE LOADER ON PDP-11'S WITHOUT SWITCH REG

To continue loading, change 1001 in the above example to 1.

6. If more tapes are to be loaded as explained in Section 6.2.2, put the next tape in the reader and repeat section a or b of item 5.

7. If the tape is not self-starting, the halt address of the Absolute Loader is printed, followed by an @. Type the starting address followed by a G to start the program.

   @37500
   @xxxxxG

where xxxxxxx is the starting address of the program.

N.2 M9301-YB BOOTSTRAP LOADER

The following are instructions for loading and using the Absolute Loader on a PDP-11 (e.g., PDP-11/04) without a switch register.

1. Press the BOOT/INIT switch on the PDP-11 front panel to enable the bootstrap loader. A $ and four numbers print at the terminal. The four numbers are the values of R0, R4, R6, and the PC, respectively.

   For example:

   0077400 012450 000546 004054
   $

2. Place the Absolute Loader (DEC-11-UABLBE-A-PO) in the reader.

3. Type the device code (PR for the PC11 high-speed reader or TT for the terminal reader) to load the Absolute Loader.

   $PR

   or

   $TT

   when the tape has read in, the machine halts.

4. Place the tape to be loaded by the Absolute Loader in the reader.

5. Select the type of loading from the following:
   a. Normal Loading
      For normal loading, press the CONT switch on the PDP-11 front panel.
   b. Relocated Loading
      1) Press the BOOT/INIT switch; a $ followed by the four numbers explained in item 1 prints at the terminal.
      2) Load the address of the software switch register as follows:

         $L xxx516

N-3
3) Deposit the relocation value in the software switch register as follows:

\[ \$D \text{ yyyy} \]

where yyyyy is the value explained in Section 6.2.2 for relocated loading.

4) Load the starting address of the Absolute Loader as follows:

\[ \$L \text{ xxx500} \]

5) Type S to start running the Absolute Loader.

\[ \$S \]

6. If more tapes are to be loaded as explained in Section 6.2.2, put the next tape in the reader and repeat section a or b of item 5.

7. If the tape is not self-starting,
   a. Press the BOOT/INIT switch.
   b. Load the starting address of the program with the L command, i.e.,

\[ \$L \text{xxxxxx} \]

   c. Start the program with the S command:

\[ \$S \]

The following are examples for PDP-11 with 16K words of memory.

Relocated - continuous loading:

\[ \$L \text{77516} \]

\[ \$D \text{ 1} \]

\[ \$L \text{77500} \]

\[ \$S \]

Relocated - load in specified area of core:

\[ \$L \text{77516} \]

\[ \$D \text{1001} \]

\[ \$L \text{77500} \]

\[ \$S \]

N.3 M9301-YA BOOTSTRAP LOADER

The instructions for loading and using the Absolute Loader on a PDP-11 (e.g., PDP-11/04) without a switch register but with a console terminal are the same as described in Section 0.2.
USING THE ABSOLUTE LOADER ON PDP-11'S WITHOUT SWITCH REG

PDP-11's without console terminals may only be loaded with normal loading methods. See the M9301 Maintenance Manual for instructions on placing the address of the paper tape bootstrap in the micro switch on the M9301 module. The following instructions are for PDP-11's without console terminals.

