Additional copies of this manual or other Intel literature may be obtained from:

Literature Department
Intel Corporation
3065 Bowers Avenue
Santa Clara, CA 95051

The information in this document is subject to change without notice.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

No part of this document may be copied or reproduced in any form or by any means without the prior written consent of Intel Corporation.

The following are trademarks of Intel Corporation and may be used only to describe Intel products:

<table>
<thead>
<tr>
<th>i</th>
<th>iSBC</th>
<th>Multimodule</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICE</td>
<td>Library Manager</td>
<td>PROMPT</td>
</tr>
<tr>
<td>iCS</td>
<td>MCS</td>
<td>Promware</td>
</tr>
<tr>
<td>Issue</td>
<td>Megachassis</td>
<td>RMX</td>
</tr>
<tr>
<td>Intel</td>
<td>Micromap</td>
<td>UPI</td>
</tr>
<tr>
<td>Intellec</td>
<td>Multibus</td>
<td>μScope</td>
</tr>
</tbody>
</table>

and the combination of ICE, iCS, iSBC, MCS, or RMX and a numerical suffix.
This document describes the purpose and the use of the ICE-88 In-Circuit Emulator for the Intel 8088 microprocessor.

The ICE-88 module is an optional addition to the Intellec Microcomputer Development System. The ICE-88 module aids in testing and modification of the hardware and software for new products designed around the 8088 microprocessor.

Chapter 1 describes the mission of the ICE-88 emulator as a developmental aid for system designs based on Intel’s 8088 microprocessor.

Chapter 2 gives step-by-step instructions for installing the ICE-88 hardware in the Intellec chassis and connecting the ICE-88 emulator to the user prototype system.

Chapter 3 presents a hands-on debugging session with the ICE-88 emulator.

Chapter 4 describes the elements of the ICE-88 emulator command language, and the notations, conventions, and syntactic rules used in this manual.

Chapter 5 defines the operands, operators, and expressions used in the ICE-88 commands.

Chapter 6 contains discussions and specifications of the emulation and trace control commands.

Chapter 7 contains discussions and specifications of the interrogation and utility commands.

Chapter 8 contains discussions and specifications of the compound and macro commands used in ICE-88 emulator operation.

Appendix A is a list of all the ICE-88 emulator keywords (literals), and their abbreviations, in alphabetical order.

Appendix B is a list of ICE-88 emulator error and warning messages with interpretations.

Appendix C contains a syntactic summary of the ICE-88 emulator commands.

Appendix D presents the electrical and physical characteristics of the ICE-88 emulator.

Appendix E presents the 8088 assembler instructions in hexadecimal order.

To use this manual effectively, you need to understand the 8086/8088 architecture and the technique of programming and debugging. The following publications contain detailed information related to this manual.
<table>
<thead>
<tr>
<th>Title</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISIS-II User's Guide</td>
<td>9800306</td>
</tr>
<tr>
<td>8080/8085 Assembly Language Programming Manual</td>
<td>9800301</td>
</tr>
<tr>
<td>ISIS-II 8080/8085 Macro Assembler Operator's Manual</td>
<td>9800292</td>
</tr>
<tr>
<td>MCS-86™ Assembly Language Reference Manual</td>
<td>9800640</td>
</tr>
<tr>
<td>MCS-86™ Assembler Operating Instructions for ISIS-II Users</td>
<td>9800641</td>
</tr>
<tr>
<td>MCS-86™ User's Manual</td>
<td>9800722</td>
</tr>
<tr>
<td>PL/M-86 Programming Manual</td>
<td>9800466</td>
</tr>
<tr>
<td>PL/M-86 Operator's Manual</td>
<td>9800478</td>
</tr>
<tr>
<td>MCS-86™ Software Development Utilities Operating for ISIS-II Users</td>
<td>9800639</td>
</tr>
<tr>
<td>Intellec® Microcomputer Development Systems Hardware</td>
<td></td>
</tr>
<tr>
<td>Reference Manual</td>
<td></td>
</tr>
<tr>
<td>Intellec® Series II Hardware Reference Manual</td>
<td></td>
</tr>
<tr>
<td>Intellec® Series II Installation and Service Manual</td>
<td></td>
</tr>
<tr>
<td>A Guide to Intellec® Microcomputer Development Systems</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9800565</td>
</tr>
</tbody>
</table>
## CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER 1</th>
<th>INTRODUCTION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ICE-88™ In-Circuit Emulator</td>
<td>1-1</td>
</tr>
<tr>
<td></td>
<td>Integrated Hardware/Software Development</td>
<td>1-2</td>
</tr>
<tr>
<td></td>
<td>ICE-88™ In-Circuit Emulation</td>
<td>1-2</td>
</tr>
<tr>
<td></td>
<td>User Program Execution Control</td>
<td>1-2</td>
</tr>
<tr>
<td></td>
<td>Memory Mapping</td>
<td>1-3</td>
</tr>
<tr>
<td></td>
<td>Symbolic Debugging</td>
<td>1-4</td>
</tr>
<tr>
<td></td>
<td>Display</td>
<td>1-4</td>
</tr>
<tr>
<td></td>
<td>Operating Modes</td>
<td>1-5</td>
</tr>
<tr>
<td></td>
<td>Emulation</td>
<td>1-5</td>
</tr>
<tr>
<td></td>
<td>Interrogation and Utility</td>
<td>1-6</td>
</tr>
<tr>
<td></td>
<td>Macro and Compound Commands</td>
<td>1-8</td>
</tr>
<tr>
<td></td>
<td>ICE-88™ Architecture</td>
<td>1-8</td>
</tr>
<tr>
<td></td>
<td>ICE-88™ Software</td>
<td>1-9</td>
</tr>
<tr>
<td></td>
<td>ICE-88™ Firmware</td>
<td>1-9</td>
</tr>
<tr>
<td></td>
<td>ICE-88™ Hardware</td>
<td>1-9</td>
</tr>
<tr>
<td></td>
<td>Generalized Development Cycle With ICE-88™</td>
<td>1-10</td>
</tr>
<tr>
<td></td>
<td>A Generalized Emulation Session</td>
<td>1-12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 2</th>
<th>ICE-88™ INSTALLATION PROCEDURES</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ICE-88™ Components</td>
<td>2-1</td>
</tr>
<tr>
<td></td>
<td>Required and Optional Hardware</td>
<td>2-1</td>
</tr>
<tr>
<td></td>
<td>Hardware Installation Procedures</td>
<td>2-2</td>
</tr>
<tr>
<td></td>
<td>Installation Procedure for Intellec Model</td>
<td>2-2</td>
</tr>
<tr>
<td></td>
<td>800 and 888</td>
<td>2-2</td>
</tr>
<tr>
<td></td>
<td>Installation Procedure for Intellec Series II Model</td>
<td>2-3</td>
</tr>
<tr>
<td></td>
<td>220 and 230</td>
<td>2-3</td>
</tr>
<tr>
<td></td>
<td>Confidence Test</td>
<td>2-4</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 3</th>
<th>SAMPLE ICE-88™ SESSIONS AT THE TERMINAL</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>How To Use This Chapter</td>
<td>3-1</td>
</tr>
<tr>
<td></td>
<td>Analysis of the Sample Program</td>
<td>3-2</td>
</tr>
<tr>
<td></td>
<td>Hands-On Demonstration</td>
<td>3-5</td>
</tr>
<tr>
<td></td>
<td>Session 1</td>
<td>3-6</td>
</tr>
<tr>
<td></td>
<td>Session 2</td>
<td>3-12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 4</th>
<th>ELEMENTS OF THE ICE-88™ COMMAND LANGUAGE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Introduction</td>
<td>4-1</td>
</tr>
<tr>
<td></td>
<td>Notation and Conventions Used in This Manual</td>
<td>4-3</td>
</tr>
<tr>
<td></td>
<td>Syntactic Rules Used in the Manual</td>
<td>4-4</td>
</tr>
<tr>
<td></td>
<td>Character Set</td>
<td>4-6</td>
</tr>
<tr>
<td></td>
<td>Introduction to Tokens</td>
<td>4-7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PAGE</th>
<th>Keywords</th>
<th>4-7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Reference Keywords</td>
<td>4-7</td>
</tr>
<tr>
<td></td>
<td>Registers</td>
<td>4-8</td>
</tr>
<tr>
<td></td>
<td>Status Registers</td>
<td>4-10</td>
</tr>
<tr>
<td></td>
<td>8088 Pin References</td>
<td>4-11</td>
</tr>
<tr>
<td></td>
<td>Emulation Registers</td>
<td>4-12</td>
</tr>
<tr>
<td></td>
<td>Command Keywords</td>
<td>4-12</td>
</tr>
<tr>
<td></td>
<td>Simple Commands</td>
<td>4-12</td>
</tr>
<tr>
<td></td>
<td>Compound Commands</td>
<td>4-13</td>
</tr>
<tr>
<td></td>
<td>Macro Commands</td>
<td>4-13</td>
</tr>
<tr>
<td></td>
<td>Utility Command Keywords</td>
<td>4-14</td>
</tr>
<tr>
<td></td>
<td>Number Base and Radix Commands</td>
<td>4-14</td>
</tr>
<tr>
<td></td>
<td>Memory Mapping Command Keywords</td>
<td>4-14</td>
</tr>
<tr>
<td></td>
<td>Hardware Register Command Keywords</td>
<td>4-15</td>
</tr>
<tr>
<td></td>
<td>Memory and Port Contents Command Keywords</td>
<td>4-15</td>
</tr>
<tr>
<td></td>
<td>Symbol Table and Statement Number Table Command Keywords</td>
<td>4-16</td>
</tr>
<tr>
<td></td>
<td>Emulation Control Command Keywords</td>
<td>4-16</td>
</tr>
<tr>
<td></td>
<td>Trace Control Commands</td>
<td>4-17</td>
</tr>
<tr>
<td></td>
<td>User Names</td>
<td>4-17</td>
</tr>
<tr>
<td></td>
<td>Symbols</td>
<td>4-17</td>
</tr>
<tr>
<td></td>
<td>Statement Numbers</td>
<td>4-18</td>
</tr>
<tr>
<td></td>
<td>Special Tokens</td>
<td>4-18</td>
</tr>
<tr>
<td></td>
<td>Entering Commands at the Console</td>
<td>4-19</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 5</th>
<th>EXPRESSIONS</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Operands</td>
<td>5-2</td>
</tr>
<tr>
<td></td>
<td>Numeric Constants</td>
<td>5-2</td>
</tr>
<tr>
<td></td>
<td>Masked Constants</td>
<td>5-3</td>
</tr>
<tr>
<td></td>
<td>Keyword References</td>
<td>5-3</td>
</tr>
<tr>
<td></td>
<td>Symbolic References</td>
<td>5-4</td>
</tr>
<tr>
<td></td>
<td>Statement Number Reference</td>
<td>5-5</td>
</tr>
<tr>
<td></td>
<td>Memory References</td>
<td>5-5</td>
</tr>
<tr>
<td></td>
<td>Typed Memory Reference</td>
<td>5-6</td>
</tr>
<tr>
<td></td>
<td>Port References</td>
<td>5-7</td>
</tr>
<tr>
<td></td>
<td>String Constants</td>
<td>5-7</td>
</tr>
<tr>
<td></td>
<td>Parenthesized Expressions</td>
<td>5-9</td>
</tr>
<tr>
<td></td>
<td>Operators</td>
<td>5-9</td>
</tr>
<tr>
<td></td>
<td>Classes of Operators</td>
<td>5-9</td>
</tr>
<tr>
<td></td>
<td>Arithmetic Operators</td>
<td>5-9</td>
</tr>
<tr>
<td></td>
<td>Content Operators</td>
<td>5-12</td>
</tr>
<tr>
<td></td>
<td>Relational Operators</td>
<td>5-12</td>
</tr>
<tr>
<td></td>
<td>Logical Operators</td>
<td>5-12</td>
</tr>
<tr>
<td></td>
<td>Arithmetic and Logical Semantic Rules</td>
<td>5-14</td>
</tr>
<tr>
<td></td>
<td>How Expressions are Evaluated</td>
<td>5-15</td>
</tr>
<tr>
<td></td>
<td>“Case Studies” in Evaluating Expressions</td>
<td>5-15</td>
</tr>
<tr>
<td></td>
<td>Command Contexts</td>
<td>5-22</td>
</tr>
</tbody>
</table>
## CHAPTER 6
**EMULATION AND TRACE CONTROL COMMANDS**

<table>
<thead>
<tr>
<th>Emulation Control Commands</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discussion</td>
<td>6-2</td>
</tr>
<tr>
<td>Execution Match Condition</td>
<td>6-4</td>
</tr>
<tr>
<td>Non-Execution Match Condition</td>
<td>6-6</td>
</tr>
<tr>
<td>Address Match Range</td>
<td>6-6</td>
</tr>
<tr>
<td>Match Status List</td>
<td>6-7</td>
</tr>
<tr>
<td>Data Match Range</td>
<td>6-8</td>
</tr>
<tr>
<td>Segment Register Usage</td>
<td>6-8</td>
</tr>
<tr>
<td>Match Condition Restrictions</td>
<td>6-8</td>
</tr>
<tr>
<td>Setting the Go-Register</td>
<td>6-10</td>
</tr>
<tr>
<td>Setting Tracepoint Registers</td>
<td>6-11</td>
</tr>
<tr>
<td>Command Signal Timeout</td>
<td>6-12</td>
</tr>
<tr>
<td>Emulation Timer</td>
<td>6-12</td>
</tr>
<tr>
<td>Set Breakpoint Register Command</td>
<td>6-13</td>
</tr>
<tr>
<td>Set Tracepoint Register Command</td>
<td>6-14</td>
</tr>
<tr>
<td>Go Command</td>
<td>6-15</td>
</tr>
<tr>
<td>Set Go-Register (GR) Command</td>
<td>6-16</td>
</tr>
<tr>
<td>STEP Command</td>
<td>6-17</td>
</tr>
<tr>
<td>Display Emulation Register Command</td>
<td>6-18</td>
</tr>
<tr>
<td>Set CLOCK Command</td>
<td>6-19</td>
</tr>
<tr>
<td>Display CLOCK Command</td>
<td>6-19</td>
</tr>
<tr>
<td>Set RTIMEOUT Command</td>
<td>6-20</td>
</tr>
<tr>
<td>Display RTIMEOUT Command</td>
<td>6-20</td>
</tr>
<tr>
<td>ENABLE/DISABLE RDY Command</td>
<td>6-21</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Trace Control Command</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discussion</td>
<td>6-22</td>
</tr>
<tr>
<td>Trace Display Mode</td>
<td>6-23</td>
</tr>
<tr>
<td>Moving the Buffer Pointer</td>
<td>6-23</td>
</tr>
<tr>
<td>Displaying Trace Data</td>
<td>6-24</td>
</tr>
<tr>
<td>Trace Display Formats</td>
<td>6-25</td>
</tr>
<tr>
<td>Display of Trace Data in Frames Mode</td>
<td>6-25</td>
</tr>
<tr>
<td>Display of Trace Data in Instructions Mode</td>
<td>6-26</td>
</tr>
<tr>
<td>Operands Fields</td>
<td>6-26</td>
</tr>
<tr>
<td>Display of Cycles Instruction Mode</td>
<td>6-27</td>
</tr>
<tr>
<td>Gaps in Trace Instruction Mode</td>
<td>6-28</td>
</tr>
<tr>
<td>Extended Example of Trace Displays</td>
<td>6-29</td>
</tr>
<tr>
<td>Set TRACE Display Mode Command</td>
<td>6-31</td>
</tr>
<tr>
<td>ENABLE/DISABLE TRACE Command</td>
<td>6-32</td>
</tr>
<tr>
<td>Display TRACE Command</td>
<td>6-33</td>
</tr>
<tr>
<td>MOVE, OLDEST, and NEWEST Commands</td>
<td>6-33</td>
</tr>
<tr>
<td>PRINT Command</td>
<td>6-34</td>
</tr>
</tbody>
</table>

## CHAPTER 7
**INTERROGATION AND UTILITY COMMANDS**

<table>
<thead>
<tr>
<th>Utility Commands Involving ISIS-II</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Discussion</td>
<td>7-2</td>
</tr>
<tr>
<td>ICE-88™ Command</td>
<td>7-5</td>
</tr>
<tr>
<td>EXIT Command</td>
<td>7-6</td>
</tr>
<tr>
<td>LOAD Command</td>
<td>7-7</td>
</tr>
<tr>
<td>SAVE Command</td>
<td>7-8</td>
</tr>
<tr>
<td>LIST Command</td>
<td>7-9</td>
</tr>
<tr>
<td>Number Bases and Radix Commands</td>
<td>7-10</td>
</tr>
<tr>
<td>Discussion</td>
<td>7-10</td>
</tr>
<tr>
<td>Console Input Radixes: SUFFIX Command</td>
<td>7-10</td>
</tr>
<tr>
<td>Console Output Radixes: Base Command</td>
<td>7-11</td>
</tr>
<tr>
<td>Set or Display Console Input Radix Commands</td>
<td>7-12</td>
</tr>
<tr>
<td>Set or Display Console Output Radix Commands</td>
<td>7-13</td>
</tr>
<tr>
<td>Hardware Register Commands</td>
<td>7-14</td>
</tr>
<tr>
<td>Set Register Command</td>
<td>7-14</td>
</tr>
<tr>
<td>RESET HARDWARE Command</td>
<td>7-18</td>
</tr>
<tr>
<td>Memory Mapping Commands</td>
<td>7-19</td>
</tr>
<tr>
<td>Discussion</td>
<td>7-20</td>
</tr>
<tr>
<td>MAP DISK Command</td>
<td>7-24</td>
</tr>
<tr>
<td>MAP INTELLEC Command</td>
<td>7-25</td>
</tr>
<tr>
<td>Set MAP Status Command</td>
<td>7-26</td>
</tr>
<tr>
<td>Display MAP Status Command</td>
<td>7-28</td>
</tr>
<tr>
<td>RESET MAP Command</td>
<td>7-29</td>
</tr>
<tr>
<td>Set Memory and Port Contents Commands</td>
<td>7-30</td>
</tr>
<tr>
<td>Setting Memory Contents</td>
<td>7-30</td>
</tr>
<tr>
<td>Port Content References</td>
<td>7-35</td>
</tr>
<tr>
<td>Set Memory Command</td>
<td>7-37</td>
</tr>
<tr>
<td>Set Input/Output Port Contents Command</td>
<td>7-38</td>
</tr>
<tr>
<td>Symbol Table and Statement Number Table Commands</td>
<td>7-39</td>
</tr>
<tr>
<td>Discussion</td>
<td>7-39</td>
</tr>
<tr>
<td>DEFINE Symbol Command</td>
<td>7-45</td>
</tr>
<tr>
<td>Display Symbols Command</td>
<td>7-46</td>
</tr>
<tr>
<td>Display Statement Numbers Command</td>
<td>7-47</td>
</tr>
<tr>
<td>Display Modules Command</td>
<td>7-48</td>
</tr>
<tr>
<td>Change Symbol Command</td>
<td>7-49</td>
</tr>
<tr>
<td>REMOVE Symbols Command</td>
<td>7-50</td>
</tr>
<tr>
<td>TYPE Command</td>
<td>7-51</td>
</tr>
<tr>
<td>SET DOMAIN Command</td>
<td>7-52</td>
</tr>
<tr>
<td>RESET DOMAIN Command</td>
<td>7-53</td>
</tr>
<tr>
<td>Display Commands</td>
<td>7-54</td>
</tr>
<tr>
<td>Discussion</td>
<td>7-54</td>
</tr>
<tr>
<td>Registers</td>
<td>7-54</td>
</tr>
<tr>
<td>Status Register</td>
<td>7-56</td>
</tr>
<tr>
<td>Pin References</td>
<td>7-58</td>
</tr>
<tr>
<td>General Formats for Numeric Values</td>
<td>7-58</td>
</tr>
<tr>
<td>Display NESTING Command</td>
<td>7-68</td>
</tr>
<tr>
<td>The EVALUATE Command</td>
<td>7-68</td>
</tr>
<tr>
<td>Display Processor and Status Registers Command</td>
<td>7-70</td>
</tr>
<tr>
<td>Display Memory Command</td>
<td>7-71</td>
</tr>
<tr>
<td>Display I/O Command</td>
<td>7-72</td>
</tr>
<tr>
<td>Display STACK Command</td>
<td>7-73</td>
</tr>
<tr>
<td>Display Boolean Command</td>
<td>7-74</td>
</tr>
<tr>
<td>Display NESTING Command</td>
<td>7-75</td>
</tr>
<tr>
<td>EVALUATE Command</td>
<td>7-76</td>
</tr>
</tbody>
</table>
CHAPTER 8
ICE-88™ ENHANCEMENTS

Compound Commands ........................................ 8-1
REPEAT Command ......................................... 8-1
COUNT Command ............................................ 8-3
IF Command .................................................. 8-5
Nesting Compound Commands ............................... 8-6
Macro Commands ............................................ 8-8
Defining and Invoking Macros ............................ 8-8
Local and Global Defaults ............................... 8-9
Formal and Actual Parameters .......................... 8-10
Details on Macro Expansion ............................. 8-12
Macro Table Commands .................................. 8-12
Saving Macros .............................................. 8-13
Further Examples ......................................... 8-14
Off-line Facilities ......................................... 8-16
INCLUDE Command ........................................ 8-16
WRITE Command .......................................... 8-17

APPENDIX A
ICE-88™ KEYWORDS AND THEIR ABBREVIATIONS

APPENDIX B
ERROR MESSAGES

APPENDIX C
ICE-88™ COMMAND SYNTAX SUMMARY

APPENDIX D
ELECTRICAL AND PHYSICAL CHARACTERISTICS OF ICE-88™

APPENDIX E
INSTRUCTIONS IN HEXADECIMAL ORDER

TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Summary of Emulation Commands</td>
<td>1-6</td>
</tr>
<tr>
<td>1-2</td>
<td>Summary of Basic Interrogation and Utility Commands</td>
<td>1-7</td>
</tr>
<tr>
<td>3-1</td>
<td>Key Addresses in CARS Logic</td>
<td>3-16</td>
</tr>
<tr>
<td>4-1</td>
<td>Definition of GO Command Functions</td>
<td>4-2</td>
</tr>
<tr>
<td>4-2</td>
<td>Notational Symbols</td>
<td>4-3</td>
</tr>
<tr>
<td>4-3</td>
<td>Classes of Hardware Elements</td>
<td>4-8</td>
</tr>
<tr>
<td>5-1</td>
<td>Elements of Numeric Constants</td>
<td>5-3</td>
</tr>
<tr>
<td>5-2</td>
<td>ASCII Printing Characters and CODES (20H-7EH)</td>
<td>5-8</td>
</tr>
<tr>
<td>5-3</td>
<td>ICE™ Operators</td>
<td>5-10</td>
</tr>
<tr>
<td>5-4</td>
<td>Classes of Operators</td>
<td>5-11</td>
</tr>
<tr>
<td>5-5</td>
<td>Content Operators</td>
<td>5-12</td>
</tr>
<tr>
<td>5-6</td>
<td>Arithmetic and Logical Semantic Rules</td>
<td>5-14</td>
</tr>
<tr>
<td>5-7</td>
<td>Conditions and Notation for Examples</td>
<td>5-17</td>
</tr>
<tr>
<td>5-8</td>
<td>Representative Cases of Expressions</td>
<td>5-18</td>
</tr>
<tr>
<td>5-9</td>
<td>Command Contexts</td>
<td>5-23</td>
</tr>
<tr>
<td>7-1</td>
<td>8088 General Registers</td>
<td>7-14</td>
</tr>
<tr>
<td>7-2</td>
<td>Pointer Registers</td>
<td>7-15</td>
</tr>
<tr>
<td>7-3</td>
<td>Index Registers</td>
<td>7-15</td>
</tr>
<tr>
<td>7-4</td>
<td>Segment Registers</td>
<td>7-15</td>
</tr>
<tr>
<td>7-5</td>
<td>Status Registers</td>
<td>7-15</td>
</tr>
<tr>
<td>7-6</td>
<td>Pin References</td>
<td>7-16</td>
</tr>
<tr>
<td>7-7</td>
<td>Flag References</td>
<td>7-16</td>
</tr>
<tr>
<td>7-8</td>
<td>Symbolic References and Statement References</td>
<td>7-41</td>
</tr>
<tr>
<td>7-9</td>
<td>Classes of Hardware Elements</td>
<td>7-54</td>
</tr>
<tr>
<td>7-10</td>
<td>Numeric Value Display Formats</td>
<td>7-58</td>
</tr>
<tr>
<td>7-11</td>
<td>Display Values Per Line</td>
<td>7-60</td>
</tr>
<tr>
<td>8-1</td>
<td>Tracking a COUNT Command</td>
<td>8-4</td>
</tr>
<tr>
<td>FIGURE</td>
<td>TITLE</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------</td>
<td>--------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>1-1</td>
<td>8088 CPU Functional Block Diagram</td>
<td>1-1</td>
</tr>
<tr>
<td>1-2</td>
<td>ICE-88™ Functional Block Diagram</td>
<td>1-10</td>
</tr>
<tr>
<td>1-3</td>
<td>Typical Development Cycle with ICE™ Module</td>
<td>1-11</td>
</tr>
<tr>
<td>3-1</td>
<td>CARS Module Listing</td>
<td>3-3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3-2</td>
<td>DELAY Module Listing</td>
<td>3-4</td>
</tr>
<tr>
<td>4-1</td>
<td>Example of a GO Command</td>
<td>4-1</td>
</tr>
<tr>
<td>5-1</td>
<td>A Simple Module of Evaluation</td>
<td>5-16</td>
</tr>
<tr>
<td>6-1</td>
<td>Non-Execution Match Condition</td>
<td>6-9</td>
</tr>
</tbody>
</table>
This manual presents the operation of the ICE-88 In-Circuit Emulator for the Intel 8088 microprocessor. As an introduction to the use of this microprocessor design aid, this chapter contains a brief discussion of integrated hardware/software development, in-circuit emulation, and the ICE-88 architecture. Also, a generalized development cycle with the ICE-88 emulator and a generalized emulation session are presented.

**ICE-88 In-Circuit Emulator**

The ICE-88 module provides in-circuit emulation for 8088 microprocessor-based systems. Figure 1-1 shows the functional block diagram of the 8088 CPU. The ICE-88 module consists of three circuit boards which reside in the Intellec Microcomputer Development System. A cable and buffer box connect the Intellec system to the user system by replacing the user's 8088. In this manner the Intellec debug functions are extended into the user system. Using the ICE-88 module, the designer can execute prototype software in continuous or single-step mode and can substitute Intellec equivalents for user devices, such as memory.

![Figure 1-1. 8088 CPU Functional Block Diagram](image)
Integrated Hardware/Software Development

The ICE-88 emulator allows hardware and software development to proceed concurrently. This is more effective than the traditional method of independent hardware and software development followed by a system integration phase. With the ICE-88 emulator, prototype hardware can be added to the system as it is designed. The software and hardware can be used to test each other as the product is developed.

Conceptually, the ICE-88 emulator can be viewed as assisting three stages of development:

1. The ICE-88 emulator can be operated without being connected to the user's system, so its debugging capabilities can be used to facilitate software development before any of the user's hardware is available.
2. To begin integration of software and hardware development efforts, the user's prototype need consist of no more than an 8088 CPU socket. Through the ICE-88 emulator's mapping capabilities, Intellec system equivalents (such as Intellec memory) can be substituted for missing prototype hardware. As each section of the user's hardware is completed, it can be added to the prototype, replacing the Intellec equivalent. Thus each section of the hardware and software can be "system" tested as it becomes available.
3. When the user's prototype is complete, it can be tested using the system software which will drive the final product. The ICE-88 emulator can be used for real time emulation of the 8088 to debug the system as a complete unit.

Thus the ICE-88 emulator provides the user with the ability to debug a prototype or production system at any stage in its development without introducing extraneous hardware or software test tools.

In-Circuit Emulation

The ICE-88 In-Circuit Emulator is a diagnostic tool that is used for testing and debugging the hardware and software of user-designed 8088 microcomputer-based systems. Such testing may begin during the early phases of user system development and may continue throughout the life cycle of the user's system.

The interface between the in-circuit emulator and the user system is implemented at the connector pins of the user system microprocessor chip. These pins carry the information that establishes the characteristics and status of the user system. The interface makes it possible for the in-circuit emulator to continually monitor user operations and to provide control of these operations. More specifically, the in-circuit emulator monitors execution of the user program and controls the conditions under which the user program execution is initiated and terminated.

User Program Execution Control

Starting and stopping execution of the user program at predefined points or conditions is an essential task of the in-circuit emulator as it is often not feasible or desirable to execute the entire user program. For example, a single routine may be executed because either other routines have not yet been coded or because a fault (bug) has been isolated to that routine.

The starting address for execution is readily established by loading a known value into the program counter of the user processor while the processor is inactive. Termination of execution is a more involved procedure which requires the in-circuit emulator to halt the processor when a predetermined multi-condition state exists at
the 8088 pins. This process requires prior storage of state values within the in-circuit emulator hardware and dynamic comparison of these values with the states of specified data, address and/or status pins of the processor. The point at which the user program execution is terminated is known as the breakpoint.

A breakpoint may be specified to cause the user program to halt execution when a given memory location is addressed during a processor fetch (i.e., loaded into the 8088 execution queue). However, very often the operator is more interested in the data value of a memory location or an I/O port. In the latter cases both the type of instruction (read, write, input, or output) and the data value are prespecified and are dynamically compared with the processor pin states. It is also possible to specify "don't care" comparisons with the data pins and thereby halt execution whenever the designated type of instruction is extracted from the queue for execution.

A wide range of breakpoint conditions are possible through comparison of the processor chip states with predesignated values. The full range of breakpoint conditions that may be specified by the operator are presented in subsequent chapters.

**Memory Mapping**

Memory for the user system can be resident in the user system or "borrowed" from the Intellec System through the ICE-88 emulator's mapping capability.

The ICE-88 emulator allows 1 megabyte of user memory to be addressed by the 8088. This user memory space consists of 1024 1K byte segments that can be mapped in 1K blocks to:

1. Physical memory in the user's system,
2. Either of two 1K blocks of the ICE-88 emulator high speed memory,
3. Intellec expansion memory,

The first 64K of Intellec RAM memory is dedicated to Intellec system software. Therefore the RAM boards within the Intellec system that are used by the ICE-88 emulator to store the user program employ effective addresses beyond the 64K byte memory accessible to Intellec system software.

Mapping consists of specifying where each "logical" memory block that the 8088 addresses will physically exist within various physical memories. During emulation the memory map is used to determine the existence and physical location of the logical memory space being referenced by the user program.

If a logical segment of addresses is not activated by associating the segment with a physical memory, the segment is "guarded." A guarded segment is logically nonexistent and any reference to the segment by the user program results in an error. Thus, the ICE-88 emulator can trap memory accesses outside the intended memory for program and data. All blocks are initially guarded following system reset and any segment may be guarded on command after its initial activation.

Mapping enables the user to allocate segments of user memory space to physical memories other than the RAM/ROM of the user system. This feature permits testing of the user program prior to installation of user memory and also provides a convenient means of executing modified code in "borrowed" memory while the bulk of user program is resident within the user system.
Symbolic Debugging

Symbols and PL/M statement numbers may be substituted for numeric values in any of the ICE-88 commands. This allows the user to make symbolic references to I/O ports, memory addresses, and data in a user program. Thus the user need not search listings for addresses of variables or program subroutines.

Symbols can be used to reference variables, procedures, program labels, and source statements. Thus a variable can be displayed or changed by referring to it by name rather than by its absolute location in memory. Using symbols for statement labels, program labels, and procedure names allows the user to set breakpoints or disassemble a section of code into its assembly mnemonics much more easily.

Furthermore, each symbol may have associated with it one of the types BYTE, WORD, INTEGER, SINTEGER (for short, 8-bit integer) or POINTER. Thus when the user examines or modifies a variable from the source program, he doesn't need to remember its type. For example, the command “!VAR” displays the value in memory of variable VAR in a format appropriate to its type, while the command “!VAR = !VAR + 1” increments the value of the variable by one.

The user symbol table generated along with the object file during a PL/M-86 compilation or by the 8086 Assembler is loaded into the ICE-88 emulator along with the user program which is to be emulated. The user may add to this symbol table any additional symbolic values for memory addresses, constants, or variables that are found to be useful during system debugging.

In addition, the ICE-88 emulator provides access to all the 8088 registers and flags through mnemonic reference. The READY, NMI, TEST, HOLD, RESET, INTR, and MN/MX pins can also be read.

Display

Three basic types of data are available for display: trace data, 8088 termination conditions, and test parameters. Trace data is collected from the 8088 pins during execution of the user program. Trace data collection can be continuous or selective. Tracepoints allow the user to selectively turn trace off and on as desired during emulation. The tracepoints are stored by the ICE-88 hardware on command prior to emulation. If trace data collected exceeds the capacity of the trace buffer, the older trace data is overwritten by current data. Trace buffer pointers entered by the operator permit selection of the trace information for display.

The 8088 termination conditions are the status values of the 8088 processor that are accessible following termination of user program execution. The 8088 termination conditions include the values of registers, flags, input pins, I/O ports, status information, and the contents of the logical user memory space locations currently activated by the memory map. Some of this information is the same as that collected in the trace buffer. All 8088 termination conditions are displayed by console entry of the memory or port address or the name of the register, flag, or input pin.

Hardware resident test parameters are entered by the operator and stored within the ICE-88 emulator hardware. Such information includes breakpoints, tracepoints, the memory map, and the tracepointer used for control of trace data display. The operator displays this information to verify the correct entry or to determine the values of test parameters that were previously entered.
Software resident test parameters are entered by the operator and stored within ICE-88 software. These parameters are used to establish values that affect hardware only upon entry of other commands. For example, the symbol manipulation commands establish the relationship between the object code of the user program and symbols, statement numbers, and module names that are used by the operator to reference the user program code and data symbolically.

Operating Modes

The ICE-88 software is a development system-based program which provides the user with easy-to-use commands for defining breakpoints, initiating emulation, and interrogating and altering user status recorded during emulation. ICE-88 commands are configured with a broad range of modifiers which provide the user with maximum flexibility in describing the operation to be performed.

There are two distinct phases of operation when the ICE-88 emulator is used for debugging. The interval when the user program is being executed is referred to as the emulation phase. The interval when the operator establishes and modifies test parameters and displays (or prints) test results is the interrogation phase.

Emulation

Emulation commands to the ICE-88 emulator control the process of setting up and running an emulation of the user's program and examining the results of the emulation. Breakpoints and tracepoints enable the ICE-88 emulator to halt and provide a detailed trace in any part of the user's program. A summary of the emulation commands is shown in table 1-1.

Breakpoints. The ICE-88 emulator has two breakpoint registers which allow the user to halt emulation when a specified condition is met. The breakpoint registers may be set up as execution or non-execution breakpoints. An execution breakpoint consists of a single address which causes a break whenever the 8088 executes an instruction byte which was obtained from that address. A non-execution breakpoint causes an emulation break when a specified condition other than an instruction execution occurs. This condition can contain up to four parts:

1. A set of address values,
2. A particular status of the 8088 bus (one or more of memory or I/O read or write, instruction fetch, halt, or interrupt acknowledge),
3. A set of data values,
4. A segment register (break occurs when the register is used in an effective address calculation).

Break on a set of address values has three capabilities:
1. To break on a single address.
2. To set any number of breakpoints within a limited range (1024 bytes) of memory.
3. To break in an unlimited range. Execution is halted on any memory access to an address greater than or equal to (or less than or equal to) the breakpoint address.

An external breakpoint match output for user access is provided on the buffer, which allows synchronization of other test equipment when a break occurs.
Tracepoints. The ICE-88 emulator has two tracepoint registers which establish match conditions to conditionally start and stop trace collection. The trace information is gathered at least twice per bus cycle, first when the address signals are valid and second when the data signals are valid. Trace information is also collected each CPU cycle during which the execution queue is active.

Each trace frame contains the 20 address/data line values and detailed information on the status of the 8088. The trace memory can store up to 1023 frames, or an average of about 300 bus cycles, of trace data. The trace memory contains the last 1023 frames of trace data collected, even if this spans several separate emulations. The user has the option of displaying each frame of the trace data or displaying by instruction in actual 8086 Assembler mnemonics. The trace data can be made available after an emulation.

Table 1-1. Summary of Emulation Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GO</td>
<td>Initializes emulation and allows the user to specify the starting point and breakpoints. Example: GO FROM .START TILL. DELAY EXECUTED where START and DELAY are statement labels.</td>
</tr>
<tr>
<td>STEP</td>
<td>Allows the user to single-step through the program.</td>
</tr>
<tr>
<td>GR</td>
<td>Sets the GO-register to a set of one or more breakpoint conditions or causes the display of the current GO-register settings.</td>
</tr>
<tr>
<td>ENABLE/DISABLE TRACE</td>
<td>Turn trace data collection on or off.</td>
</tr>
<tr>
<td>TRACE</td>
<td>Set trace display mode to display trace data in frame or instruction format or display current trace display mode.</td>
</tr>
<tr>
<td>OLDEST</td>
<td>Move trace buffer pointer to top of trace buffer.</td>
</tr>
<tr>
<td>NEWEST</td>
<td>Move trace buffer pointer to bottom of trace buffer.</td>
</tr>
<tr>
<td>MOVE</td>
<td>Move trace buffer pointer forward or backwards in buffer a specified number or buffer entries.</td>
</tr>
<tr>
<td>PRINT</td>
<td>Display one or more entries from the trace data buffer.</td>
</tr>
<tr>
<td>CLOCK</td>
<td>Specify system clock as internal (8088-provided) or external (user-provided) or cause current clock setting to be displayed.</td>
</tr>
<tr>
<td>RWTIMEOUT</td>
<td>Allows the user to time out READ/WRITE command signals based on the time taken by the 8088 to access expansion Intellec memory or disk-based memory.</td>
</tr>
<tr>
<td>ENABLE/DISABLE RDY</td>
<td>Allows the user to enable or disable the user ready signal for accessing Intellec resident memory or disk memory.</td>
</tr>
</tbody>
</table>

Interrogation and Utility

Interrogation and utility commands give the user convenient access to detailed information about the program and the state of the 8088 which is useful in
debugging hardware and software. Changes can be made in both user program memory and the state of the 8088. Commands are also provided for various utility operations such as loading and saving program files, defining symbols and macros, setting up the memory map, and returning control to ISIS-II. A summary of the basic interrogation and utility commands is shown in table 1-2.

Table 1-2. Summary of Basic Interrogation and Utility Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory/Register Commands</td>
<td>Display or change the contents of:</td>
</tr>
<tr>
<td></td>
<td>• Memory</td>
</tr>
<tr>
<td></td>
<td>• 8088 Registers</td>
</tr>
<tr>
<td></td>
<td>• ICE-88 Pseudo-Registers</td>
</tr>
<tr>
<td></td>
<td>• 8088 Status flags</td>
</tr>
<tr>
<td></td>
<td>• 8088 Input pins</td>
</tr>
<tr>
<td></td>
<td>• 8088 I/O ports</td>
</tr>
<tr>
<td>Memory Mapping Commands</td>
<td>Display, declare, set, or reset the ICE-88 memory mapping.</td>
</tr>
<tr>
<td>ASM</td>
<td>Disassembles the memory into 8086 assembler mnemonics.</td>
</tr>
<tr>
<td>LOAD</td>
<td>Fetches user symbol table and object code from the input file.</td>
</tr>
<tr>
<td>SAVE</td>
<td>Sends user symbol table and object code to the output file.</td>
</tr>
<tr>
<td>LIST</td>
<td>Sends a copy of all output (including prompts, input line echos, and error messages) to the chosen output device (e.g., disk, printer) as well as the console.</td>
</tr>
<tr>
<td>EVALUATE</td>
<td>Displays the value of an expression in binary, octal, decimal, hexadecimal, and ASCII.</td>
</tr>
<tr>
<td>Symbol Manipulation</td>
<td>These commands allow the user to:</td>
</tr>
<tr>
<td>Commands</td>
<td>Display any or all symbols, program modules, and program line numbers and their associated values (locations in memory).</td>
</tr>
<tr>
<td></td>
<td>Set the domain (choose the particular program module) for the line numbers.</td>
</tr>
<tr>
<td></td>
<td>Define new symbols as they are needed in debugging.</td>
</tr>
<tr>
<td></td>
<td>Remove any or all symbols, modules, and program statements.</td>
</tr>
<tr>
<td></td>
<td>Change the value or type of any symbol.</td>
</tr>
<tr>
<td>TYPE</td>
<td>Assigns or changes the type of any symbol in the symbol table.</td>
</tr>
<tr>
<td>SUFIX/BASE</td>
<td>Establishes the default base for numeric values in input text/output display (binary, octal, decimal, or hexadecimal).</td>
</tr>
</tbody>
</table>
Macro and Compound Commands

The ICE-88 emulator allows the user to program the operation by using macros and compound commands.