1. Place the Absolute Loader tape (DEC-ll-UABLB-A-PO) in the reader.

2. Press the BOOT/INIT switch. When the tape has read in, the machine halts.

3. Place the self-starting tape to be loaded by the Absolute Loader in the reader.

4. Press the CONT switch.
INDEX

Abbreviations, standard
PDP-11, K-1
Absolute and relocatable
program sections,
LINK-11S, 3-2
Absolute expressions, PAL-11S,
1-9
Absolute Loader, 6-1, 6-10, F-3
PAL-11S, 1-24
Accessing internal registers,
ODT-11, 5-6
Adding devices to IOX, 7-21
Address Mode syntax,
PAL-11A, C-2
PAL-11S, B-2
Addressing,
PAL-11A, 2-9
PAL-11S, 1-12
Altering register contents,
9-6
Arithmetic and logical
operators,
PAL-11A, 2-8
PAL-11S, 1-8
ASCII,
character set, A-1
conversion, PAL-11A, 2-8
conversion, PAL-11S, 1-8
.ASCII directive,
PAL-11A, 2-17
PAL-11S, 1-21
.ASECT and .CSECT program
section directives,
PAL-11S, 1-19
Assembler directives,
PAL-11A, 2-3, 2-15
PAL-11S, 1-18, B-8
Assembly and linking instruc-
tions, J-1
Assembly dialogue, PAL-11A,
2-23
Assembly listing,
PAL-11A, 2-24
PAL-11S, 1-30
Assembly Location Counter,
PAL-11A, 2-8
PAL-11S, 1-10
Assignment, direct,
PAL-11A, 2-5
PAL-11S, 1-6
Autodecrement Mode,
PAL-11A, 2-11
PAL-11S, 1-14
Autoincrement Mode,
PAL-11A, 2-10
PAL-11A deferred, 2-11
PAL-11S, 1-13
Blank operator field, PAL-11A,
2-16
Bootstrap Loader, 6-1, F-1
loading and verifying the, 6-7
Bootstrap tapes, loading into
core, 6-8
Bootstrap, paper tape, 6-2
Breakpoints,
ODT-11, 5-6
ODT-11X, 5-12, 5-14
Buffer arrangement in data
transfer command, 7-4
Buffer size, IOX, 7-5
Buffering, double, 7-15
Byte count, IOX, 7-8
 BYTE directive,
PAL-11A, 2-17
PAL-11S, 1-21
Byte offset, PAL-11S, 1-17
Calculating offsets,
ODT-11, 5-9
ODT-11X, 5-11
Changing, opening, and closing
locations,
ODT-11, 5-4
ODT-11X, 5-10
Character location pointer (Dot),
ED-11, 4-2
Character set,
ASCII, A-1
PAL-11A, 2-2
PAL-11S, 1-2
Closing, opening, and changing
locations,
ODT-11, 5-4
ODT-11X, 5-10
Codes, PAL-11A error, 2-25
Coding techniques, 9-6
Commands and functions, ODT-11,
5-3
Command delimiters, ED-11, 4-2
Command Mode and Text Mode, ED-11,
4-1
Command properties, line-oriented,
ED-11, 4-3
Commands, ED-11, 4-4
to modify the text, 4-8
to move Dot and Mark, 4-6
Comments,
PAL-11A, 2-4
PAL-11S, 1-4
Communication and data flow,
ODT-11X, 5-15
Communication with IOX, 7-1

Index-1
INDEX (Cont.)

Conditional assembly directives,
  PAL-11S, 1-23
Control format, PAL-11S, 1-4
Conversion, PAL-11S ASCII, 1-8
Conversion tables, L-1
Core memory dumps, 6-14, F-4
Counter, PAL-11S program, 1-12
Creating a paper tape, ED-11,
  4-11
CTRL/U, PAL-11S, 1-24

Data transfer commands,
  buffer arrangement in, 7-4
  device conflicts in, 7-12
Data transfers, IOX, 7-11
Decimal numbers, PAL-11S, 1-8
Deferred Autodecrement Mode,
  PAL-11A, 2-11
  PAL-11S, 1-14
Deferred Autoincrement Mode,
  PAL-11A, 2-11
  PAL-11S, 1-13
Deferred Immediate (Absolute)
  and Immediate Mode, PAL-11A,
    2-12
Deferred Index Mode,
  PAL-11A, 2-12
  PAL-11S, 1-14
Deferred Register Mode, PAL-11S,
    1-13
Deferred Relative and Relative
  Mode, PAL-11A, 2-13
Device Assignment Table, IOX,
    7-3
Device conflicts in data
  transfer commands, 7-12
Device Interrupt Table (DIT),
    7-20
Device Status Table, (DST),
    7-21
Dialogue,
  PAL-11A assembly, 2-23
  PAL-11A initial, 2-18
  PAL-11S initial, 1-24
Direct assignment,
  PAL-11A, 2-5
  PAL-11S, 1-6
Directives,
  PAL-11A .ASCII, 2-17
  PAL-11A .BYTE, 2-17
  PAL-11A .END, 2-16
  PAL-11A .EOT, 2-15
  PAL-11A .EVEN, 2-16
  PAL-11A .WORD, 2-16
  PAL-11S .ASCII, 1-21
  PAL-11S .ASECT, 1-19
  PAL-11S .BYTE, 1-21

Directives (cont.),
  PAL-11S .CSECT, 1-19
  PAL-11S .END, 1-20
  PAL-11S .EOT, 1-20
  PAL-11S .EVEN, 1-20
  PAL-11S .GLOBL, 1-18
  PAL-11S .LIMIT, 1-23
  PAL-11S .RAD50, 1-22
  PAL-11S .TITLE, 1-18
  PAL-11S .WORD, 1-20
Directory, PAL-11S global symbol,
    1-30
Done bit, IOX, 7-7
(Dot) character location pointer,
  ED-11, 4-2
Double buffering, 7-15
Dumps,
  core memory, 6-14, F-4
  output formats, 6-17
  storage maps, 6-17
Duplication, tape, I-1

ED-11,
Character location pointer,
  (Dot), 4-2
command delimiters, 4-2
Command Mode and Text Mode, 4-1
commands, 4-4
commands to modify the text, 4-8
commands to move Dot and Mark,
  4-6
creating a paper tape, 4-11
editing example, 4-12
error corrections, 4-10
grouping of commands, D-3
input and output commands, 4-4,
  D-1
line-oriented command properties,
  4-3
Mark, 4-3
operating procedures, 4-10, D-4
page buffer, 4-3
pointer-positioning commands,
  D-2
restarting, 4-11
search commands, 4-7, D-2
software error halts, 4-17
starting, 4-11
symbols, D-3
Editing example, ED-11, 4-12
Editor (ED-11), PAL-11S, 1-2
.END directive,
  PAL-11A, 2-16
  PAL-11S, 1-20
End-of-File Bit, IOX, 7-7
End-of-Medium Bit, IOX, 7-7

Index-2
INDEX (Cont.)

.EOT directive, PAL-11A, 2-15
PAL-11S, 1-20
Error codes, IOX, 7-6
PAL-11A, 2-7, 2-25, C-8
PAL-11S, 1-31, B-10
Error corrections, ED-11, 4-10
Error detection, ODT-11X, 5-13
Error halts, PAL-11A software, 2-26
Error procedure and messages, LINK-11S, 3-7
Error, Q, PAL-11S, 1-2
.EVEN directive, PAL-11A, 2-16
PAL-11S, 1-20
Example of program using IOX, 7-17
Expressions, absolute, PAL-11S, 1-9
external, PAL-11S, 1-9
mode of, PAL-11S, 1-9
PAL-11A, 2-7
PAL-11S, 1-7
relocatable, PAL-11S, 1-9
External expression, PAL-11S, 1-9
External symbol, PAL-11S, 1-5

Fatal errors, IOX, 7-17, G-2
Fields, PAL-11A instruction operand, 2-14
Format control, PAL-11A, 2-4
PAL-11S, 1-4
Format, PAL-11S statement, 1-2
Forms, PAL-11A instruction, 2-14
FPMP-11,
non-OTS routines, H-7
OTS routines, H-2
routines accessed via trap handler, H-7
summary, H-1
Functions and commands, ODT-11, 5-3

General registers, PAL-11S, 1-6
Global symbol directory, PAL-11S, 1-30
Global symbols, LINK-11S, 3-2
PAL-11S, 1-5

.GLOBL directive, PAL-11S, 1-18
Grouping of commands, ED-11, D-3

Halts, PAL-11A software error, 2-26

Immediate and Deferred Immediate
(Absolute) Mode, PAL-11A, 2-12
PAL-11S, 1-15
Index Mode, PAL-11A, 2-11
PAL-11S, 1-14
Initial dialogue, PAL-11A, 2-18
PAL-11S, 1-24
Initial operating procedures, PAL-11A, C-9
PAL-11S, B-11
Initialization, 7-4
Input and output, LINK-11S, 3-3
Input/output commands, ED-11, 4-4, D-1
Instruction forms, PAL-11A, 2-14
PAL-11S, 1-17
Instruction mnemonic, PAL-11A, 2-3
Instruction operand fields, PAL-11A, 2-14
PAL-11S, 1-17
Instruction summary, IOX, G-1
Instructions, Assembly and linking, J-1
PAL-11A, C-3
PAL-11S, B-2
Integer conversions, octal-decimal, L-1
Internal information, IOX, 7-19
Internal registers, accessing, ODT-11, 5-6
Internal symbol, PAL-11S, 1-5
Introduction, ODT-11, 5-1
IOX,
adding devices to, 7-21
buffer size, 7-5
byte count, 7-8
communication with, 7-1
data transfers, 7-11
device assignment table, 7-3
Done Bit, 7-7
End-of-File Bit, 7-7
End-of-Medium Bit, 7-7
terminating, 7-6
example of program using, 7-17

Index-3
INDEX (Cont.)