A macro consists of a set of ICE-88 commands with up to ten command parameters and is typically used to perform any task that is required frequently. Commands are provided to define, display, and delete macros, to invoke macros with an optional list of arguments, and to save macros in a diskette file or to load previously created macros from a diskette file.

As an example, the following macro may be used to emulate a user program from a start address until a breakpoint is encountered, then to continue until a condition is satisfied:

```
DEFINE MACRO GO
    IP = OFFSET %0 :DISPLACEMENT OF START ADDRESS
    CS = SEGMENT %0 :BASE OF START ADDRESS
    REPEAT
        GO TILL %1 :EMULATE TO BREAKPOINT
        DISPLAY
        UNTIL %2 :INVOKED MACRO TO DISPLAY VARIABLES OF INTEREST
    ENDR
    EM

:GO .START, #20 EXECUTED OR .A LEN 10T READ, !FLAG = 0
```

The last line invokes macro GO, causing emulation to begin at label START, to break whenever statement #20 is executed or any element of a 10-byte array A is read, and then to continue unless the variable FLAG has a value of zero.

Compound commands are control structures to either conditionally execute other commands (IF), or to execute other commands until some condition is met or the commands have been executed a certain number of times (COUNT, REPEAT).

For example, the following compound command is used to repeat a set of commands until a condition is met:

```
IP = OFFSET .START :DISPLACEMENT OF START ADDRESS
CS = SEGMENT .START :BASE OF START ADDRESS
REPEAT
    UNTIL IP = 1000H :BREAK CONDITION
    STEP :SINGLE STEP
ENDR
```

In this command the condition IP = 1000H is tested every STEP. If the sequence of STEPs reaches IP = 1000H, the loop will terminate.

ICE-88 Architecture

This section contains a brief description of the software, firmware and hardware that compose the ICE-88 emulator. The information serves as an introduction to more detailed information presented in the remaining chapters of this manual.
Software

The ICE-88 software together with ISIS-II and the user program symbol table is resident within the 64K byte memory of the Intellec system. None of this space is available to user program code. User program address space mapped to the Intellec system resides in RAM boards (i.e., extended Intellec memory) whose physical addresses are above the reserved 64K byte address range.

The functions performed by the software are dependent on the ICE-88 emulator operating mode. In the interrogation mode, the software provides arithmetic and logical conversions as necessary to establish compatibility between the ICE-88 hardware and the operator. This task includes conversion of operator commands to a form usable by the firmware and the evaluation of symbolic entries as necessary to provide absolute address and data values to the hardware. The software also reconverts hardware supplied information (trace data, error codes, map data, etc.) to forms that are meaningful to the operator. In the emulation mode, the software supports the accessing of user code from the diskette. In this mode the software also terminates emulation when directed by the hardware (breakpoint) or the operator (ESCape key).

Firmware commands are hardware related commands that are sent to the ICE-88 firmware to initiate a specific action. In general, each ICE-88 (operator-entered) command is an element of higher level language that is converted to a specific series of lower level firmware commands (assuming that the command requires a hardware action). Thus, while the LOAD command merely specifies loading of a user program into user address space, the actual process requires reading of the memory map and writing of the user code into user, ICE memory, Intellec memory, or diskette memory as indicated by the map. Not only are multiple firmware commands required, but the set of firmware commands issued is dependent on the parameters included within the command.

ICE-88 Firmware

The ICE-88 firmware consists of a 12K-byte ROM-resident program that is executed by an 8080 "ICE processor" of the ICE-88 hardware. The firmware performs three major functions. During start-up or system reset, the firmware resets all hardware test parameters and performs a series of go/no-go tests to ensure proper operations of the hardware. In the interrogation mode, the firmware decodes the firmware commands and initiates the specified hardware operations, including the sequencing of data transfers to and from the software. In the emulation mode, the firmware supports user program activities that require use of Intellec resources such as the transfer of user code from diskette or extended Intellec memory.

Hardware

The ICE-88 hardware consists of five circuit boards, a buffer box assembly, and four cables. Three of the circuit boards plug into the Intellec chassis:

- FM Controller Board
- 88 Controller Board
- Trace Board

Two smaller circuit boards are housed within the buffer box assembly:

- Buffer Board 1
- Buffer Board 2
The buffer box cable assembly interconnects the user hardware and the ICE-88 emulator circuit boards within the Intellec chassis. Connection to the user system is made by this cable via the 40-pin socket that normally contains the 8088 user processor. When the ICE-88 emulator is thus connected to the user system, the functions of the user processor are assumed by an 8088 located within the buffer box assembly. The 8088 in the buffer box assembly is called the user processor within this manual. The buffer box assembly is located near the user end of this cable assembly.

X and Y cables interconnect the buffer box and two circuit boards in the Intellec chassis. The T cable provides direct connection between the 88 Controller Board and the Trace Board.

A block diagram of the ICE-88 hardware is shown in figure 1-2.

**Generalized Development Cycle with ICE-88**

Figure 1-3 diagrams a generalized product development cycle using the ICE-88 emulator as a design aid. The sequence of events in developing a new product using the Intellec system with the ICE-88 emulator is approximately as follows.

- Complete the specifications for the prototype hardware design, software control logic, and integrated system performance.
- Organize both the hardware and software designs into logical blocks that are readily understandable, have well-defined inputs and outputs, and are easy to test. Breaking down the design is an interactive process, but is extremely valuable in reducing the time required for prototyping, programming, testing, and modification.

---

**Figure 1-2. ICE-88™ Functional Block Diagram**
• Program the software modules in PL/M-86 and/or in ASM-86 assembly language, naming and storing the programmed modules as files under ISIS-II. Compile or assemble the modules, linking and loading the combinations you are ready to test, creating an object-code (machine language) version. Desk-check each module as it is completed.

• As software modules are ready for testing, load them into ICE, Intellec system or diskette and emulate them via the ICE processor, using the ICE-88 emulator in the 'software' mode. The ICE-88 system allows you to use ICE-supplied memory as part of the 'prototype' system. The advantages of this feature to software development include:

1. You do not have to be concerned about overflowing your prototype system memory in the initial stages of software development. You have the freedom to test the program and compact it later without having to make room for extra memory in your prototype.

2. You may test your program in RAM memory, and make patches quickly and easily without having to erase and reprogram PROM memory. In later test phases, the ICE module can control program execution from PROM or ROM in your prototype. The ICE module can map RAM memory to ICE-supplied memory to replace prototype memory in set increments to test out software changes before reprogramming.

• As software modules pass initial stages of check-out, they can be loaded in the 2K of ICE-88 memory for emulation and testing in 'real-time.'

• Hardware prototyping can begin with just a 8088 CPU socket. Through ICE-88 emulator mapping capabilities, ICE-supplied equivalents can be substituted for missing prototype hardware. As each module of the user's hardware becomes available, it can be added to the prototype, replacing the ICE-supplied equivalent. In this way, modules of software and hardware can be system-tested as they become available.

• You can use memory in ICE-supplied systems to check the interaction of prototype hardware and proven software. The ability to map memory is helpful in isolating system problems. You can exercise all prototype memory from a program residing initially ICE-supplied memory, and reassign memory block-by-block to the user's system as code is verified. Hardware failures can then be isolated quickly, because interactions between prototype parts occur only at your command. You do not have to use the prototype to debug itself.

• The debugging/testing process can proceed through each hardware and software module, using ICE-88 commands to control execution and check that each module gets data or control information from the correct locations, and places correct data or other signals in the proper cells or output locations for subsequent modules to use.

---

Figure 1-3. Typical Development Cycle with ICE™ Module
• Eventually, you test all hardware and software together. The program can reside in RAM or PROM in your system, or in RAM in Intellec systems. All other hardware can be in the prototype. The ICE-88 emulator connected to the system through the microprocessor socket can emulate, test, and trace all the operations of the system.

• After the prototype has been completely tested, the ICE-88 emulator can be used to verify the product in production test. The test procedures you developed for the final prototype testing can serve as the basis for production test routines, running the program from metal-masked ROM in the production system.

A Generalized Emulation Session

This section describes the main steps in an emulation session. You may not always perform all the procedures given here in every emulation session, but the main outline is the same in all sessions. The discussion emphasizes some of the features of the ICE-88 emulator that have not been presented earlier. For the details of the command language, see Chapters 4 through 8.

1. Install the ICE-88 hardware in the Intellec chassis (see Chapter 2).

2. If you are using any prototype hardware, remove socket protector and attach the cable that connects the user hardware to the ICE-88 circuit boards to the prototype via the 40-pin socket. Otherwise leave socket protector attached to the cable.

3. Boot the system, and obtain the hyphen prompt from the ISIS-II system. Enter the ICE-88 command, and obtain the asterisk prompt from the ICE-88 emulator.

4. From the software to be tested, determine how many memory addresses in the Intellec system are required to perform the current emulation. For example, if your program presently uses 3K of memory but your prototype has only 1K installed, you need to “borrow” 2K of memory from the ICE-88 emulator.

ICE-88 system memory is available from three sources: 2K of “real-time” ICE memory, extended Intellec RAM memory, and diskette memory. This memory is available for user program mapping and is organized in blocks of 1K (1024) bytes of contiguous memory. 1024 such blocks are logically available; the amount that is physically available depends upon what you have installed in the Intellec system.

The ICE memory provides you with 2K of RAM memory that enables you to run object code at approximately real-time speed.

Intellec memory is capable of providing 960 1K blocks of logical address space. The Intellec system software occupies the lowest 64K of Intellec RAM memory. Therefore, any Intellec memory available to the user programs must be mapped to addresses above 64K (extended Intellec memory). The amount of Intellec memory physically available is dependent upon the number of card slots available in the Intellec system and the memory physically installed. (Do not use 016 memory boards.) If diskette memory is used, the full range of 1024 blocks of logical memory is available to the user program up to the size of the diskette.

Typically, your program occupies logical locations in low memory. If you intend to use Intellec memory for this emulation, you must map the memory space used by your program into extended Intellec memory. The ICE-88 emulator stores the mapping in its memory map, and refers each memory reference in your program to the proper physical location in Intellec memory.
For example, suppose your code requires absolute addresses 0000H to 0FFFFH (the H means hexadecimal radix), or 4096 contiguous locations beginning at location 0; the lowest address in memory. To map these addresses into the beginning of extended Intellec memory, the mapping command would be:

MAP INTELLEC = 64 LENGTH 62

This command declares that 62K of RAM memory is physically available in extended Intellec memory starting at the lower boundary of extended memory.

MAP 0 LENGTH 4 = INTELLEC

This command maps the logical memory required by your program to the address space in lower Intellec extended memory.

5. Load your program from diskette into the memory locations you have mapped, using the LOAD command.

6. The ICE-88 emulator has three modes of operation: interrogation, continuous emulation, and single-step emulation. The asterisk prompt signals that the ICE-88 emulator is in the interrogation mode, ready to accept any command.

7. In the interrogation mode, prepare the system for emulation by defining symbols and setting emulation breakpoints and tracepoints.

ICE-88 software provides keywords for all 8088 registers and flags. In addition, you may use symbols to refer to memory locations and contents. The user symbol table is generated along with the object file during PL/M compilation or ASM assembly. This table can be loaded into Intellec memory when the user program is loaded.

You are encouraged to add to this symbol table any additional symbolic values for memory addresses, constants, or variables that you may find useful during system debugging. Symbols may be substituted for numeric values in any of the ICE-88 commands.

Symbolic reference is a great advantage to the designer. You do not need to recall or look up the addresses of key locations in your program, as they change with each assembly; you can use meaningful symbols from your source program instead. This facility is especially valuable for high-level language debugging. You can completely debug a program written in PL/M by referencing symbols defined in the source code. You do not need to become involved with the machine level code generated by the compiler. For example, the command:

GO FROM .START TILL .RSL T WRITTEN

begins real-time emulation of the program at the address referenced by the label START in the designer’s PL/M-86 program. The command also specifies that the program is to break emulation when the 8088 microprocessor writes to the memory location referenced by RSL T. You do not have to be concerned with the physical locations of START and RSL T. The ICE-88 software supplies them automatically from information stored in the symbol table.

8. Enter a GO command to begin real-time emulation. The ICE-88 emulator uses a pseudo-register called the GO-register to contain the halting conditions that you have specified, either in the GO command or previously.

9. When emulation halts, you display the trace data collected during that emulation. The ICE-88 emulator loads trace data into a trace buffer. Using ICE-88 commands, you can position the trace buffer pointer to the information that you desire to review, and display one, several, or all the entries in the buffer. You can set the display mode to one frame per line or one instruction per line of display.
10. To control emulation more precisely and to obtain more detailed trace data than with continuous emulation, you can command the ICE-88 emulator to execute single-step emulation. After each step emulated, you can display the current entry in the trace buffer and the current settings of the 8088 registers and pins.

11. You can examine and change memory locations, 8088 registers and flags, and I/O ports, to provide you with valuable information on program operation. You may alter data or register values to observe their effect on the next emulation, or you can patch in changes to your program code itself. You can display and change symbolic values in the symbol table and breakpoint and tracepoint values.

12. Alternate between interrogation and emulation until you have checked everything you want to check.

13. At the end of the emulation session, you can save your debugged code on an ISIS-II diskette file, using the ICE-88 SAVE command. The operation can be specified to save program code, symbol tables, and (for PL/M programs) the source code line number table.

You can start another session immediately, resetting all parameters to their initial values with a few simple commands, or you can exit to ISIS-II to terminate the session.

This introduction is intended to show you some of the scope and power of the ICE-88 emulator in operation, and to suggest how this integrated software/hardware design aid can fit into your development cycle. Chapter 2 contains installation instructions. Chapter 3 contains a hands-on tutorial involving a sample program to be debugged. Chapter 4 describes the meta-notation used in this manual to specify command syntax and semantics. Chapter 5 presents a detailed description of expressions used in this manual. The remaining chapters present the details of the command language in a format and sequence designed for reference.
This chapter contains information on the installation of the ICE-88 emulator.

**ICE-88 Components**

The following items are included in the ICE-88 package:

- **FM Controller board (PN 1002609):** A circuit board that plugs into the Intellec chassis. The FM Controller contains the 8080 ICE processor, 12K-byte firmware ROM, and 3K-byte scratchpad RAM.
- **88 Controller board (PN 1002585):** A circuit board that plugs into the Intellec chassis. The 88 Controller contains the 2K-byte ICE RAM, 1K by 6-bit MAP memory, and 512 byte 2-Port memory.
- **Trace board (PN 1001849):** A circuit board that plugs into the Intellec chassis. The Trace board contains RAM for trace data, tracepoints, and breakpoints.
- **ICE-88 Buffer Box Assembly (PN 4002604):** A cable assembly that contains the ICE-88 Buffer Box Assembly. The Buffer Box contains two small circuit boards that contain the 8088 user processor and gating and control logic for communications with the user system, MAP RAM, ICE RAM, 2-Port RAM, and Trace RAM. The cable assembly also contains the user cable that plugs into the 40-pin receptacle that normally houses the user’s 8088, the X cable that attaches to the 88 Controller board, and the Y cable that attaches to the FM Controller board.
- **Intellec Microcomputer Development System 800 Triple Auxiliary Connector (PN 1001854 or 1001855) and Intellec Series II Triple Auxiliary Connector (PN 1001858):** Each connector consists of a set of three parallel circuit board connectors that provide electrical interconnection between the FM Controller, Trace board and the 88 Controller when they are installed in the Intellec chassis.
- **Ground Cable (PN 4000481):** A cable that provides signal ground to the ICE-88 Buffer Box Assembly from the user system.
- **Required software files on the ICE-88 diskette:**
  - ICE88
  - ICE88.OV0
  - ICE88.OV1
  - ICE88.OV2
  - ICE88.OV3
  - ICE88.OV4
  - ICE88.OV5
  - ICE88.OV6
  - ICE88.OV7
  - ICE88.OV8
  - ICE88.OVE

**Required and Optional Hardware**

ICE-88 emulators require one of the following hardware configurations:

- **Intellec model 800 with:**
  - CRT
  - MDS-2DS or DDS
  - 64K of RAM
  - 3 adjacent card slots available on the motherboard
- **Intellec model 888 with 64K of RAM**
- **Intellec Series II, model 220 or 230 with:**
  - 3 adjacent card slots available in the expansion chassis and 64K of RAM
The following are optional enhancements to an ICE-88 system:

- Serial or parallel printer for hard-copy output
- One or more boards of Intellec expansion memory. If Intellec expansion memory is to be used for emulating 8088 program memory, additional card slots are needed for iSBC-32 or iSBC-64 memory boards. If Intellec expansion memory is used, it is recommended that all Intellec memory consist of iSBC-32 and/or iSBC-64 memory boards. iSBC-16 memory boards contain only 16 bits of address. Therefore, if any iSBC-16 boards are present when expansion memory is being used, each 16K RAM board will be duplicated on each 64K page of addressable memory making these duplicated areas unusable for program storage.

**NOTE**

The Monitor in the Intellec model 800 and 888 occupies the upper 2K of the first 64K of Intellec memory. This address space will be duplicated on each 64K page of Intellec expansion memory used and therefore is unusable for user program storage.

**Hardware Installation Procedures**

The installation procedures of the ICE-88 emulator are presented in the next two sections as follows: the procedure for Intellec model 800 and 888; and the procedure for Intellec Series II, model 220 and 230.

**Installation Procedure for Intellec Model 800 and 888**

1. Disconnect the power cords of the Intellec system and the user system.
3. Inspect the ICE-88 components for damage.
4. Remove the top cover of the Intellec chassis.
5. Mount shorting plug P1 to terminal E5-E6, setting device code to 2. (When shipped, P1 is mounted to E5-E6.) Mount shorting plug P2 to terminal E7-E8, to select -10V as -5V power source. (When shipped, P2 is mounted to E7-E8.) Mount and align the lowest position of the MDS Triple Auxiliary Connector (PN 1001854 or 1001855) on the FM Controller. Then insert them into an odd-numbered card slot.
6. Mount the Trace board into the middle slot of the Triple Auxiliary Connector.
7. Mount the 88 Controller board into the highest numbered slot of the Triple Auxiliary Connector.
8. MOUNT THE T CABLE ASSEMBLY CONNECTING THE ICE TRACE BOARD TO THE 88 CONTROLLER BOARD, MATING THE MISSING PIN ON THE TERMINAL WITH THE BLOCKED INSERT ON THE RECEPTACLE.
9. Expand the Intellec memory as required for user software.
10. ATTACH THE RIBBON CABLE MARKED X TO THE CABLE RECEPTACLE ON THE 88 CONTROLLER BOARD MARKED X MATING THE MISSING PIN ON THE TERMINAL WITH THE BLOCKED INSERT ON THE RECEPTACLE.
11. ATTACH THE RIBBON CABLE MARKED Y TO THE CABLE RECEPTACLE ON THE FM CONTROLLER BOARD MARKED Y MATING THE MISSING PIN ON THE TERMINAL WITH THE BLOCKED INSERT ON THE RECEPTACLE.
12. If a user prototype is to be connected, remove the Socket Protector Assembly from the user end of the ICE-88 Buffer Box Assembly and insert the 40-pin cable terminal into the 8088 socket on the user system. The Socket Protector Assembly guards the terminal pins from damage and inadvertent grounding.

13. Mount the male plug of the Ground Connector into the female receptacle of the 68-136 Terminal Pin at the user end of the cable assembly.

14. Mount the clip end of the Ground Connector to an appropriate point in the user system to provide signal ground.

15. Replace the top cover of the Intellec chassis.

16. Insert the power cords of the Intellec system and the user system into their sockets. Connect both to power sources.

**Installation Procedure for Intellec Series II Model 220 and 230**

1. Disconnect the power cords of the Intellec chassis and user system.

2. Install the Intellec peripherals (diskette drives, CRT, printer), following the instruction guidelines given in the Intellec Series II Installation and Service Manual.

3. Inspect the ICE-88 components for damage.

4. Remove the front cover of the expansion chassis to be used to house the ICE-88 circuit boards.

5. Mount shorting plug P1 to terminal E5-E6, setting device code to 2. (When shipped, P1 is mounted to E5-E6.) Mount shorting plug P2 to terminal E7-E8 to select −10V as −5V power source. (When shipped, P2 is mounted to E7-E8.) Mount and align the lowest position of the EMDS Triple Auxiliary Connector (PN 1001858) on the FM Controller. Then insert them into a card slot.

6. Mount the ICE-88 Trace board into the middle slot of the Triple Auxiliary Connector.

7. Mount the 88 Controller board into the highest numbered slot of the Triple Auxiliary Connector.

8. MOUNT THE ICE-88 T CABLE ASSEMBLY CONNECTING THE ICE TRACE BOARD TO THE 88 CONTROLLER BOARD, MATING THE MISSING PIN ON THE TERMINAL WITH THE BLOCKED INSERT ON THE RECEPTACLE.

9. Expand the Intellec memory as required for user software.

10. ATTACH THE RIBBON CABLE MARKED X TO THE CABLE RECEPTACLE ON THE 88 CONTROLLER BOARD MARKED X MATING THE MISSING PIN ON THE TERMINAL WITH THE BLOCKED INSERT ON THE RECEPTACLE.

11. ATTACH THE RIBBON CABLE MARKED Y TO THE CABLE RECEPTACLE ON THE FM CONTROLLER BOARD MARKED Y MATING THE MISSING PIN ON THE TERMINAL WITH THE BLOCKED INSERT ON THE RECEPTACLE.

12. If a user prototype is to be connected, remove the Socket Protector Assembly from the user end of the ICE-88 Buffer Box Assembly and insert the 40-pin cable terminal into the 8088 socket on the user system. The Socket Protector Assembly guards the terminal pins from damage and inadvertent grounding.

13. Mount the male plug of the Ground Connector into the female receptacle of the 68-136 Terminal Pin at the user end of the cable assembly.

14. Mount the clip end of the Ground Connector to an appropriate point in the user system to provide signal ground.
15. Replace the front cover of the expansion chassis.
16. Insert the power cords of the Intellec system and the user system into their sockets. Connect both to power sources.

The distance between the two connectors of the T ribbon cable can be adjusted by bending the cable to match the spacing between the Trace board and the 88 Controller board.

NOTE
Keep Socket Protector Assembly mounted on the end of the ICE Buffer Box Assembly whenever the terminal is not attached to a user system. This will prevent inadvertant pin damage.

When removing the Socket Protector Assembly from the end of the ICE Buffer Box Assembly, use care to prevent pin damage.

Confidence Testing

At the conclusion of the installation procedures for the Intellec Model 800 or 888, or the Intellec Series II, execute the DIAG88 confidence program. The DIAG88 program resets and invokes the ICE-88 emulator, executes a set of hardware confidence tests, and terminates by returning a set of “PASS” or “FAIL” display messages. Execute the following sequence to run the confidence tests:

• Boot the system to run under ISIS-II and wait for the hyphen prompt from the ISIS-II system.
• Enter the command CONF and wait for the prompt, *.
• Enter the command INIT DIAG88.CON and wait for the prompt, *.
• Enter the command TEST to cause DIAG88 to execute the confidence tests. Wait for the test message displays. DIAG88 will display a “PASS/FAIL” message for each diagnostic test contained in DIAG88. If any displayed test message denotes a “FAIL,” the installed hardware is not operating properly. Inspect the hardware for improper installation and rerun DIAG88. If all the displayed test messages denote “PASS,” the hardware has been installed correctly and is operating properly.
• Enter the command EXIT to return control to the ISIS-II system.
This chapter introduces a few useful ICE-88 commands and provides hands-on experience with the ICE-88 emulator. To reduce the need for cross-reference, this chapter includes brief discussions of the commands used in the examples. The user program to be simulated is a simple traffic light controller. The user program logic is described before the hands-on session to help you understand what is going on.

How To Use This Chapter

- To use this program as a hands-on tutorial, you must enter the source code for the two modules using the ISIS text editor on your system. Omit the line number and nesting information that is on the listing; these values are assigned by the compiler and assembler.
- Compile the CARS module with the PL/M-86 compiler program. Assuming that the source file is named CARS.SRC, the compile step could look like:
  ```
  -PLM86:F1:CARS.SRC PRINT(:F1:CARS.PRT) DEBUG
  ```
  The object file created by PL/M-86 is named CARS.OBJ. (The DEBUG control generates the symbol table for use by the ICE-88 emulator.)
- Assemble the DELAY module with the ASM86 assembler. Assuming that the source file is named DELAY.SRC, the assemble step could look like:
  ```
  -ASM86:F1:DELAY.SRC PRINT(:F1:DELAY.PRT) OBJECT(:F1:DELAY.OBJ) DEBUG
  ```
  As indicated, the object module is named DELAY.OBJ.
- Link and Locate CARS and DELAY using the MCS-86 utility QRL86. The command we used looks like:
  ```
  -QRL86:F1:CARS.OBJ, :F1:DELAY.OBJ TO :F1:CARS & **ORIGIN,(0) PRINT(:F1:CARS.QRL) MAP
  ```
  The ampersand (&) is entered before the carriage return to request a continuation line. The beginning of the continuation line is indicated by a double asterisk prompt (**).
  The executable file is now named CARS; program addresses start at zero (0) because of the ORIGIN control in the QRL86 invocation.
- For further information on the procedures for editing, compiling, assembling, linking, and locating programs for the MCS-86 Microprocessor, refer to the following manuals:
  - Text Editor: *ISIS-II System User’s Guide*
  - PL/M-86 Compiler: *PL/M-86 Programming Manual*  
    *PL/M-86 Operator’s Manual*
  - ASM-86 Assembler: *MCS-86™ Assembly Language Reference Manual*  
    *MCS-86™ Assembler Operating Instructions for IS7-86 Users*
  - QRL86: *MCS-86™ Software Development Utilities Operating Instructions for IS7-II Users.*
- Study the logic of CARS, the program to be emulated. The material includes text discussion and program listings.
- Install the ICE-88 hardware, following the procedure given in Chapter 2. Leave the socket protector on to protect the pins at the end of the cable.
- Insert an ISIS-II system diskette in drive 0 and boot ISIS.
• Copy CARS to the diskette containing the ICE-88 program. Insert this
diskette in drive 1.

• Enter the command

   \texttt{:F1:ICE88}

to load the ICE software and start it executing. The ICE-88 emulator signals
readiness to accept commands by displaying an asterisk prompt (*).

• To obtain a diskette copy of the session, enter the command:

   \texttt{LIST :F1:OUTPUT.LOG}

   Of course, you can give your list file any name you desire instead of OUTPUT.LOG.

• Enter the commands as shown, and obtain the results shown in the listing.

\section*{Analysis of the Sample Program}

The application presented is a simple traffic light controller. Imagine an intersec-
tion of a main street and a side street. The desired operation is that the light
should stay green on the main street until a decision involving the number of cars
waiting on the side street and the amount of time they have been waiting has
been satisfied. We suppose that there is a sensor in the pavement on the side
street that sends an interrupt to the computer when a car arrives. We do not in-
clude the control of a yellow light on either street.

Refer to the following figures:

\begin{figure}
\centering
\includegraphics[width=\textwidth]{cars_module}
\caption{CARS Module Listing}
\end{figure}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{delay_module}
\caption{Delay Module Listing}
\end{figure}

Associated with each street is a time called the cycle length. In the program, the
variable named \texttt{SIDE\$CYCLE\$LENGTH} controls the fixed length of time the
light is green on the side street when that cycle is called into action. Even though
the light stays green on the main street until the decision rule is satisfied, we need
a variable \texttt{MAIN\$CYCLE\$LENGTH} that is involved in the decision rule.

The decision rule is as follows. The side street gets a green light if either of the
following two conditions is satisfied.

1. Two or more cars are waiting on the side street, and the main street has had
   the green light for a period of time greater than or equal to the variable
   \texttt{MAIN\$CYCLE\$LENGTH}.

2. One car is waiting on the side street, and the main street has had the green
   light for a period of time equal to or greater than two times the variable
   \texttt{MAIN\$CYCLE\$LENGTH}.

The system has one input and one output. The input is a signal that a car has ar-
rived on the side street since the last time we sampled the input. The variable
\texttt{CARS\$WAITING} contains the number of cars waiting on the side street. The
output goes to the traffic light controller. We assume that sending the controller
a 1 makes the light on the main street green and the light on the side street red;
sending it a 0 makes the light on the main street red and the light on the side
street green. The variable \texttt{LIGHT\$STATUS} represents this output.
/* TRAFFIC LIGHT CONTROLLER PROGRAM */

CARS:
DO;
DECLARE (MAIN$TIME, SIDE$TIME) BYTE;
DECLARE MAIN$CYCLE$LENGTH BYTE DATA(8), SIDE$CYCLE$LENGTH BYTE DATA(5);
DECLARE CARS$WAITING BYTE;
DECLARE LIGHT$STATUS BYTE;
DECLARE FOREVER LITERALLY 'WHILE I';

SIDE$STREET$CAR:
PROCEDURE;
CARS$WAITING = CARS$WAITING + 1;
END SIDE$STREET$CAR;

/* FOLLOWING PROCEDURE CODED IN ASSEMBLY LANGUAGE AND LINKED IN */
DELAY:
PROCEDURE (TIME$HUNDREDTHS) EXTERNAL;
DECLARE TIME$HUNDREDTHS BYTE;
END DELAY;

DISPLAY:
PROCEDURE (CYCLE$TIME);
DECLARE CYCLE$TIME BYTE;
LIGHT$STATUS = LIGHT$STATUS;
END DISPLAY;

CYCLE:
PROCEDURE;
LIGHT$STATUS = 0; /* MAIN RED, SIDE GREEN */
SIDE$TIME = 0;
DO WHILE SIDE$TIME <= SIDE$CYCLE$LENGTH;
CALL DISPLAY(SIDE$TIME);
CALL DELAY(100);
SIDE$TIME = SIDE$TIME * 1;
END;
LIGHT$STATUS = 1; /* MAIN GREEN, SIDE RED */
END CYCLE;

/* MAIN PROGRAM — EXECUTION BEGINS HERE */
LIGHT$STATUS = 1; /* START WITH MAIN GREEN */
CARS$WAITING = 0;
MAIN$TIME = 0;
DO FOREVER;
CALL DISPLAY(MAIN$TIME);
CALL DELAY(100);
MAIN$TIME = MAIN$TIME * 1;
IF (CARS$WAITING >= 2) AND (MAIN$TIME >= MAIN$CYCLE$LENGTH)
OR (CARS$WAITING = 1) AND (MAIN$TIME >= 2 * MAIN$CYCLE$LENGTH) THEN
DO;
CALL CYCLE;
CARS$WAITING = 0;
MAIN$TIME = 0;
END;
END CARS;

Figure 3-1. CARS Module Listing
Figure 3-2. DELAY Module Listing
The program is initialized with constants and variables set as follows.

\[
\begin{align*}
\text{MAINECYCLELENGTH} &= 8 \text{ seconds} \\
\text{SIDECELCYCLELENGTH} &= 5 \text{ seconds} \\
\text{MAINTIME} &= 0 \text{ (Time since last change to MAIN GREEN, SIDE RED)} \\
\text{SIDETIME} &= \text{not set yet. (Time since last change to SIDE GREEN)} \\
\text{LIGHTSTATUS} &= 1 \text{ (MAIN GREEN, SIDE RED)} \\
\text{CARS\$WAITING} &= 0
\end{align*}
\]

The CARS program contains a procedure CYCLE to change the lights back and forth. CYCLE holds the side street light green (main red) until its counter, SIDETIME, exceeds the SIDECYCLELENGTH (nominally 5 seconds).

The ICE-88 test suite includes commands that simulate the arrival of cars on the side street, and that display the values of the program variables involved in the light change logic. The procedure SIDESTREETCAR represents the nucleus of the interrupt routine that would handle the sensor interrupts in a real traffic light controller program. The interrupt-enabling logic is omitted for simplicity. Procedure DISPLAY is a "vestige" of a previous version of CARS. CARS also calls DELAY when a "one-second" timer is required.

**Hands-On Demonstration**

This demonstration involves the one program CARS. The version we ran did not have any serious logic errors, so that the effects of the ICE-88 commands could be clearly seen. The length of the delay produced by the DELAY routine is longer than desired; you may adjust the calling parameter if you desire a "true" one-second delay.

The material represents two separate sessions at the terminal. The beginning and end of each session is clearly indicated. By using two sessions we can demonstrate the use of the PUT and INCLUDE commands.

The pair of sessions is organized as follows—session 1 shows how to define and save macros on file; the macros defined in this session are of two kinds: general purpose MCS-86 utilities (PUSH88, POP88, SETIP) and macros that are particular to CARS, the demonstration program.

The demonstration emphasizes the ICE-88 macro facility, showing how four basic ICE operations (initialize, emulate, display, change) can be organized into named blocks—the building blocks of test sequences.

The define macro command has the syntax:

\[
\text{DEFINE MACRO macro-name cr} \\
[\text{command cr}]... \\
\text{EM}
\]

The commands inside a macro definition (including calls to other macros) are not examined or executed by the ICE-88 emulator until the macro is invoked. A macro call has the format: \text{"macro-name"}. More details on commands are given in the following discussion.
Session 1

0 We begin the session by entering :F1:ICE88 to ISIS-II (hyphen prompt), and receive the ICE-88 sign-on message and asterisk prompt. To record the session on diskette file, we enter a LIST command with the drive and filename that is to contain the output of the ICE-88 operations (including error messages if any).

Many of the commands include comments. A comment is preceded by a semicolon (;) to separate it from the command.

The discussion is keyed to the listing by margin numbers.

1 Macro PUSH88 simulates the MCS-86 PUSH instruction. SP is the stack pointer; SS is the base address of the stack segment.

POP88 is the reverse procedure, simulating the MCS-86 POP instruction.

The parameter %0 in both PUSH88 and POP88 lets us “push” or “pop” any register (or expression, etc.) as long as it can be expressed as a WORD-type quantity.

2 Macro SETIP resets the instruction pointer CS:IP to the address (symbol, expression, etc.) passed as a parameter when the macro is invoked. CS is the base of the code segment and IP is the instruction pointer relative to CS. Like PUSH88 and POP88, SETIP is a useful macro for restarting emulation at a desired point (without “softwiring” start addresses into your emulation macros).

3 Macro TYPES demonstrates how to set up to use typed memory references. A symbol that stands for the address of a variable (not a procedure name) can be defined or assigned a memory-type. Examples of memory-types are BYTE, WORD, and POINTER. In our CARS program, all the key variables are of type BYTE. Since the symbols are loaded with the program rather than being DEFINED, we assign types to the variable with the commands of the form:

```
TYPE .symbol-name = memory-type
```

Then, as shown later on (e.g., in macro VARIABLES, step 6 of session 1) we can refer to the contents of any typed variable with a typed memory reference of the form

```
!symbol-name
```

The contents produced by a typed memory reference are automatically of the type assigned or declared.

See Chapter 7 for more details on memory types.
```
0    "F1:ICE88
    "ISIS-II ICE-88 V1.0
    "LIST :F1:DEC10.LOG ;SESSION ONE
    
1    DEFINE MACRO PUSH88
    *SP = SP - 2T ;MOVE POINTER TO NEW TOP OF STACK
    *WORD SS:SP = %0 ;PUSH PARAMETER ON STACK
    *EM ;END OF MACRO PUSH88
    
    DEFINE MACRO POP88
    *%0 = WORD SS:SP ;POP PARAMETER OFF STACK
    *SP = SP + 2T ;MOVE POINTER TO NEW TOP OF STACK
    *EM ;END OF MACRO POP88
    
2    DEFINE MACRO SETIP
    *CS = SEG (%0)
    *IP = OFF (%0)
    *EM ;END OF MACRO SETIP
    
3    DEFINE MACRO TYPES
    *TYPE .MAINTIME = BYTE ;FROM PLM LISTING
    *TYPE .SIDETIME = BYTE ;FROM PLM LISTING
    *TYPE .MAINCYCLELENGTH = BYTE ;FROM PLM LISTING
    *TYPE .SIDECYCLELENGTH = BYTE ;FROM PLM LISTING
    *TYPE .CARSWAITING = BYTE ;FROM PLM LISTING
    *TYPE .LIGHTSTATUS = BYTE ;FROM PLM LISTING
    *EM ;END OF MACRO TYPES
```
4 We define a macro INIT to handle the map and load steps for our program, CARS.

In the macro INIT, the command MAP 0 LENGTH 2 = ICE 0 assigns two memory blocks (1K segments) to ICE memory. The CARS program was LOCATed at ORIGIN 0 (QRL step discussed above) to facilitate mapping to ICE memory.

INIT defines a useful symbol, .START = CS:IP. After LOAD, CS:IP points to the first executable instruction in the user program. CS is the base of the code segment and IP is the instruction pointer (relative to CS).

Then, INIT calls the TYPES macro defined in step 3. This macro will become part of INIT whenever INIT is called. Until INIT is called, however, the call to TYPES is not executed.

Finally, INIT displays the symbol and statement number tables, to verify that the LOAD step has been completed, and that the TYPES macro has executed.

5 Macro EXAM is designed to test the logic that controls the light change. Basically, the macro block is an indefinite REPEAT loop (the block beginning with REPEAT and ending with ENDREPEAT). On each iteration a single step is emulated (one instruction). Following that, we use an IF command to look for certain addresses and take appropriate actions. The action taken in all cases is to display the PL/M statement or an equivalent message using the WRITE command (see step 6 for more on the WRITE command). In addition, we skip both DELAY and DISPLAY by popping the return address and call parameter off the stack.
ICE-88 Sample ICE-88 Sessions at the Terminal

4 *DEFINE MACRO INIT
  • *MAP 0 LENGTH 2 = ICE 0 ; MEMORY SPACE FOR CARS PROGRAM
  • *MAP 0 LENGTH 2 ; DISPLAY WHAT WE MAPPED
  • *LOAD : F1: CARS
  • *DEFINE .START = CS: IP ; HANDY SYMBOL FOR RESTARTING
  • *TYPES ; MACRO FOR TYPE DEFINITIONS
  • *SYMBOLS
  • *LINES ; DISPLAY SYMBOL AND LINE NUMBER TABLES
  • *EM ; END OF MACRO INIT

5 *DEFINE MACRO EXAM
  • *REPEAT
  • *STEP
  • *IF CS = SEG(. DISPLAY) AND IP = OFF(. DISPLAY) THEN
  • *WRITE 'CALL DISPLAY'
  • * : POP88 IP ; RESTORE RETURN ADDRESS
  • *SP = SP + 2T ; DISCARD PARAMETER
  • *ORIF CS = SEG(. DELAY) AND IP = OFF(. DELAY) THEN
  • *WRITE 'CALL DELAY'
  • * : POP88 IP
  • *SP = SP + 2T
  • *ORIF CS = SEG(. CARS#30) AND IP = OFF(. CARS#30) THEN
  • *WRITE 'STARTING MAIN LOOP'
  • *ORIF CS = SEG(. CARS#34) AND IP = OFF(. CARS#34) THEN
  • *WRITE 'START OF IF TEST'
  • * : VARIABLES
  • *ORIF CS = SEG(. CYCLE) AND IP = OFF(. CYCLE) THEN
  • *WRITE 'CALL CYCLE'
  • *ENDIF
  • *ENDREPEAT
  • *EM ; END OF MACRO EXAM
6 Macro VARIABLES employs the WRITE command to display the key program variables with identifiers. The general syntax of WRITE is:

\[
\text{WRITE } \text{element }, \text{element} \ldots
\]

The elements to be "written" (displayed at the console) can be strings (e.g., 'SIDETIME') enclosed in single quotes, expressions, or constants of the form BOOL expression. Two or more elements can be combined by listing them in the order you desire, separated by commas. The strings in our commands serve to label the displays.

VARIABLES also uses typed memory references; they have the format:

\[
!\text{symbol-name}
\]

Note that each of the symbol names in VARIABLES must be assigned a type (by macro TYPES) before LOOK can be called. Then, the typed reference produces the contents of the given address. For example, if symbol .A is the address of a variable of type BYTE, !A means the same thing as BYTE.A; if .A is a WORD-type variable, however, !A means WORD.A.

The display produced by VARIABLES is shown in session 2 below.

7 Macro TEST combines several macros and simple commands in a test suite. First we initialize the system by executing GO FROM START TILL ..CARS#30 EXECUTED. Statement #30 in the beginning of the main loop.

Next, TEST uses a parameter to assign a value to CARSWAITING. When we call TEST, we supply any value we wish as a parameter.

Last, TEST calls EXAM to single step through the program displaying any calls that occur.

8 The DIR MAC command produces a display of the titles of all the macros we defined, in the order they were defined.

9 To display the definition of any macro, use a command with the form:

\[
\text{MACRO } \text{macro-name}
\]

Using this command, we display the definition of macro SETIP.