IOX (cont.),
fatal errors, 7-17, G-2
instruction summary, G-1
internal information, 7-19
Mode Byte, 7-5
modes, 7-8
program flow summary, G-1
reenabling the reader and
restarting, 7-16
Status Byte, 7-6
using, 7-3

Label,
PAL-11A, 2-3
PAL-11S, 1-3
.LIMIT directive, PAL-11S, 1-23
Line-oriented command
properties, ED-11, 4-3
LINK-11S,
absolute and relocatable
program sections, 3-2
error procedure and
messages, 3-7
global symbols, 3-2
input and output, 3-3
load map, 3-4
load modules, 3-3
loading and command string,
3-5
object module, 3-3
operating procedures, 3-5
Linking and assembly instruc-
tions, 9-1
Linking and relocation,
PAL-11S, 1-11
Listing, assembly,
PAL-11A, 2-24
PAL-11S, 1-30
Load map, LINK-11S, 3-4
Load modules, LINK-11S, 3-3
Loader,
Absolute, 6-1, 6-10, F-3
Bootstrap, 6-1, F-1
PAL-11S Absolute, 1-24
Loading,
bootstrap tapes into core,
6-8
PAL-11A, 2-18
PAL-11S, 1-24
unused trap vectors, 9-5
Loading and command string,
LINK-11S, 3-5
Loading and verifying the
Bootstrap Loader, 6-7
Location counter, PAL-11S,
assembly, 1-10

Logical and arithmetic operators,
PAL-11A, 2-8
PAL-11S, 1-8

Mark, ED-11, 4-3
Mathematical constants in octal,
L-7
Mnemonic, PAL-11A instruction,
2-3

Mode,
IOX, 7-8
PAL-11A, Autodecrement, 2-11
PAL-11A, Autoincrement Deferred, 2-11
PAL-11A, Deferred Autodecrement, 2-11
PAL-11A, Deferred Index, 2-12
PAL-11A, Immediate and Deferred
Immediate (Absolute), 2-12
PAL-11A, Index, 2-11
PAL-11A, Relative and Deferred
Relative, 2-13
Mode Byte, IOX, 7-5
Mode of expressions, PAL-11S, 1-9

Negative numbers, PAL-11S, 1-8
Non-OTS routines, FPMP-11, H-7
Notation, scales of, L-6
Numbers,
decimal, PAL-11S, 1-8
negative, PAL-11S, 1-8
octal, PAL-11S, 1-8
PAL-11A, 2-7
positive, PAL-11S, 1-8

Object module, LINK-11S, 3-3
Object module output, PAL-11S,
1-30
Octal-decimal integer conversions,
L-1
Octal numbers, PAL-11S, 1-8
ODT,
command syntax, 5-2
priority level, 5-10
ODT-11,
accessing internal registers,
5-6
breakpoints, 5-6
calculating offsets, 5-9
commands and functions, 5-3
introduction, 5-1
opening, changing, and closing
locations, 5-4

Index-4
INDEX (Cont.)

ODT-11 (cont.),
operating procedures, 5-20
running the program, 5-7
searches, 5-8, 5-18
starting and restarting, 5-21
summary, E-1
teletype interrupt, 5-10

ODT-11X,
breakpoints, 5-12, 5-14
calculating offsets, 5-11
communication and data flow, 5-15
error detection, 5-13
opening, changing, and closing locations, 5-10
programming considerations, 5-14
single-Instruction Mode, 5-12

Offsets,
calculating, ODT-11, 5-9
calculating, ODT-11X, 5-11
PAL-11S byte, 1-17

One device, single buffer
transfer on, 7-14

Opening, changing, and closing locations,
ODT-11, 5-4
ODT-11X, 5-10

Operand,
PAL-11A, 2-3
PAL-11S, 1-4

Operand fields, instruction,
PAL-11A, 2-14
PAL-11S, 1-17

Operating procedures,
ED-11, 4-10, D-4
LINK-11S, 3-5
ODT-11, 5-20
PAL-11A, 2-17
PAL-11A initial, C-9
PAL-11S, 1-24
PAL-11S initial, B-11

Operator,
PAL-11A, 2-3
PAL-11S, 1-3

Operators,
PAL-11A, arithmetic and logical, 2-8
PAL-11S, arithmetic and logical, 1-8