10 We save our macro definition on a permanent file for use in later sessions.

11 The EXIT command closes all files and returns us to ISIS-II (hyphen prompt).
6  *DEFINE MACRO VARIABLES
   *WRITE 'LIGTHSTATUS = ',!LIGTHSTATUS,', CARSWAITING = ',!CARSWAITING
   *WRITE 'MAINCYCLELENGTH = ',!MAINCYCLELENGTH,', MAINTIME = ',!MAINTIME
   *WRITE 'SIDE CYCLELENGTH = ',!SIDE CYCLELENGTH,', SIDETIME = ',!SIDETIME
   *EM ;END OF MACRO VARIABLES

7  *DEFINE MACRO TEST
   *GO FROM .START TILL ..CARS#30 EXECUTED
   *!CARSWAITING = %0
   *:*:VARIABLES
   *:*:EXAM
   *:*:EM
   *:*:END OF MACRO TEST

8  *DIR MAC
   PUSH88
   POP88
   SETIP
   INIT
   EXAM
   VARIABLES
   TEST

9  *MACRO SETIP
   DEFINE MACRO SETIP
   CS = SEG (%0)
   IP = OFF (%0)
   EM ;END OF MACRO SETIP

10  *PUT :F1:TEST.INC MACRO

11  *EXIT
Session 2

0. We call up the ICE-88 program and specify a LIST file as before.

1. The INCLUDE command causes the ICE-88 emulator to read commands from a file rather than from the console. The form of this command is just:

   INCLUDE :drive: filename

   In our case, the file :F1:TEST.INC contains the macro definitions from session 1. Thus, when we enter the command:

   INCLUDE :F1:TEST.INC

   The ICE-88 emulator reads in all the macro definitions from PUSH88 through TEST, then returns control to the console.
ICE-88 Sample ICE-88 Sessions at the Terminal

0 \> :F1:ICE88
ISIS-II ICE-88 V1.0
* LIST :F1:DEC11.LOG ;SESSION TWO

1 *INCLUDE :F1:TEST.INC
*DEFINE MACRO PUSH88
*SP = SP - 2T ;MOVE POINTER TO NEW TOP OF STACK
*WORD SS:SP = %0 ;PUSH PARAMETER ON STACK
*EM ;END OF MACRO PUSH88
*DEFINE MACRO POP88
*%0 = WORD SS:SP ;POP PARAMETER OFF STACK
*SP = SP + 2T ;MOVE POINTER TO NEW TOP OF STACK
*EM ;END OF MACRO POP88

*DEFINE MACRO SETIP
*CS = SEG (%0)
*IP = OFF (%0)
*EM ;END OF MACRO SETIP

*DEFINE MACRO TYPES
*TYPE .MAINTIME = BYTE ;FROM PLM LISTING
*TYPE .SIDETIME BYTE ;FROM PLM LISTING
*TYPE .MAINCYCLELENGTH = BYTE ;FROM PLM LISTING
*TYPE .SIDECYCLELENGTH = BYTE ;FROM PLM LISTING
*TYPE .CARSWAITING = BYTE ;FROM PLM LISTING
*TYPE .LIGHTSTATUS = BYTE ;FROM PLM LISTING
*EM ;END OF MACRO TYPES

*DEFINE MACRO INIT
*MAP 0 LENGTH 2 = ICE 0 ;MEMORY SPACE FOR CARS PROGRAM
*MAP 0 LENGTH 2 ;DISPLAY WHAT WE MAPPED
*LOAD :F1:CARS
*DEFINE .START = CS:IP ;HANDY SYMBOL FOR RESTARTING
*TYPE ;MACRO FOR TYPE DEFINITIONS
*Symbols
*LINES ;DISPLAY SYMBOL AND LINE NUMBER TABLES
*EM ;END OF MACRO INIT

*DEFINE MACRO EXAM
*REPEAT
*STEP
*IF CS = SEG(.DISPLAY) AND IP = OFF(.DISPLAY) THEN
*WRITE 'CALL DISPLAY'
*:POP88 IP ;RESTORE RETURN ADDRESS
*SP = SP + 2T ;DISCARD PARAMETER
*ORIF CS = SEG(.DELAY) AND IP = OFF(.DELAY) THEN
*WRITE 'CALL DELAY'
*:POP88 IP
*ORIF CS = SEG(.CARS#30) AND IP = OFF(.CARS#30) THEN
*WRITE 'STARTING MAIN LOOP'
*ORIF CS = SEG(.CARS#34) AND IP = OFF(.CARS#34) THEN
*WRITE 'START OF IF TEST'
*:VARIABLES
*ORIF CS = SEG(.CYCLE) AND IP = OFF(.CYCLE) THEN
*WRITE 'CALL CYCLE'
*:ENDIF
*:ENDREPEAT
*EM ;END OF MACRO EXAM

*DEFINE MACRO VARIABLES
*WRITE 'LIGHTSTATUS = ',LIGHTSTATUS, ' CARSWAITING = ',CARSWAITING
*WRITE 'MAINCYCLELENGTH = ',MAINCYCLELENGTH, ' MAINTIME = ',MAINTIME
*WRITE 'SIDECYCLELENGTH = ',SIDECYCLELENGTH, ' SIDETIME = ',SIDETIME
*EM ;END OF MACRO VARIABLES

*DEFINE MACRO TEST
*GO FROM .START TILL ..CARS#30 EXECUTED
*!CARSWAITING = %0
*:VARIABLES
*:EXAM
*EM ;END OF MACRO TEST
2 We invoke macro INIT with a macro call of the form:

```
:macro-name
```

In our example, the call is:

```
:INIT
```

First, the macro is "expanded" to form a block of executable commands. The expansion of INIT involves expanding the macro TYPES, at the point that TYPES is called within INIT (see the definition of INIT on the previous page). As INIT is expanded, each command is checked for syntax; any error here would abort the macro call. However, no errors occur and we reach the EM token marking the end of the macro expansion.

3 The commands in INIT now execute. The MAP commands allocate space in ICE memory for our code and display the resulting map (T indicates decimal radix):

```
0000T=ICE 0001T=ICE 0001T
```

The SYMBOLS command displays the symbol table; the listing shows the symbol names and corresponding addresses. We see that .START is present in an unnamed module at the head of the table, and other symbols are listed in the order they appear within the two program modules, ..CARS and ..DELAY. Note the type specifications on the program variables named in the macro TYPES.

The LINES command displays the statement numbers and corresponding addresses from CARS. DELAY has no line numbers because the assembler does not produce a line number table.
"INIT
•*MAP 0 LENGTH 2 = ICE 0 ;MEMORY SPACE FOR CARS PROGRAM
•*MAP 0 LENGTH 2 ;DISPLAY WHAT WE MAPPED
•*LOAD :F1:CARS
•*DEFINE .START = CS:IP ;HANDY SYMBOL FOR RESTARTING
•*TYPES ;MACRO FOR TYPE DEFINITIONS
•**TYPE .MAINTIME = BYTE ;FROM PLM LISTING
•**TYPE .SIDETIME = BYTE ;FROM PLM LISTING
•**TYPE .MAINCYCLELENGTH = BYTE ;FROM PLM LISTING
•**TYPE .SIDE CYCLE LENGTH BYTE ;FROM PLM LISTING
•**TYPE .CARS WAITING = BYTE ;FROM PLM LISTING
•**TYPE .LIGHT STATUS = BYTE ;FROM PLM LISTING
•**EM ;END OF MACRO TYPES
•*SYMBOLS
•*LINES ;DISPLAY SYMBOL AND LINE NUMBER TABLES
•*EM ;END OF MACRO INIT

0000T=ICE 0000T 0001T=ICE 0001T
•START=0002H
MODULE .CARS
•MEMORY=0120H
•MAINTIME=0106H OF BYT
•SIDETIME=0107H OF BYT
•MAIN CYCLE LENGTH=0104H OF BYT
•SIDE CYCLE LENGTH=0105H OF BYT
•CARS WAITING=0108H OF BYT
•LIGHT STATUS=0109H OF BYT
•SIDE STREET CAR=008EH
•DISPLAY=008EH
•CYCLE=00A7H
MODULE .DELAY
•DELAY=00DEH
•LAB1=00E5H
•LAB2=00E7H
•M=0000H
•MEMORY .=012:0000H
•STACK .BASE=0011:0000H
•TEMP=0101H
MODULE .CARS
#1=008EH
#2=008EH
#3=008EH
#4=008EH
#5=008EH
#6=008EH
#7=008EH
#8=0091H
#9=0096H
#11=009BH
#12=009BH
#13=009BH
#14=009BH
#15=009BH
#16=00A3H
#17=00A4H
#18=00AAH
#19=00AFH
#20=00B4H
#21=00C1H
#22=00CBH
#23=00CEH
#24=00D3H
#25=00D6H
#26=00DBH
#27=0022H
#28=0015H
#29=001AH
#30=001FH
#31=0018H
#32=0026H
#33=002CH
#34=0037H
#35=007CH
4. We now invoke macro TEST to exercise the program logic. We wish to demonstrate that the code will branch to CYCLE when either of the following two conditions is true:

1) \( \text{CARSWAITING} = 1 \) AND \( \text{MAINTIME} \geq 16 \) seconds

2) \( \text{CARSWAITING} \geq 2 \) AND \( \text{MAINTIME} \geq 8 \) seconds

In the definition of TEST (look back at step 1) the second command is:

\[ !\text{CARSWAITING} = \%0 \]

The parameter \( \%0 \) lets us set the contents of CARSWAITING to any BYTE quantity we require. Thus we test condition 1 by our command:

TEST 1

This results in the expansion:

\[ !\text{CARSWAITING} = 1 \]

The expansion of TEST involves the expansion of the macros VARIABLES and EXAM at the point each is called in the body of TEST.

Macro TEST now begins to execute. To help us follow the displays, we can identify some key addresses in the portions of code we are checking, as shown in table 3-1.

### Table 3-1. Key Addresses in CARS Logic

<table>
<thead>
<tr>
<th>ADDRESS (IP)</th>
<th>LINE # NUMBER</th>
<th>PLACE IN CARS CODE</th>
</tr>
</thead>
<tbody>
<tr>
<td>0002H</td>
<td>#27</td>
<td>START OF MAIN PROGRAM (.START)</td>
</tr>
<tr>
<td>001FH</td>
<td>#30, 31</td>
<td>START OF MAIN ‘DO’ BLOCK</td>
</tr>
<tr>
<td>0037H</td>
<td>#34</td>
<td>START OF IF TEST</td>
</tr>
<tr>
<td>007CH</td>
<td>#35, 36</td>
<td>START OF CONDITIONAL ‘DO’ BLOCK</td>
</tr>
<tr>
<td>007FH</td>
<td>#37</td>
<td>POINT OF RETURN FROM ‘CALL CYCLE’</td>
</tr>
<tr>
<td>0089H</td>
<td>#39, 40</td>
<td>END OF BOTH CONDITIONAL AND MAIN ‘DO’ BLOCKS</td>
</tr>
<tr>
<td>0098H</td>
<td>#13</td>
<td>BEGINNING OF DISPLAY (SKIPPED)</td>
</tr>
<tr>
<td>00A7H</td>
<td>#17</td>
<td>BEGINNING OF CYCLE</td>
</tr>
<tr>
<td>00DBH</td>
<td>#26</td>
<td>END OF CYCLE</td>
</tr>
<tr>
<td>00DEH</td>
<td>(#17)</td>
<td>BEGINNING OF DELAY (SKIPPED)</td>
</tr>
</tbody>
</table>
ICE-88

Sample ICE-88 Sessions at the Terminal

#36=007CH
#37=007FH
#38=0084H
#39=0089H
#40=0089H
#41=0OBCH

MODULE .DELAY

.:

4:*TEST 1
 *GO FROM .START TILL .CARS#30 EXECUTED
 *ICARSWAITING = 1
 *:

*VARIABLES
 *WRITE 'LIGHTSTATUS = ',!LIGHTSTATUS, ', CARSWAITING = ',!ICARSWAITING
 *WRITE 'MAINCYCLELENGTH = ',!MAINCYCLELENGTH, ', MAINTIME = ',!MAINTIME
 *WRITE 'SIDE CYCLELENGTH = ',!SIDE CYCLELENGTH, ', SIDETIME = ',!SIDETIME
 *EM
 ;END OF MACRO VARIABLES

*EXAM
 *REPEAT
 *:

*STEP

*IF CS = SEG(.DISPLAY) AND IF = OFF(.DISPLAY) THEN
 *WRITE 'CALL DISPLAY'
 *:

*IP = WORD SS:SP
 ;POP PARAMETER OFF STACK
 *SP = SP + 2T
 ;MOV POINTER TO NEW TOP OF STACK
 *EM
 ;END OF MACRO POP88

*IF CS = SEG(.DISPLAY) AND IF = OFF(.DISPLAY) THEN
 *WRITE 'CALL DELAY'
 *:

*IP = WORD SS:SP
 ;POP PARAMETER OFF STACK
 *SP = SP + 2T
 ;DISCARD PARAMETER
 *EM
 ;END OF MACRO POP88

*ORIF CS = SEG(.CARS#30) AND IF = OFF(.CARS#30) THEN
 *WRITE 'STARTING MAIN LOOP'
 *:

*ORIF CS = SEG(.CARS#34) AND IF = OFF(.CARS#34) THEN
 *WRITE 'START OF IF TEST'
 *:

*VARIABLES
 *WRITE 'LIGHTSTATUS = ',!LIGHTSTATUS, ', CARSWAITING = ',!ICARSWAITING
 *WRITE 'MAINCYCLELENGTH = ',!MAINCYCLELENGTH, ', MAINTIME = ',!MAINTIME
 *WRITE 'SIDE CYCLELENGTH = ',!SIDE CYCLELENGTH, ', SIDETIME = ',!SIDETIME
 *EM
 ;END OF MACRO VARIABLES

*ORIF CS = (.CYCLE) AND IF = OFF(.CYCLE) THEN
 *WRITE 'CALL CYCLE'
 *:

*ENDIF
 *ENDREPEAT

*EM
 ;END OF MACRO EXAM

*EM
 ;END OF MACRO TEST
5 The messages 'EMULATION BEGUN' and 'EMULATION TERMINATED, CS:IP = 0000:0023' are produced by the command 'GO FROM .START TILL ..CARS#30 EXECUTED.'

6 The next three display lines are produced by macro VARIABLES. We see that LIGHTSTATUS is 1 (main street green), and that CARSWAITING has been set to 1 by the command '!CARSWAITING = 1' in TEST. MAINCYCLELENGTH and SIDECYCLELENGTH are constants at 8 and 5 respectively. MAINTIME and SIDETIME both are zero.

7 This is the beginning of the REPEAT loop in macro EXAM. The first STEP ends with address 0098H in the instruction pointer; this is the beginning of DISPLAY and macro EXAM displays the 'CALL DISPLAY' message.

8 When the beginning of DELAY appears in CS:IP, EXAM displays 'CALL DELAY'.

9 Address 0037H is the address of the first instruction generated by the IF statement on line #34 of CARS. EXAM displays the message 'START OF IF TEST', and also displays (via a call to macro VARIABLES) the values of the variables involved in the IF condition. The only change since the last such display is that MAINTIME has been incremented to 1.
ICE-88

Sample ICE-88 Sessions at the Terminal

5 EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0023H

6 LIGHTSTATUS = 0001H, CARSWAITING = 0001H
   MAINCYCLELENGTH = 0008H, MAINTIME = 0000H
   SIDECYCLELENGTH = 0005H, SIDETIME = 0000H

7 EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0098H
   CALL DISPLAY
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0028H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0029H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:00DEH

8 CALL DELAY
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0030H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0033H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:00DEH

9 START OF IF TEST
   LIGHTSTATUS = 0001H, CARSWAITING = 0001H
   MAINCYCLELENGTH = 0008H, MAINTIME = 0001H
   SIDECYCLELENGTH = 0005H, SIDETIME = 0000H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:003CH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:003EH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0040H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0045H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0046H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:004AH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:004CH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:004EH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0050H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0052H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0053H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0058H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:005AH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:005DH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:005EH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0062H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0064H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0066H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0068H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:006AH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:006BH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:006CH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:006EH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:0070H
   EMULATION BEGUN

3-19
10  The previous twenty-eight STEPs comprise the IF-test. Address 0079H is the end of the IF test, and 0089H is the end of the main loop. Since address 007CH did not appear, we know that the conditional loop did not execute.

11  Here we are at the beginning of the main loop in CARS.

12  This time through the IF test, MAINTIME is 2. We omit most of the STEPs through the test (address 003CH to 0079H) from the text; these are identical to the series shown at step 9.

13  The start of the main loop again; still no light change.
EMULATION TERMINATED, CS:IP=0000:0072H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0073H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0075H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0077H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H

EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0023H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0098H
CALL DISPLAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0028H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0029H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00DEH
CALL DELAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0030H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0033H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0037H

START OF IF TEST
LIGHTSTATUS = 0001H, CARSWAITING = 0001H
MAINCYCLELENGTH = 0008H, MAINTIME = 0002H
SIDE CYCLE LENGTH = 0005H, SIDETIME = 0000H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003EH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0040H
EMULATION BEGUN

...
14 MAINTIME equals 3; still a long way to go until MAINTIME equals 16. We now omit all steps from the text except the beginning and end of the IF test, so that we can concentrate on the value of MAINTIME.

15 MAINTIME equals 4.

16 MAINTIME equals 5.
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0033H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST

14 LIGHTSTATUS = 0001H, CARSWAITING = 0001H
MAINCYCLELENGTH = 0008H, MAINTIME = 0003H
SIDE CYCLE LENGTH = 0005H, SIDETIME = 0000H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN

•

•

•

EMULATION TERMINATED, CS:IP=0000:0077H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN

•

•

EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST

15 LIGHTSTATUS = 0001H, CARSWAITING = 0001H
MAINCYCLELENGTH = 0008H, MAINTIME = 0004H
SIDE CYCLE LENGTH = 0005H, SIDETIME = 0000H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN

•

•

EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN

•

•

EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST

16 LIGHTSTATUS = 0001H, CARSWAITING = 0001H
MAINCYCLELENGTH = 0008H, MAINTIME = 0005H
SIDE CYCLE LENGTH = 0005H, SIDETIME = 0000H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN

•

•
17 MAINTIME equals 6.
18 MAINTIME equals 7.
19 MAINTIME equals 8.
EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN

• • •

EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST
17 LIGHTSTATUS = 0001H, CARSWAITING = 0001H
MAINCYLENGTH = 0008H, MAINTIME = 0006H
SIDE CYCLE LENGTH = 0005H, SIDETIME = 0000H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN

• • •

EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN

• • •

EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST
18 LIGHTSTATUS = 0001H, CARSWAITING = 0001H
MAINCYLENGTH = 0008H, MAINTIME = 0007H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN

• • •

EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN

• • •

EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST
19 LIGHTSTATUS = 0001H, CARSWAITING = 0001H
MAINCYLENGTH = 0008H, MAINTIME = 0008H
SIDE CYCLE LENGTH = 0005H, SIDETIME = 0000H
EMULATION BEGUN

• • •
20 MAINTIME equals 9.

21 MAINTIME equals 10 (0AH).
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN

EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN

EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST
20 LIGHTSTATUS = 0001H, CARSWAITING = 0001H
MAINCYCLELENGTH = 0008H, MAINTIME = 0009H
SIDECYCLELENGTH = 0005H, SIDETIME = 0000H
EMULATION BEGUN

EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN

EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST
21 LIGHTSTATUS = 0001H, CARSWAITING = 0001H
MAINCYCLELENGTH = 0008H, MAINTIME = 000AH
SIDECYCLELENGTH = 0005H, SIDETIME = 0000H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN

EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN
22 MAINTIME equals 11 (0BH).
23 MAINTIME equals 12 (0CH).
24 MAINTIME equals 13 (0DH).
EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST
22  LIGHTSTATUS = 0001H, CARSWAITING = 0001H
   MAINCYCLELENGTH = 0008H, MAINTIME = 0008H
   SIDECYCLELENGTH = 0005H, SIDETIME = 0000H
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:003CH
   EMULATION BEGUN
   ...
   ...
   ...
EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN
   ...
   ...
   ...
EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST
23  LIGHTSTATUS = 0001H, CARSWAITING = 0001H
   MAINCYCLELENGTH = 0008H, MAINTIME = 0008H
   SIDECYCLELENGTH = 0005H, SIDETIME = 0000H
   EMULATION BEGUN
   ...
   ...
   ...
   ...
EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN
   ...
   ...
   ...
EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST
24  LIGHTSTATUS = 0001H, CARSWAITING = 0001H
   MAINCYCLELENGTH = 0008H, MAINTIME = 000DH
   EMULATION BEGUN
   EMULATION TERMINATED, CS:IP=0000:003CH
   EMULATION BEGUN
   ...
   ...
   ...
EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN
25 MAINTIME equals 14 (0EH).

26 MAINTIME equals 15 (0FH).

27 Finally, MAINTIME equals 16 or two times MAINCYCLELENGTH. This time the condition ‘CARSWAITING = 1 AND MAINTIME >= 2 * MAINCYCLELENGTH’ is TRUE and we should see a call to CYCLE at the end of the IF test.
EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST
25 LIGHTSTATUS = 0001H, CARSWAITING = 0001H
MAINCYCLELENGTH = 0008H, MAINTIME = 0000H
SIDELENGTH = 0005H, SIDETIME = 0000H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN

EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN

EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST
26 LIGHTSTATUS = 0001H, CARSWAITING = 0001H
MAINCYCLELENGTH = 0008H, MAINTIME = 0000H
SIDELENGTH = 0005H, SIDETIME = 0000H
EMULATION BEGUN

EMULATION TERMINATED, CS:IP=0000:0079H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN

EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST
27 LIGHTSTATUS = 0001H, CARSWAITING = 0001H
MAINCYCLELENGTH = 0008H, MAINTIME = 0010H
SIDELENGTH = 0005H, SIDETIME = 0000H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003EH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0040H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0045H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0046H
28 Address 7CH in the beginning of the conditional ‘DO’ loop.

29 Here is the beginning of CYCLE (0A7H). CYCLE is a loop controlled by the statement ‘DO WHILE SIDETIME <= SIDECYCLELENGTH’; SIDECYCLELENGTH is 5, so the loop should exit when SIDETIME equals 6. We could have included ICE-88 commands in macro EXAM to examine CYCLE more closely (LIGHTSTATUS should be set to zero, and SIDETIME should increment on each iteration). In our example, however, we simply wait for CYCLE to return to the main program. The rest of the display on this page shows two iterations of CYCLE. We have omitted the printout of the remaining iterations.
ICE-88

Sample ICE-88 Sessions at the Terminal

EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:004AH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:004CH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:004FH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0050H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0052H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0053H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0058H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:005AH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:005DH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:005EH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0062H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0064H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0066H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0068H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:006AH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:006CH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:006FH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0070H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0072H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0073H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0075H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0077H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:007CH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00A7H
CALL CYCLE
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00A8H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00AAH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00AFH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00B4H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00B8H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00BCH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00C1H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00C5H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0098H
CALL DISPLAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00CAH

3-33
30 Address 0DBH is the end of CYCLE. Two steps later, 07FH is the return address from the call to cycle.
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00CBH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00DEH
CALL DELAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00D3H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00B4H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00B8H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00BCH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00C1H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00C5H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0098H
CALL DISPLAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00CAH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00CBH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00DEH
CALL DELAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00D3H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00B4H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00B8H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00BCH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00C1H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00C5H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0098H
CALL DISPLAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00CAH

•
•
•

EMULATION TERMINATED, CS:IP=0000:0098H
CALL DISPLAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00CAH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00CBH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00DEH
CALL DELAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00D3H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00B4H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00B8H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00BCH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00BEH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00B6H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00D6H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00DBH
EMULATION BEGUN
31 Here we are back at the start of the main loop.

32 The display of variables shows LIGHTSTATUS at 1 and CARSWAITING at zero. SIDETIME is 6 as we expected. MAINTIME is 1 and will continue to increment as long as we allow the program to emulate.

33 We consider this test “successful,” and abort the emulation by pressing the ESC key.

34 Now to test the second condition. The macro call ‘TEST 2’ produces an expansion of macro TEST; this time CARSWAITING is set to 2. Otherwise, the expansion produces an executable macro identical to ‘TEST 1’ shown in step 4.
Sample ICE-88 Sessions at the Terminal

EMULATION TERMINATED, CS:IP=0000:00DCH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:007FH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0084H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0089H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:201FH

31 STARTING MAIN LOOP
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0023H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0030H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0033H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0037H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003EH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0040H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0043H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0047H

32 LIGHTSTATUS = 0001H, CARSWAITING = 0000H
MAINCYCLELENGTH = 0008H, MAINTIME = 0001H
SIDECYCLELENGTH = 0005H, SIDETIME = 0006H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:004EH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0051H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0053H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0057H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0061H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0063H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0067H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:006BH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:006DH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0073H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0077H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:007BH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:007FH

33 PROCESSING ABORTED

*...:TEST 2
*...:GO FROM .START Till ..CARS#30 EXECUTED
*...:VARIABLES
*...:WRITE 'LIGHTSTATUS = ',!LIGHTSTATUS, ', CARSWAITING = ',!CARSWAITING
*...:WRITE 'MAINCYCLELENGTH = ',!MAINCYCLELENGTH, ', MAINTIME = ',!MAINTIME
*...:WRITE 'SIDE CYCLE LENGTH = ',!SIDE CYCLE LENGTH, ', SIDETIME = ',!SIDETIME
*...:EM ;END OF MACRO VARIABLES
*...:EXAM
*...:REPEAT
*...:STEP
*...:IF CS = SEG(.DISPLAY) AND IP = OFF(.DISPLAY) THEN
*...:WRITE 'CALL DISPLAY'
*...:POPOP88 IP ;RESTORE RETURN ADDRESS
*...:IP = WORD SS:SP ;POP PARAMETER OFF STACK
*...:SP = SP + 2T ;MOVE POINTER TO NEW TOP OF STACK
*...:EM ;END OF MACRO POP88
*...:SP = SP + 2T ;DISCARD PARAMETER
*...:ORIF CS = SEG(.DELAY) AND IP = OFF(.DELAY) THEN
*...:WRITE 'CALL DELAY'
*...:POPOP88 IP
*...:IP = WORD SS:SP ;POP PARAMETER OFF STACK
*...:SP = SP + 2T ;MOVE POINTER TO NEW TOP OF STACK
*...:EM ;END OF MACRO POP88
*...:SP = SP + 2T
*...:ORIF CS = SEG(.CARS#30) AND IP = OFF(.CARS#30) THEN
*...:WRITE 'STARTING MAIN LOOP'
*...:ORIF CS = SEG(.CARS#34) AND IP = OFF(.CARS#34) THEN
*...:WRITE 'START OF IF TEST'
*...:VARIABLES
*...:WRITE 'LIGHTSTATUS = ',!LIGHTSTATUS, ', CARSWAITING = ',!CARSWAITING
*...:WRITE 'MAIN CYCLE LENGTH = ',!MAIN CYCLE LENGTH, ', MAINTIME = ',!MAINTIME
35 We emulate to the start of the main loop, as before. This time CARSWAITING is 2, and we should CALL CYCLE as soon as MAINTIME equals 8. We omit the intermediate steps.

36 MAINTIME is 8, equal to MAINCYCLELENGTH.
ICE-88 Sample ICE-88 Sessions at the Terminal

```plaintext
*****WRITE 'SIDECYCLELENGTH =',!SIDECYCLELENGTH,', SIDETIME =',!SIDETIME
*****EM ;END OF MACRO VARIABLES
*****ORIF CS = SEG(.CYCLE) AND IP = OFF(.CYCLE) THEN
*****WRITE 'CALL CYCLE'
*****ENDIF
*****ENDREPEAT
***EM ;END OF MACRO EXAM

35 EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0023H
LIGHTSTATUS = 0001H, CARSWAITING = 0002H
MAINCYCLELENGTH = 0008H, MAINTIME = 0000H
SIDECYCLELENGTH = 0005H, SIDETIME = 0006H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0098H
CALL DISPLAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0028H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0029H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00DEH
CALL DELAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:030H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0033H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST
LIGHTSTATUS = 0001H, CARSWAITING = 0002H
MAINCYCLELENGTH = 0008H, MAINTIME = 0000H
SIDECYCLELENGTH = 0005H, SIDETIME = 0006H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN

36 START OF IF TEST
LIGHTSTATUS = 0001H, CARSWAITING = 0002H
MAINCYCLELENGTH = 0008H, MAINTIME = 0000H
SIDECYCLELENGTH = 0005H, SIDETIME = 0006H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003EH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0041H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0045H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0046H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:004AH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:004CH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:004FH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0050H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0052H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0053H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0058H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:005AH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:005CH
```
37 And here's the beginning of CYCLE. Once more we omit the steps in CYCLE from the test.

38 This is the end of CYCLE, and the return to the main program.

39 Back to the start of the main loop.
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:005EH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0062H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0064H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0066H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0068H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:006AH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:006CH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:006EH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0070H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0072H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0073H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0075H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0077H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:007CH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00B8H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00BCH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00BEH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00D6H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00DBH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00DCH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:007FH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0084H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0088H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:001FH
STARTING MAIN LOOP
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0023H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0098H
CALL DISPLAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0028H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0029H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:00DEH
CALL DELAY
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0030H
40 We consider this test "successful," abort emulation, and exit from the ICE-88 emulator back to ISIS-II (hyphen prompt).
ICE-88

Sample ICE-88 Sessions at the Terminal

EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0033H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:0037H
START OF IF TEST

40 LIGHTSTATUS = 0001H, CARSWAITING = 0000H
MAINCYCLELENGTH = 0008H, MAINTIME = 0001H
SIDE CYCLE LENGTH = 0005H, SIDETIME = 0006H
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003CH
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=0000:003EH
PROCESSING ABORTED
*
*EXIT
Introduction

The ICE-88 software provides you with an easy-to-use English language command set for controlling ICE-88 emulator execution in a variety of functional modes.

The commands enable you to:

- Initialize the ICE-88 system; map your program to memory in your system, ICE-88 memory, disk memory, or in the Intellec Microcomputer Development System; and load your program from a diskette file.
- Specify starting and stopping conditions for emulation.
- Execute real-time emulation of your software (and hardware).
- Execute single-step emulation.
- Specify conditions for trace data collection.
- Collect and display trace data on conditions occurring during emulation.
- Display and alter 8088 registers, memory locations, and I/O ports.
- Copy the (modified) program from mapped memory to a diskette file, and exit the ICE-88 system.

An example of one complete command, in this case one of the forms of the GO command, is shown in figure 4-1. This command causes emulation to start and specifies the conditions that will halt emulation. The command is made up of ten separate “words” (character strings that are referred to as tokens): GO, FROM, 0123H, TILL, 1000H, TO, 1100H, READ, USING, and CS. Each of these tokens provides a particular element of information necessary to inform the ICE-88 emulator of the specific command functions (see table 4-1). The tokens also form the following command components: the FROM clause, match-range, match-status, segment-register-usage, match-condition, and TILL clause. This string of tokens requests the ICE-88 emulator to “start emulation at location 0123H and to continue emulation until data is read from any memory location within the match-range (partition) of addresses 1000H through 1100H using the CS segment register in the effective address calculation.” Each command is composed of one or more such tokens.
## Table 4-1. Definition of GO Command Functions

<table>
<thead>
<tr>
<th>Token Number</th>
<th>Name</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>GO</td>
<td>GO command specifier; requests and initiates emulation.</td>
</tr>
<tr>
<td>2</td>
<td>FROM</td>
<td>Indicates that the next token or expression is the starting address for emulation.</td>
</tr>
<tr>
<td>3</td>
<td>0123H</td>
<td>Starting address in hexadecimal radix.</td>
</tr>
<tr>
<td>2,3</td>
<td>FROM clause</td>
<td>FROM 0123H causes the instruction pointer (IP) to be loaded with 0123H, the starting address for emulation. Also, the code segment register (CS) is loaded with 0.</td>
</tr>
<tr>
<td>4</td>
<td>TILL</td>
<td>Indicates that the breakpoint (halting) parameters are to follow.</td>
</tr>
<tr>
<td>5</td>
<td>1000H</td>
<td>Specifies the lowest address of a range of memory locations. This parameter is the lower bound of a memory partition.</td>
</tr>
<tr>
<td>6</td>
<td>TO</td>
<td>Indicates that the upper bound (address) of the range (partition) of memory locations is to follow.</td>
</tr>
<tr>
<td>7</td>
<td>1100H</td>
<td>Specifies the highest address of the range of memory locations.</td>
</tr>
<tr>
<td>5,6,7</td>
<td>match-range</td>
<td>Emulation is to halt if an access to any memory location whose address falls within the range of 1000H to 1100H.</td>
</tr>
<tr>
<td>8</td>
<td>READ</td>
<td>Emulation is to halt if any of the above memory locations are read.</td>
</tr>
<tr>
<td>9</td>
<td>USING</td>
<td>Indicates a segment register is to follow.</td>
</tr>
<tr>
<td>10</td>
<td>CS</td>
<td>The code segment register (CS) must be used in the effective address calculation in order to match.</td>
</tr>
<tr>
<td>9,10</td>
<td>segment-register-usage</td>
<td>Emulation is to halt if the CS is used.</td>
</tr>
<tr>
<td>5 thru 10</td>
<td>match-condition</td>
<td>Emulation is to halt if data is read from any memory location in the match-range.</td>
</tr>
<tr>
<td>4 thru 10</td>
<td>TILL clause</td>
<td>TILL 1000H TO 1100H READ USING CS specifies that the emulation is to halt whenever the match-condition is met. This clause is also called the “GO-register” in the ICE-88 language.</td>
</tr>
</tbody>
</table>

Note: The match-condition consists of the three sub-conditions: match-range (tokens 5,6,7), match-status (token 8), and segment-register-usage (tokens 9,10). All three of these conditions must be matched for emulation to halt.

As briefly indicated in figure 4-1, the commands are written in an ICE-88 command language composed of a unique character set and vocabulary of tokens augmented by a particular set of syntactic rules. The tokens are constructed from the character set and in turn are used to construct commands. The tokens consist of a set of predefined literals augmented by user-defined literals that provide symbolic references. Table 4-1 contains the definition of each token shown in figure 4-1.
The purpose of this chapter is to present a detailed specification of the ICE-88 command language. The language consists of two parts, a vocabulary that is used to convey elements of information to the ICE-88 emulator and a “grammar” (syntactic rules) used to group command words into command constructs such as the FROM clause shown in figure 4-1. The remainder of this chapter is devoted to the presentation of the command language. The initial discussion deals with class-names and the notation used to describe the language and will include a listing of the syntactic rules that govern command construction. This will be followed by a presentation of the command literals (keywords) and a discussion of symbolic references.

**Notation and Conventions Used in This Manual**

This manual employs a set of notational symbols and conventions to describe the structure of commands and other language constructs. The features of this notation are described in the following paragraphs. Table 4-2 contains the notational symbols used to define and describe the command structures.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>≡</td>
<td>“is defined as”</td>
</tr>
<tr>
<td>::</td>
<td>Mutual exclusion</td>
</tr>
<tr>
<td>...</td>
<td>May be repeated indefinitely</td>
</tr>
<tr>
<td>{ }</td>
<td>At least one entry must be selected. If more than one entry is selected, they may be selected in any order.</td>
</tr>
<tr>
<td>{ }</td>
<td>One of the enclosed entries must be selected.</td>
</tr>
<tr>
<td>[ ]...</td>
<td>Selection of the enclosed entries is optional. If more than one entry is selected, they may be selected in any order.</td>
</tr>
<tr>
<td>[ ]</td>
<td>Selection of the enclosed entries is optional but only one entry may be selected. If this symbol encloses only one entry, that entry is optional.</td>
</tr>
</tbody>
</table>

In addition to the above notational symbols, a set of class-names is used to assist in the definition and description of entities in the ICE-88 command language. Each class-name is an identifier for a specific set of characters, mnemonics, or constructs, and is always shown in *lower-case italics*. Any character string not in lower-case italics is a specific character, mnemonic or construct. For example, the class-name *segment-register* refers to the entire class of segment registers. The character string CS refers to the Code Segment Register, which is one of the four segment registers.

As shown in figure 4-1, the smallest meaningful unit of information contained within a command is a mnemonic character string that is the equivalent of a word. Examples are: GO, 0123H, and FROM. These mnemonics are assigned the class-name: *token*. In addition to these basic elements, the tokens are combined into multi-token forms such as the FROM clause and match-condition shown in figure 4-1.
The ICE-88 vocabulary is made up of two classes of mnemonics: tokens and special-tokens:

\[
\begin{align*}
token & : constant :: keyword :: symbol :: string :: operand \\
special-token & : operator :: punctuation-mark :: delimiter :: terminator
\end{align*}
\]

The notational symbol (:\:) specifies mutual exclusion. That is, a token is a constant or keyword or symbol or string or operand.

Each of the above classes of tokens and special-tokens is defined later in this chapter or in the next chapter in the discussion of expressions.

**Syntactic Rules Used in the Manual**

This manual employs a set of conventions to describe the structure of commands and other ICE-88 language forms. Items 1 and 2 below specify the use of class-names and tokens respectively. The features of this notation system are as follows:

1. A lower-case italicized entry in the description of a command is the class-name for a set or class of tokens. To create an actual operable command, you must enter a particular member of this class. A class-name never appears in an actual operable command. For example, the lower-case entry:

   \[\text{breakpoint-register}\]

   means that the command will accept any of the three tokens: BR0, BR1 or BR (BR means BR0 and BR1). Classes of tokens that have generalized usage, such as the classes of reference keywords and command keywords, are explained and assigned class-names in this chapter. Additional classes of tokens that appear in the syntax descriptions of particular commands are explained in the discussion of semantics that accompanies those commands.

2. An upper-case entry is a token that must be used literally as given. A valid abbreviation of that token may substitute for the full token as given. The token may be a command word, or it may be a particular member of a class of references. For example, the upper-case entry

   \[\text{DEFINE}\]

   is a command word that must be used as given unless abbreviated. The abbreviation DEF may be used in place of DEFINE. As another example, the upper-case entry

   \[\text{BR1}\]

   means that breakpoint register 1 must be named as and where given.

3. A single required entry is shown without any enclosures, whereas a single optional entry is denoted by enclosing in brackets. For example, in the command syntax

   \[\text{STEP [FROM address]}\]

   the token STEP is required. The significance of the brackets around the entry: FROM address means that its selection is optional in this command.
4. Where only one entry must be selected from a menu of two or more entries, the choices for the required entry are denoted by enclosing them in braces. For example,

\[ \text{TRACE} = \{ \text{FRAME}, \text{INSTRUCTION} \} \]

indicates that FRAME or INSTRUCTION must be selected; the tokens TRACE and \( = \) are required as given.

5. An optional entry is enclosed in brackets [ ]. For example,

\[ \text{STEP} \{ \text{FROM address} \} \]

means that the command word STEP is required but the clause FROM address is optional in this command.

Where a choice exists for an optional entry, the choices are given in a vertical arrangement enclosed in brackets. For example, the command

\[ \text{DEFINE} \{ \text{module-name} \} \text{ symbol = expr} \]

\[ \{ \text{OF BYTE, OF WORD, OF SINTeger, OF INTEGER, OF POINTER} \} \]

means that DEFINE, symbol, \( = \), and expr are required in this command and module-name is optional. The brackets around the vertical arrangement of memory type designators denotes that selection of a memory type designator is optional but only one designator may be selected per DEFINE command.

6. A group of required inclusive choices is given in a vertical arrangement and enclosed in braces (\{\}) followed by a repeat symbol (\(...\)). "Inclusive" means that more than one of the items can be entered in the same command, and items can appear in any order; no item can be entered more than once. The menu of inclusive items represents a required entry or entries. For example:

\[ \text{match-status-list} \equiv \{ \text{READ, WRITTEN, INPUT, OUTPUT, FETCHED, HALT, ACKNOWLEDGE} \}, ... \]

This notation indicates that one or more items from the vertical list is required to specify a match-status-list. If more than one item is used, they can be in any order and must be separated by commas.

To complete the example:

\[ \text{WRITTEN, HALT, READ, FETCHED} \]

is a valid match-status-list.
7. A group of optional inclusive choices is given in a vertical arrangement and enclosed in brackets and followed by a repeat symbol (...). "Inclusive" means that more than one of the items can be entered in the same command, and the items can appear in any order; no item can be used more than once. The menu of inclusive items represent an optional entry or entries. For example:

\[\text{LOAD } \text{path-name} \left[ \begin{array}{c} \text{NOCODE} \\ \text{NOSYMBOL} \\ \text{NOLINE} \end{array} \right] \ldots\]

This notation indicates that none, one, or more than one choice of NOCODE, NOSYMBOL, and/or NOLINE may be included in one LOAD command; if more than one is used, the entries can be in any order.