OTS routines, FPMP-11, H-2

Output formats, dumps, 6-17
Output, PAL-11S object module, 1-30

Page size,
PAL-11A, 2-4
PAL-11S, 1-4

PAL-11A,
Address Mode syntax, C-2
addressing, 2-9
arithmetic and logical operators, 2-8
ASCII conversion, 2-8
.ASCII directive, 2-17
.assembler directives, 2-3, 2-15
assembly dialogue, 2-23
assembly listing, 2-24
Assembly Location Counter, 2-8
Autodecrement Mode, 2-11
Autoincrement Mode, 2-10
blank operator field, 2-16
.BYTE directive, 2-17
character set, 2-2
comments, 2-4
Deferred Autodecrement Mods, 2-11
Deferred Autoincrement Mods, 2-11

Deferred Index Mode, 2-12
direct assignment, 2-5
.END directive, 2-16
.EOT directive, 2-15
error code, 2-7
error codes, 2-25, C-8
.EVEN directive, 2-16
expressions, 2-7
format control, 2-4
Immediate and Deferred Immediate (Absolute) Mode, 2-12

Index Mode, 2-11
initial dialogue, 2-18
initial operating procedures, C-9

instruction forms, 2-14
instruction mnemonic, 2-3
instruction operand fields, 2-14
instructions, C-3
label, 2-3
loading, 2-18
numbers, 2-7
operand, 2-3
operating procedures, 2-17
operator, 2-3
page size, 2-4
permanent symbols, 2-5
Program Counter, 2-9
Register Mode, 2-10
register symbols, 2-6
Relative and Deferred Relative Mode, 2-13
software error halts, 2-26
special characters, C-1
statements, 2-2
INDEX (Cont.)

PAL-11A (cont.),
user-defined symbols, 2-5
.WORD directive, 2-16
PAL-11S
absolute expression, 1-9
Absolute Loader, 1-24
Address Mode syntax, B-2
addressing, 1-12
ASCII conversion, 1-8
.ASCII directive, 1-21
assembler directives, 1-18,
B-8
assembly listing, 1-30
Autodecrement Mode, 1-14
Autoincrement Mode, 1-13
.BYTE directive, 1-21
byte offset, 1-17
character set, 1-2
comments, 1-4
conditional assembly directives, 1-23
control format, 1-4
CTRL/U, 1-24
decimal numbers, 1-8
Deferred Autodecrement
Mode, 1-14
Deferred Autoincrement
Mode, 1-13
Deferred Index Mode, 1-14
Deferred Register Mode, 1-13
direct assignment, 1-6
Editor (ED-11), 1-2
.END directive, 1-20
.EOT directive, 1-20
error codes, 1-31, B-10
.EVEN directive, 1-20
expressions, 1-7
external symbol, 1-5
general registers, 1-6
global symbol, 1-5
global symbol directory, 1-30
.GLOBL directive, 1-18
Immediate and Deferred
Immediate (Absolute) Modes,
1-15
Index Mode, 1-14
initial dialogue, 1-24
initial operating procedures,
B-11
instruction forms, 1-17
instruction operand fields,
1-17
instructions, B-2
internal symbol, 1-5
label, 1-3
.LIMIT directive, 1-23
loading, 1-24
negative numbers, 1-8
object module output, 1-30
PAL-11S (cont.),
octal numbers, 1-8
operand, 1-4
operating procedures, 1-24
operator, 1-3
page size, 1-4
permanent symbols, 1-5
positive numbers, 1-8
Program Counter, 1-12
program section directives
(.ASELECT and .CSECTION), 1-19
.RAD50 directive, 1-22
Register Mode, 1-12
register symbols, 1-6
Relative and Deferred Relative
Modes, 1-15
relocatable expression, 1-9
relocation and linking, 1-11
relocation directory, 1-31
RUBOUT, 1-24
software error halts, 1-32
statement format, 1-2
statements, 1-2
symbol table, 1-5
symbols, 1-5
terminators, B-1
text block, 1-31
.TITLE directive, 1-18
truncation, 1-8
user-defined symbols, 1-5
.WORD directive, 1-20
Paper tape bootstraps, 6-2
Permanent symbols,
PAL-11A, 2-5
PAL-11S, 1-5
Pointer-positioning commands,
ED-11, D-2
Position-independent code, writing,
9-1
Positive numbers, PAL-11S, 1-8
Powers of two, L-5
Priority level, ODT, 5-10
Program Counter,
PAL-11A, 2-9
PAL-11S, 1-12
Program flow summary, IOX, C-1
Program section directives
(.ASELECT and .CSECTION) PAL-11S,
1-19
Programming considerations,
ODT-11X, 5-14
Q error, PAL-11S, 1-2
Index-6
INDEX (Cont.)