To complete the example:

\text{LOAD :F0:TEST NOSYMBOL NOCODE NOLINE}

is a valid command.

8. Where mutually exclusive entries can be shown on one line, the following shorthand notation can be used:

\text{SUFFIX = Y :: O :: Q :: T :: H}

This example is equivalent to

\text{SUFFIX = \{ Y, O, Q, T, H \}}

9. Where an entry can be repeated indefinitely at the user's option, the syntax is notated by enclosing the repeatable entry in brackets [ ] followed by an ellipsis \ldots. For example,

\text{operand \{operator operand\} \ldots}

indicates that \text{operator operand} can be repeated as many times as desired.

Character Set

The valid characters in the ICE-88 command language include upper and lower case alphabetic ASCII characters A through Z and the set of digits 0 through 9. The space serves as a delimiter for tokens, and carriage-return/line-feed characters are also valid, delimiting command lines. A question mark, ?, @ sign, $ sign, and underscore (_) are also valid in user-defined names entered in the command language.

The algebraic operators + and - (binary and unary), the asterisk (*), slash (/), relational operators (=, <, >), the ampersand, semicolon, colon, period, parentheses, exclamation mark (!), pound sign (#), percent sign (%) and comma, constitute the only other valid ASCII characters for the ICE-88 emulator. Non-printing characters are ignored, except tabs, form feed, etc. are treated as spaces. Other characters are errors.
Elements of the ICE-88 Command Language

alphabetic characters:

ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz@?_

numeric characters:

0 1 2 3 4 5 6 7 8 9 (A B C D E F: hexadecimal characters)

special characters:

+ - <= => $ '& ). (; */ ! : , %

This character set is used to construct the vocabulary that constitutes the command language.

Introduction to Tokens

A *token* in the ICE-88 command language is roughly equivalent to a "word" in the English language. It consists of a string of alphanumeric characters that may be augmented by a one or two special character prefix that serves as a *token identifier*. Tokens are divided into the following types: *keywords*, *usernames*, and *special-tokens*. Examples are:

REGISTERS, .START, .MODULE, .SAM, 0400, 123AH.

Keywords

The ICE-88 emulator recognizes a general class of predefined *tokens* that are fixed in the command language. They provide two functions. *Reference keywords* are used to specify locations having unique predefined functions. *Command keywords* specify command type and subfunctions within a command. The following sections define and describe these keyword classes. Each class and associated keyword set is presented in the following paragraphs. Appendix A contains a listing of ICE-88 keywords and their abbreviations.

The reason for discussing the various classes and subsets here is to smooth the later discussions of commands, where the class-names are used to show what elements may appear in which commands.

Reference Keywords

The command language contains a set of system defined mnemonic *tokens* that are used to address system objects. Each device such as the accumulator or a register is assigned a specific mnemonic that is to be used to address and access the contents of that device. These identifiers are called *reference keywords*. *Reference keywords* are used in commands to refer to 8088 processor registers and flags, emulation registers, memory locations, and I/O ports.

The total set of *reference keywords* is subdivided by types, each of which is referenced by a class name. Class names are always shown in lower case italics. For example, the class name *general-register* denotes the set of four 16-bit general work registers in the 8088 processor. A reference keyword is assigned to each element within the given class and is always shown in upper case. For example, "RAX" denotes the contents of the accumulator (RAX register) of the general register set.
Registers

The 8088 register structure contains three files of four 16-bit registers, a set of miscellaneous registers, and a set of four pseudo-registers. The three files of registers are the general register file, the pointer and index file, and the segment register file (see table 4-3). The miscellaneous set consists of the instruction pointer, flag register, CAUSE register, OPCODE register, PIP register, TIMER register, HTIMER register, BUFFERSIZE register, LOWER register, and UPPER register. The pseudo-register set consists of the breakpoint and trace point registers. The miscellaneous register set and the pseudo-registers provide a variety of functions to the ICE-88 emulator that are described in the appropriate command sections of this manual. The register structures are described in the following paragraphs.

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Hardware Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>general-register</td>
<td>8-bit and 16-bit work register</td>
</tr>
<tr>
<td>pointer-register</td>
<td>16-bit address register</td>
</tr>
<tr>
<td>index-register</td>
<td>16-bit address register</td>
</tr>
<tr>
<td>segment-register</td>
<td>16-bit segment reference register</td>
</tr>
<tr>
<td>status-register</td>
<td>8 and 16-bit status registers</td>
</tr>
<tr>
<td>emulation-register</td>
<td>breakpoint and tracepoint registers</td>
</tr>
</tbody>
</table>

General Register File. The RAX, RBX, RCX, and RDX registers compose the General Register File. These registers participate interchangeably in 8088 arithmetic and logical operations. These registers are assigned the following mnemonics:

- **RAX**: Accumulator
- **RBX**: Base Register
- **RCX**: Count Register
- **RDX**: Data Register

Note: these are the 8088 AX, BX, CX, DX registers (i.e., the ICE-88 software requires ‘R’ to be prefixed to the 8088 names).

The general registers are unique within the 8088 as their upper and lower bytes are individually addressable. Thus, each of the general registers contains two 8-bit register files called the H file and L file as illustrated below.

General Register File

Pointer and Index Register File. The BP, SP, SI, and DI registers are called the Pointer and Index Register File. The registers in this group are similar in that they generally contain offset addresses used for addressing within a segment. They can participate interchangeably in 16-bit arithmetic and logical operations and can also be used in address computation. The mnemonics associated with these registers are:
SP: Stack Pointer  
BP: Base Pointer  
SI: Source Index  
DI: Destination Index

The pointer and index registers are illustrated below.

---

**Point and Index Register File**

**Segment Register File.** The CS, DS, SS, and ES registers constitute the Segment Register File. These registers provide a significant function in the memory addressing mechanisms of the 8088. They are similar in that they are used in all memory address computations. The mnemonics associated with these registers are:

- CS: Code Segment Register
- DS: Data Segment Register
- SS: Stack Segment Register
- ES: Extra Segment Register

The contents of the CS register define the current code segment. All instruction fetches are taken to be relative to CS using the instruction pointer (IP) as an offset.

The contents of the DS register define the current data segment. All data references except those involving BP, SP, or DI in a string instruction are taken by default to be relative to DS.

The contents of the SS register define the current stack segment. All data references which implicitly or explicitly involve SP or BP are taken by default to be relative to SS.

The contents of the ES register define the current extra segment. The extra segment has no specific use, although it is usually treated as an additional data segment.

The segment registers are illustrated below.

---

**Segment Register File**
Elements of the ICE-88 Command Language

Status Registers

The instruction pointer, flag register, CAUSE register, OPCODE register, PIP register, TIMER register, HTIMER register, BUFFERSIZE register, LOWER register, and UPPER register, constitute the status register set. These registers provide a variety of functions to the ICE-88 emulator. These registers are assigned the following mnemonics:

- IP: Instruction Pointer
- RF: Flag Register
- CAUSE: CAUSE Register
- OPCODE: OPCODE Register
- PIP: Previous Instruction Register
- TIMER: TIMER Register
- HTIMER: HTIMER Register
- BUFFERSIZE: BUFFERSIZE Register
- LOWER: LOWER Register
- UPPER: UPPER Register

The contents of the IP register define the offset to the CS register in instruction address computations. The Flag Register contains the status flag values in the same format as that pushed by the 8088 PUSHF instruction. The CAUSE register retains the cause of the last break in emulation and the OPCODE register stores the opcode fetched in the last instruction-fetch cycle in trace data. The Previous Instruction Register stores the displacement part of the address of the last instruction-fetch in trace data. TIMER contains the low-order 16 bits of the 2-MHz timer indicating how long emulation has run (read only). HTIMER contains the high-order 16 bits of the timer (read-only). BUFFERSIZE contains the count (displayed in decimal only) of frames of valid trace data collected in the trace buffer (16 bit, read-only). LOWER contains the lowest available address in Intellec memory above the ICE-88 software (16 bit, read-only). UPPER contains the highest available address in Intellec system memory below the user's symbol table (16 bit, read-only).

The status registers are illustrated below.

![Status Registers Diagram]

Status Registers

The Flag Register contains nine status bits. The following mnemonics are assigned to each of the status values in the register:

- AFL: Auxiliary-carry out of low byte to high byte
- CFL: Carry or borrow out of high bit
- DFL: Direction of string manipulation instruction
- IFL: Interrupt-enable (external)
OFL: Overflow flag for signed arithmetic
PFL: Parity
SFL: Sign of the result of an operation
TFL: Trap used to place processor in single step mode for debug
ZFL: Zero indicates a zero value result of an instruction

AFL is set if an instruction caused a carry out of bit 3 and into bit 4 of a resulting value. CFL is set if an instruction caused a carry or a borrow out of the high order bit. DFL controls the direction of the string manipulation instructions. IFL enables or disables external interrupts. OFL denotes an overflow condition in a signed arithmetic operation. SFL indicates the sign of the result of an operation. TFL places the processor in a single-step mode for program debugging. ZFL indicates a zero valued result of an instruction. The positions of the status bits in the RF Register are shown below.

<table>
<thead>
<tr>
<th>15</th>
<th>14</th>
<th>13</th>
<th>12</th>
<th>11</th>
<th>10</th>
<th>9</th>
<th>8</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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>OFL</td>
<td>DFL</td>
<td>IFL</td>
<td>TFL</td>
<td>SFL</td>
<td>ZFL</td>
<td>AFL</td>
<td>PFL</td>
<td>CFL</td>
<td></td>
</tr>
</tbody>
</table>

Flag Register

The CAUSE Register stores the cause for the last break in emulation. The contents of this register are defined below.

<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>
</table>

Cause Register

The byte returned by the "Read Break Cause" hardware command contains the following bit values (if bit = 1, then the specified condition is true, otherwise false):

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Breakpoint 0 matched</td>
</tr>
<tr>
<td>1</td>
<td>Breakpoint 1 matched</td>
</tr>
<tr>
<td>2</td>
<td>Both breakpoints matched sequentially</td>
</tr>
<tr>
<td>3</td>
<td>Guarded memory access occurred</td>
</tr>
<tr>
<td>4</td>
<td>User aborted processing</td>
</tr>
<tr>
<td>5</td>
<td>Timeout on user READY</td>
</tr>
<tr>
<td>6</td>
<td>Timeout on user HOLD</td>
</tr>
</tbody>
</table>

8088 Pin References

The ICE-88 emulator provides access to seven 8088 pins. The pin names reference 1-bit values. Pin names are read-only references only. The following mnemonics are assigned to reference the 8088 processor pins shown below.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>8088 Pin</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDY</td>
<td>READY</td>
<td>Acknowledgement from addressed memory or I/O device that it has completed data transfer.</td>
</tr>
<tr>
<td>NMI</td>
<td>NMI</td>
<td>Non-maskable interrupt.</td>
</tr>
</tbody>
</table>
Elements of the ICE-88 Command Language

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEST</td>
<td>Used by the wait-for-signal instruction for processor synchronization purposes.</td>
</tr>
<tr>
<td>HOLD</td>
<td>Request for local bus “hold.”</td>
</tr>
<tr>
<td>RST</td>
<td>Causes processor to immediately terminate present activity.</td>
</tr>
<tr>
<td>MN</td>
<td>Specifies minimum/maximum configuration (1 = minimum).</td>
</tr>
<tr>
<td>IR</td>
<td>Maskable interrupt request.</td>
</tr>
</tbody>
</table>

Emulation Registers

The emulation registers consist of the breakpoint registers and the trace registers.

<table>
<thead>
<tr>
<th>Type</th>
<th>Class Name</th>
<th>Keywords</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breakpoint register</td>
<td>break-reg</td>
<td>BR0, BR1, BR</td>
</tr>
<tr>
<td>Trace point register</td>
<td>trace-reg</td>
<td>ONTRACE, OFFTRACE</td>
</tr>
<tr>
<td>GO register</td>
<td>go-reg</td>
<td>GR</td>
</tr>
</tbody>
</table>

The term `break-reg` is the class name for the two breakpoint registers used to halt emulation. The term `trace-reg` is the class name for the two registers that control tracing. The term `go-reg` refers to the GO-register, an ICE-88 pseudo-register that controls the breaking of real-time emulation.

Command Keywords

The command keywords specify command types and command functions to be executed. ICE-88 commands are of three major types: simple commands, compound commands, and macro commands. The following sections define the associated command formats and illustrate the use of keywords in each of the command types. Each of the formats is specified and illustrated by example using appropriate command keywords. The full vocabulary of command keywords is presented following the command descriptions.

Simple Commands

Simple commands are of one of three types.

- **Set/change commands**
- **Display commands**
- **Execution commands**

The following sections describe the formats and provides examples of each of these simple commands.

Set/Change Commands. The set/change commands have the following format:

\[ \text{item-type} \text{[item-qualifier]}\ldots = \text{new-setting} \]

where

- **item-type** A keyword or user name of an alterable element.
- **item-qualifier** A keyword, user name or value used to provide further specification of the particular element that is to be set or altered.
- **new-setting** The value that the specified item is to be set to.
Examples:

BR0 = 1000H EXECUTED
BYTE 10FFH = 3AH
..MOD1 .SYMBA .SYMBB = 10FFH

Display Commands. The display commands have the following format:

`item-type [item-qualifier]`...

where

- `item-type` A keyword or user name of a displayable element or set of displayable elements.
- `item-qualifier` A keyword, user name or value used to provide further specification of the particular element(s) to be displayed.

Examples:

BR0
BYTE 10FFH
..MOD1 .SYMBA .SYMBB
REGISTER
RAX
FLAG
STACK 10

Execution Commands. The execution commands have the following format:

`command-verb [command-parameter]`...

where

- `command-verb` A command keyword that describes an action that is to be performed.
- `command-parameter` A keyword that specifies the objects of the action denoted by the `command-verb`.

Examples:

GO
GO FROM .START
GO FROM .START TILL 1000H EXECUTED
TRACE
PRINT ALL
PRINT 10
MOVE -10

Compound Commands

Complete description of the formats of the compound commands and the use of keywords with these commands is contained in Chapter 8.

Macro Commands

Complete description of the formats of the macro commands and the use of keywords with these commands is contained in Chapter 8.
Utility Command Keywords

The Intel Systems Implementation Supervisor (ISIS-II) is the diskette operating system for the Intellec Microcomputer Development System. The ICE-88 emulator runs under ISIS-II control, and can call upon ISIS-II for file management functions through the utility commands. These commands employ the following command keywords:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXIT</td>
<td>Commands control to be returned to ISIS-II.</td>
</tr>
<tr>
<td>LIST</td>
<td>Commands emulation output to be copied to printer or file.</td>
</tr>
<tr>
<td>LOAD</td>
<td>Commands user program to load into memory accessed by the ICE-88 emulator.</td>
</tr>
<tr>
<td>NOCODE</td>
<td>A Modifier specifying that program code is not to be saved.</td>
</tr>
<tr>
<td>NOLINE</td>
<td>A Modifier specifying that the line number table is not to be saved.</td>
</tr>
<tr>
<td>NOSYMBOL</td>
<td>A Modifier specifying that the symbol table is not to be saved to diskette.</td>
</tr>
<tr>
<td>SAVE</td>
<td>Commands user program to be saved on an external device.</td>
</tr>
</tbody>
</table>

Number Base and Radix Commands

The ICE-88 emulator commands and displays involve several different number bases (radixes). This section describes the command keywords and radixes used to control the number base.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASE</td>
<td>Set or display console output radix.</td>
</tr>
<tr>
<td>SUFFIX</td>
<td>Set or display console input radix.</td>
</tr>
<tr>
<td>EVALUATE</td>
<td>Commands a numeric constant or expression to be displayed in all five possible output radixes.</td>
</tr>
<tr>
<td>H</td>
<td>Hexadecimal (base 16).</td>
</tr>
<tr>
<td>O</td>
<td>Octal (base 8).</td>
</tr>
<tr>
<td>Q</td>
<td>Octal (base 8).</td>
</tr>
<tr>
<td>T</td>
<td>Decimal (base 10).</td>
</tr>
<tr>
<td>Y</td>
<td>Binary (base 2).</td>
</tr>
<tr>
<td>ASCII</td>
<td>ASCII character code.</td>
</tr>
</tbody>
</table>

Memory Mapping Command Keywords

These commands display, declare, set or reset the ICE-88 memory mapping. The ICE-88 emulator uses these maps to determine what memory is installed on a prototype system and what memory resources are being “borrowed” from the Intellec system and ICE-88 emulator for testing purposes. These commands employ the following keywords:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISK</td>
<td>Maps logical memory segments into a diskette file.</td>
</tr>
<tr>
<td>GUARDED</td>
<td>Declares memory segments to be guarded. Accesses to addresses in these segments are error conditions.</td>
</tr>
<tr>
<td>ICE</td>
<td>Maps memory segments into ICE “real-time” memory.</td>
</tr>
<tr>
<td>INTELLEC</td>
<td>Maps memory segments to expanded Intellec memory.</td>
</tr>
<tr>
<td>MAP</td>
<td>Commands the ICE-88 emulator to display, declare, set, or reset memory mapping.</td>
</tr>
<tr>
<td>NOVERIFY</td>
<td>Specifies that the normal read-after-write verification of data loaded into memory be suppressed.</td>
</tr>
<tr>
<td>RESET</td>
<td>Resets the memory mapping.</td>
</tr>
<tr>
<td>USER</td>
<td>Maps logical segments into user’s prototype memory.</td>
</tr>
</tbody>
</table>
Hardware Register Command Keywords

This section presents the keywords used in the ICE-88 emulator to specify and modify hardware register commands.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOCK</td>
<td>Command keyword indicating that a system clock specification is to follow.</td>
</tr>
<tr>
<td>DISABLE</td>
<td>Command keyword indicating that a command function is to be disabled.</td>
</tr>
<tr>
<td>DONE</td>
<td>Command modifier setting timeout on DONE.</td>
</tr>
<tr>
<td>ENABLE</td>
<td>Command keyword indicating that a command function is to be enabled.</td>
</tr>
<tr>
<td>ERROR</td>
<td>Command modifier specifying that an error is to be reported whenever the command signal times out.</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>The ICE-88 emulator is to operate from an external (user-provided) clock.</td>
</tr>
<tr>
<td>FLAG</td>
<td>Contents of the 9 flags are to be displayed.</td>
</tr>
<tr>
<td>HARDWARE</td>
<td>Reset command modifier, causes a hardware reset.</td>
</tr>
<tr>
<td>INFINITE</td>
<td>Set command signal timeout to &quot;infinite,&quot; disabling timeout.</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>The ICE-88 emulator is to operate from an internal (8088-provided) clock.</td>
</tr>
<tr>
<td>NOERROR</td>
<td>Specifies that the command signal is not to halt emulation.</td>
</tr>
<tr>
<td>PIN</td>
<td>Contents of the six 8088-input pins are to be displayed.</td>
</tr>
<tr>
<td>REGISTER</td>
<td>Contents of the thirteen 16-bit 8088 registers and RF are to be displayed.</td>
</tr>
<tr>
<td>RWTIMEOUT</td>
<td>Used to enable or disable memory access timeout.</td>
</tr>
</tbody>
</table>

Memory and Port Contents Command Keywords

These commands give access to the content or current value stored in designated memory locations or input/output ports.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSOLUTE</td>
<td>Display all addresses as 20-bit numbers.</td>
</tr>
<tr>
<td>BASE</td>
<td>Display all addresses in base and displacement format.</td>
</tr>
<tr>
<td>BOOL</td>
<td>Display expression as a boolean value.</td>
</tr>
<tr>
<td>BYTE</td>
<td>1-byte, unsigned integer value</td>
</tr>
<tr>
<td>INTEGER</td>
<td>2-byte, signed value</td>
</tr>
<tr>
<td>LENGTH</td>
<td>Indicates that an integer value denoting the length of a partition is to follow.</td>
</tr>
<tr>
<td>ASM</td>
<td>Indicates that a range of memory is to be disassembled into 8088 assembly language mnemonics.</td>
</tr>
<tr>
<td>NESTING</td>
<td>Indicates that the starting and return addresses of all currently active procedures are to be displayed.</td>
</tr>
<tr>
<td>POINTER</td>
<td>4-byte, pointer value</td>
</tr>
<tr>
<td>PORT</td>
<td>Reference to 8-bit I/O ports</td>
</tr>
<tr>
<td>SINTEGER</td>
<td>1-byte, signed integer value</td>
</tr>
<tr>
<td>STACK</td>
<td>Indicates that words from the user’s stack is to be displayed.</td>
</tr>
<tr>
<td>SYMBOLICALLY</td>
<td>Causes each numeric value output to be displayed as a symbol or source statement number plus a remainder.</td>
</tr>
<tr>
<td>WORD</td>
<td>2-byte, unsigned integer value</td>
</tr>
<tr>
<td>WPORT</td>
<td>Reference to 16-bit I/O ports</td>
</tr>
</tbody>
</table>
Symbol Table and Statement Number Table Command Keywords