.RAD50 directive, PAL-11S, 1-22
Real-time capability, 7-1
Reenabling the reader and
   restarting, IOX, 7-16
Register contents, altering,
   9-6
Register Mode,
   PAL-11A, 2-10
   PAL-11S, 1-12
Register symbols,
   PAL-11A, 2-6
   PAL-11S, 1-6
Registers, general, PAL-11S,
   1-6
Relative and Deferred Relative
   Mode,
   PAL-11A, 2-13
   PAL-11S, 1-15
Relocatable expression,
   PAL-11S, 1-9
Relocation and linking,
   PAL-11S, 1-11
Relocation directory, PAL-11S,
   1-31
Restarting and starting ODT-11,
   5-21
Restarting ED-11, 4-11
Routines accessed via trap
   handler, FPMP-11, H-7
RUBOUT, PAL-11S, 1-24
Running the program, ODT-11,
   5-7

Scales of notation, L-6
Search commands, ED-11, 4-7,
   D-2
Searches, ODT-11, 5-8, 5-18
Single buffer transfer on
   one device, 7-14
Single-Instruction Mode,
   ODT-11X, 5-12
Software error halts,
   ED-11, 4-17
   PAL-11A, 2-26
   PAL-11S, 1-32
Special characters, PAL-11A,
   C-1
Standard PDP-11 abbreviations,
   K-1
Starting and restarting ODT-11,
   5-21
Starting ED-11, 4-11
Statement format, PAL-11S, 1-2
Statements,
   PAL-11A, 2-2
   PAL-11S, 1-2
Status Byte, IOX, 7-6

Storage maps, dumps, 6-17
Subroutines, 9-7
Summary,
   FPMP-11, H-1
   ODT-11, E-1
Symbol,
   external, PAL-11S, 1-5
   global, PAL-11S, 1-5
   internal, PAL-11S, 1-5
Symbol table, PAL-11S, 1-5
Symbols,
   ED-11, D-3
   PAL-11A permanent, 2-5
   PAL-11A register, 2-6
   PAL-11A user-defined, 2-5
   PAL-11S, 1-5
   permanent, PAL-11S, 1-5
   register, PAL-11S, 1-6
   user-defined, PAL-11S, 1-5

Table, symbol, PAL-11S, 1-5
Tables, conversion, L-1
Tape duplication, 1-1
Techniques, coding, 9-6
Teletype interrupt, ODT-11, 5-19
Terminators, PAL-11S, B-1
Text block, PAL-11S, L-31
.TITLE directive, PAL-11S, 1-18
Trap vectors, loading unused, 9-5
Truncation, PAL-11S, 1-8
Two, powers of, L-5

Unused trap vectors, loading, 9-5
User-defined symbols,
   PAL-11A, 2-5
   PAL-11S, 1-5
Using IOX, 7-3

Verifying and loading the bootstrap loader, 6-7

.WORD directive,
   PAL-11A, 2-16
   PAL-11S, 1-20
Writing position-independent
   code, 9-1

Index-7
NOTE: This form is for document comments only. Problems with software should be reported on a Software Problem Report (SPR) form.

Did you find errors in this manual? If so, specify by page.

____________________________________________________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

Did you find this manual understandable, usable, and well-organized? Please make suggestions for improvement.

____________________________________________________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

Is there sufficient documentation on associated system programs required for use of the software described in this manual? If not, what material is missing and where should it be placed?

____________________________________________________________________________________

____________________________________________________________________________________

____________________________________________________________________________________

Please indicate the type of user/reader that you most nearly represent.

☐ Assembly language programmer
☐ Higher-level language programmer
☐ Occasional programmer (experienced)
☐ User with little programming experience
☐ Student programmer
☐ Non-programmer interested in computer concepts and capabilities

Name_________________________________________ Date______________________

Organization_____________________________________________

Street_____________________________________________________

City________________________ State________ Zip Code_________

or

Country

If you require a written reply, please check here. ☐