The ICE-88 emulator maintains a symbol table and source program statement number table to enable the user to refer to memory addresses and other values by using symbolic references and statement number references in ICE-88 commands. The following are command keywords contained in these commands:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE</td>
<td>Command keyword used to define a symbol.</td>
</tr>
<tr>
<td>DOMAIN</td>
<td>Keyword used to establish a default module for source statement number references.</td>
</tr>
<tr>
<td>LINE</td>
<td>Specifies the display of all of the source statement number table.</td>
</tr>
<tr>
<td>MODULE</td>
<td>Command modifier for the ICE-88 module table.</td>
</tr>
<tr>
<td>OF</td>
<td>Specifies that a memory type designation is to follow.</td>
</tr>
<tr>
<td>REMOVE</td>
<td>Specifies that symbolic reference(s) is/are to be deleted.</td>
</tr>
<tr>
<td>SYMBOL</td>
<td>Command modifier for the entire ICE-88 symbol table.</td>
</tr>
<tr>
<td>TYPE</td>
<td>Indicates an assignment or change of memory type to a symbolic reference.</td>
</tr>
</tbody>
</table>

Emulation Control Command Keywords

The emulation control commands permit the user to specify the starting address where emulation is to begin, and to specify and display the software or hardware conditions for halting emulation and returning control to the console for further commands. These commands employ the following keywords:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACKNOWLEDGE</td>
<td>Match on 8088 interrupt acknowledge.</td>
</tr>
<tr>
<td>AND</td>
<td>Indicates a match on both breakpoint registers required to halt emulation.</td>
</tr>
<tr>
<td>DOWN</td>
<td>Less than or equal to the referenced address or data value.</td>
</tr>
<tr>
<td>EXECUTED</td>
<td>An instruction fetch out of the execution queue.</td>
</tr>
<tr>
<td>FOREVER</td>
<td>All break conditions disabled.</td>
</tr>
<tr>
<td>FETCHED</td>
<td>Memory read into the execution queue.</td>
</tr>
<tr>
<td>FROM</td>
<td>Keyword introducing a starting address.</td>
</tr>
<tr>
<td>GO</td>
<td>Command keyword that starts emulation.</td>
</tr>
<tr>
<td>HALT</td>
<td>8088 processor halt.</td>
</tr>
<tr>
<td>INPUT</td>
<td>I/O port read.</td>
</tr>
<tr>
<td>LOCATION</td>
<td>Denotes the following constant or expression to be an address.</td>
</tr>
<tr>
<td>OR</td>
<td>Indicates that a match on either breakpoint register will halt emulation.</td>
</tr>
<tr>
<td>OBJECT</td>
<td>Indicates that a memory reference or typed memory reference is to follow.</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>I/O port write.</td>
</tr>
<tr>
<td>READ</td>
<td>Memory read.</td>
</tr>
<tr>
<td>STEP</td>
<td>Single-step emulation command.</td>
</tr>
<tr>
<td>TILL</td>
<td>A keyword introducing one or more match or halt conditions.</td>
</tr>
<tr>
<td>UP</td>
<td>Greater than or equal to the referenced address or data value.</td>
</tr>
<tr>
<td>USING</td>
<td>Indicates that a segment register is to be specified.</td>
</tr>
<tr>
<td>VALUE</td>
<td>Denotes the following constant or expression to be a data value.</td>
</tr>
<tr>
<td>WRITTEN</td>
<td>Memory write.</td>
</tr>
</tbody>
</table>
Trace Control Commands

The trace control commands allow the user to display or change the match condition in either or both of two tracepoint registers and to establish a tracepoint to conditionally start and stop trace collection. These commands employ the following keywords:

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDR</td>
<td>20-bit address in hexadecimal format.</td>
</tr>
<tr>
<td>ALL</td>
<td>A function keyword indicating that the entire trace buffer contents are to be displayed.</td>
</tr>
<tr>
<td>CONDITIONALLY</td>
<td>Indicates trace is to be turned on when ONTRACE matches and turned off when OFFTRACE matches.</td>
</tr>
<tr>
<td>DMUX</td>
<td>Type of frame.</td>
</tr>
<tr>
<td>FRAME</td>
<td>Indicates that a trace reference is to follow or that the trace buffer is to be displayed frame by frame.</td>
</tr>
<tr>
<td>INSTRUCTION</td>
<td>A function keyword indicating that data in the trace buffer is to be displayed in instruction format.</td>
</tr>
<tr>
<td>MARK</td>
<td>Equals 1 if trace was turned off before current frame or if emulation broke before current frame.</td>
</tr>
<tr>
<td>MOVE</td>
<td>Command keyword moves the trace buffer pointer.</td>
</tr>
<tr>
<td>NEWEST</td>
<td>Moves trace buffer pointer to bottom of buffer.</td>
</tr>
<tr>
<td>NOW</td>
<td>Indicates trace setting for beginning of next emulation.</td>
</tr>
<tr>
<td>OFF</td>
<td>Indicates trace turned off.</td>
</tr>
<tr>
<td>OLDEST</td>
<td>Moves trace buffer pointer to top of buffer.</td>
</tr>
<tr>
<td>ON</td>
<td>Indicates trace turned on.</td>
</tr>
<tr>
<td>PRINT</td>
<td>Command keyword calling for a display of one or more entries from the trace data buffer.</td>
</tr>
<tr>
<td>ODEPTH</td>
<td>Queue depth.</td>
</tr>
<tr>
<td>QSTS</td>
<td>2 queue status bits, QS1, QS0.</td>
</tr>
<tr>
<td>STS</td>
<td>3 status bits S2, S1, S0.</td>
</tr>
<tr>
<td>TRACE</td>
<td>Command keyword indicating that the mode of display for trace data is to be set.</td>
</tr>
</tbody>
</table>

User Names

The command language permits the programmer and operator to employ symbolic addressing through the use of user-generated tokens as opposed to system-generated tokens (keywords). The language permits four types of user names: symbols, module names, statement numbers, and macro names (see Chapter 8).

Symbols

A symbol is a sequence of contiguous alphanumeric characters, prefixed by a period (.), that references a location in a symbol table. The symbol has two uses. The referenced table location always contains a number; it may be an address of an instruction or variable in a program module, or it may be used directly as a numerical value. In the first case, the symbol is an alternative method of program addressing (symbolic as opposed to direct numeric addresses). In the second case, it provides a method for storing and retrieving data values symbolically into/from the table itself.
As an example, consider the symbol .BEGIN in module .MAINLOOP. The entire reference to this occurrence of .BEGIN is:

\[ \text{..MAINLOOP.BEGIN} \]

where

the double period (..) designates MAINLOOP as a \textit{module name} and

the single period (.) designates BEGIN as a \textit{symbol name}.

\textbf{Statement Numbers}

In the process of compiling a source module in DEBUG mode, the PL/M compiler generates a set of (source) \textit{statement numbers}, one for each source statement in the module. Each \textit{statement number} is linked to the absolute address of the first instruction generated by the PL/M compiler for the associated source statement in the source program. Each compiled program will contain a table of \textit{statement numbers} and absolute addresses. Items (addresses) in the table are referenced by entering the associated \textit{statement number}.

The form of reference is

\[ \text{module-name \# decimal-10} \]

where

\# is the ‘number’ sign; this designates the reference as a \textit{statement number} and

\textit{decimal-10} is the (source) \textit{statement number} (a numeric constant). The default suffix of \textit{decimal-10} is always decimal.

For example,

\[ \text{..MAINLOOP\#123} \]

\#123 is \textit{statement number} 123 in the source program MAINLOOP. This reference would obtain the \textit{address} of the first instruction generated by source statement 123 of module MAINLOOP.

\textit{Statement numbers} are an alternative to program addressing, as opposed to labels in the program.

\textbf{Special Tokens}

The command language contains two \textit{special token} sets that provide special functions: \textit{operators} and \textit{punctuation}.

\textbf{Operators}

<table>
<thead>
<tr>
<th>Type</th>
<th>Class Name</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>relational</td>
<td>\textit{rel-op}</td>
<td>=, &lt;, &gt;, &lt;=, &lt;, &gt;, &gt;=</td>
</tr>
<tr>
<td>plus</td>
<td>\textit{plus-op}</td>
<td>+, −, (binary and unary)</td>
</tr>
<tr>
<td>mult</td>
<td>\textit{multi-op}</td>
<td>*, /, MOD, MASK</td>
</tr>
<tr>
<td>logical</td>
<td>\textit{log-op}</td>
<td>NOT, AND, OR, XOR</td>
</tr>
</tbody>
</table>
Punctuation

<table>
<thead>
<tr>
<th>Type</th>
<th>Class Name</th>
<th>Punctuation Characters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Punctuation</td>
<td>punct-op</td>
<td>' , ; : ! $ CR LF SP % # .. !!</td>
</tr>
</tbody>
</table>

The use of punctuation characters are defined in those sections that define command formats.

**Entering Commands at the Console**

The ICE-88 emulator displays an asterisk prompt (*) at the left margin when it is ready to accept a command from the console.

Each command is entered as a *command line*, which consists of one or more *input lines*; the length of an input line is limited to the number of characters that one line of the console display can contain.

The ICE-88 emulator recognizes the carriage return as the terminator for a command line. If it is necessary to use more than one input line to enter a command, each intermediate input line should end with an ampersand (&). When the ICE-88 emulator encounters the ampersand, it suppresses the interpretation of the command that would occur on encountering the carriage return that follows. After the carriage return is executed, the ICE-88 emulator displays a double asterisk prompt (***) to acknowledge the continuation of the command line.

Tokens in the command are separated by blanks, unless the construct requires another form of separator. For example, tokens in a list are separated by commas; in this case, blanks (spaces) may be inserted for clarity but are not required.

Any input line may include comments. The comments are preceded by a semicolon (;), and must appear after any portion of the command that is in that input line; in other words, if the first character in an input line is a semicolon (;), the entire input line is comments. Characters in a comment are not interpreted by ICE-88 and are not stored internally except in a DEFINE MACRO command. The main use of comments is to document an emulation session while it is in progress.

Comments may not be continued from input line to input line. If an ampersand is used to continue a command line that also contains comments, the ampersand must come before the comment. An ampersand that is embedded in a comment is ignored by the ICE-88 emulator.

You can use ISIS-II editing capabilities to correct errors in the current input line. The line-editing characters are as follows:

<table>
<thead>
<tr>
<th>Characters</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>RUBOUT</td>
<td>Delete last character entered in input line. The deleted character is echoed immediately. The RUBOUT function can be repeated, deleting one character each time it is pressed.</td>
</tr>
<tr>
<td>CTRL X</td>
<td>Delete entire input line. (CTRL Z gives the same result.)</td>
</tr>
<tr>
<td>CTRL R</td>
<td>Display entire input line as entered so far. This is useful after a RUBOUT, to review which characters have been deleted.</td>
</tr>
</tbody>
</table>
ESC  Cancel entire command being entered.

CRTL P  Input next character literally.

Carriage Return  Terminate input line or command line.

Line Feed  Terminate input line

Once a line terminator (carriage return or line feed) has been entered, that line can no longer be edited.

The dollar sign ($) is ignored by the ICE-88 emulator in identifiers. You can use it as a separator when you want to combine two words into one token. For example, suppose you wanted to combine the two system groups DATA and STS into one symbol for your use. Instead of DATAANDSTS, you can use the $ character as a separator: DATA$AND$STS.
An expression is a formula that evaluates to a number. The formula can contain operands, operators, and parentheses. Expressions and operands appear in the ICE-88 emulator as command arguments to specify numeric values or boolean conditions. Depending on the command context, the resulting number is interpreted either as a numeric value or as a logical state (TRUE/FALSE). All expressions and operands represent one of the following:

- **Pointer:** a pair of 16-bit unsigned integers. One integer is called the base (b) and the other integer is called the displacement (d). This manual uses the notation \( b:d \) to denote a pointer with a base \( b \) and displacement \( d \). The ':' operator is a base-displacement integer connector (see table 5-3). Pointers may be used as memory addresses; then the 20-bit absolute address is \( 16^*b + d \).

- **Integer:** a single 16-bit unsigned integer treated modulo 65536. This is a special case of a pointer, with the base value equal to 0.

The ICE-88 emulator provides only unsigned-integer arithmetic on pointers and integers. The arithmetic operations are always applied separately to bases and displacements (i.e., integer arithmetic is always 16-bit). Signed arithmetic is not provided.

A few examples are all that is necessary to illustrate the concept of expressions.

1. The simplest form of an expression is a single value. That is, an expression that contains only one operand and no operators or parentheses.

   \[
   3 \\
   \text{FFFFH} \\
   127Q
   \]

2. The following expressions contain both operands and operators.

   \[
   2 + 3 \\
   1001110011101111Y - 101Y \\
   127/44 \\
   0100:00FFH
   \]

3. The following expressions contain operands, operators and parentheses.

   \[
   2 * (6 + 4) \\
   (127 + 44)/20
   \]

4. The use of symbols to reference numeric values represents a significant capability of the ICE-88 emulator. The following examples illustrate the use of symbols in expressions.

   \[
   .SYMBA OR .SYMBC \\
   (\!VAR1 + 10)/(\!VAR2 - .SYMBD)
   \]

This introduction to expressions is sufficient for the first reading of this manual or if the reader is familiar with previous ICE products. Therefore, you may skip the remainder of this chapter and read the remaining chapters using the examples.
Expressions

5-2

to gain further familiarity with the use of expressions in ICE-88 commands. The remainder of this chapter describes how expressions are constructed by representing the types of operands and operators that can be used, and provides the rules and some examples to explain how expressions are evaluated. The chapter also describes numeric and logical (boolean) command contexts, and gives a condensed syntax summary for expressions.

Operands

All operands are composed either pointers or unsigned integers. Operands can be specified by any of the following references:

- numeric constant
- masked constant
- keyword reference
- symbolic reference
- statement number reference
- memory reference
- typed memory reference
- port reference
- string
- (expression)

The following paragraphs define and explain each of the above operand types and formats.

Numeric Constants

A numeric constant produces a 16-bit integer value and is specified by a sequence composed of decimal digits and the letters “A” through “F” (hexadecimal digits), and optionally a suffix to specify explicitly the constant’s radix:

\[
\text{numeric-constant} = digit \ldots [\text{suffix}]
\]

Where:

\[
digit = 0 :: 1 :: 2 :: 3 :: 4 :: 5 :: 6 :: 7 :: 8 :: 9 :: A :: B :: C :: D :: E :: F
\]

And:

\[
\text{suffix} = H :: T :: O :: Q :: Y
\]

In the absence of a suffix, the default input radix for the current context is used. In most cases this is the radix set by the SUFFIX command; however, some commands may default their parameters to other radices. The allowed radices and their suffixes are: hexadecimal (“H”), decimal (“T”), octal (“O” or “Q”), and binary (“Y”). The radix determines which characters are valid in the constant. Numeric constants represent fixed unsigned integer values. The elements of numeric constants are summarized in table 5-1; examples are as follows:
Examples:

1001111010100111Y
1234
1974T
A2EH
177Q

Table 5-1. Elements of Numeric Constants

<table>
<thead>
<tr>
<th>Number Base</th>
<th>Valid Digits</th>
<th>Explicit Radix</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary (base 2)</td>
<td>0,1</td>
<td>Y</td>
<td>11001110101010111Y</td>
</tr>
<tr>
<td>Octal (base 8)</td>
<td>0-7</td>
<td>Q, O</td>
<td>4726Q</td>
</tr>
<tr>
<td>Decimal (base 10)</td>
<td>0-9</td>
<td>T</td>
<td>397T</td>
</tr>
<tr>
<td>Hexadecimal (base 16)</td>
<td>0-9, A-F</td>
<td>H</td>
<td>00FE3H</td>
</tr>
<tr>
<td>(multiple of 1024)</td>
<td>0-9</td>
<td></td>
<td>64</td>
</tr>
</tbody>
</table>

A numeric constant entered through the console with an explicit radix is interpreted accordingly. If it contains any digits that are invalid for that radix, an error results.

A numeric constant entered from the console without an explicit radix is interpreted according to the implicit radix that applies to the context. In most contexts, the implicit radix is initially hexadecimal (H); in these contexts, the implicit radix can be set to Y, Q, O, T or H by using the SUFFIX command.

Masked Constants

A masked constant is syntactically identical to a numeric constant except it may not contain the “T” suffix and must contain one or more “X” characters. Each “X” character represents a ‘don’t care’ digit (1, 3, or 4 bits depending upon whether the radix is binary, octal, or hexadecimal). The radix, either explicit or implicit (i.e., previously specified), must be binary, octal or hexadecimal. The following are examples of masked constants:

- 10X1X01Y (binary - 2 don’t care bits shown explicitly)
- 3X4Q (octal - 3 don’t care bits implicit because each octal numeral represents 3 bits; equivalent to 011XXX100Y)
- 6FX1H (hexadecimal - 4 don’t care bits, implicit because each hexadecimal numeral represents 4 bits; equivalent to 01101111XXX0001Y)

Keyword References

Keyword references are used to gain access to all of the system variables, including registers, status flags, input pins, and status information. When one is used in a command, the value returned is a 16-bit integer and is the current contents of the referenced object. Thus indirection through referenced variables is obtained. The values of system variables may be used in boolean conditions in control structures. A keyword reference may reference a value less than 16 bits.
If the keyword reference returns a value of less than 16 bits, the value is coerced to 16 bits by right-justifying it and filling the high-order bits with zeroes. (Refer to Chapter 4 for a listing and description of reference keywords.)

Examples:

\[
\begin{align*}
\text{SP} &= \text{SP} - 2 \\
\text{RAX} &= \text{display the contents of the Accumulator} \\
\text{TIMER} &= \text{display the contents of the TIMER register} \\
\text{AFL OR OFL} &= \text{“OR” Auxiliary-carry and Overflow flags}
\end{align*}
\]

**Symbolic References**

A symbolic reference points to an entry in the ICE-88 symbol table. Corresponding to each symbol table entry is a pointer value that represents an address or a constant. When a symbol reference is entered as an operand, its corresponding value is obtained from the referenced table location and used in the associated expression. A symbolic reference is specified in the following format:

\[
symbolic-ref \equiv [\text{module-ref}] \text{ symbol } ...
\]

Module references and symbols are both user-assigned. Module names are LOADED with a program—they cannot be assigned from the ICE-88 emulator. Most symbols are also LOADED. The module name identifies a particular symbol table that contains the symbols associated with a particular program module. The symbol table contains the symbols that are used by that program. Module names and symbols are composed of user names and identifying prefixes. User names are composed of character strings where each character may be an alphabetic character, digit, \(\text{“@”}\), underscore \(\text{“_”}\), or \(\text{“?”}\) with the exception that the first character in the string may not be a digit. The module name is prefixed by a double period \(\text{“..”}\):

\[
\text{module-ref} \equiv ..\text{module-name}
\]

A symbol is identified by a single period prefix \(\text{“.”}\):

\[
\text{symbol} \equiv .\text{symbol-name}
\]

Therefore the format for a symbolic reference can be shown as follows:

\[
symbolic-ref \equiv [..\text{module-name}] .\text{symbol-name}...
\]

If a module name is present, then only the referenced module’s symbol table is searched; otherwise all of the current symbol table is scanned for the referenced symbol. If more than one symbol is referenced, the symbol table is scanned for the occurrence of the first symbol in the list. Then the table is scanned for the first occurrence of the second symbol following the entry for the first symbol. This is repeated in sequence for all the symbols in the list. The value returned is the pointer containing the base and displacement address values for the entry specified by the symbolic reference.

The use of the symbol table provides you with considerable freedom in referencing and retrieving user variables. The ability to assign symbolic names to variables, procedures, and module names allows you to assign them names that can be associated with their functions and interrelations within the program. For
example, assume that the symbol .X represents a variable that is used in procedures PROCX, PROCY, and PROCZ of module MODABLE. Then the value of variable X in PROCY can be retrieved with the following symbolic reference:

```
..MODABLE.PROCY.X
```

whereas the value of variable X in PROCZ is obtained by:

```
.MODABLE..PROCZ.X
```

**Statement Number Reference**

A statement number reference points to the address of the first instruction generated by the compiler for the source statement specified by the associated statement number. Statement number information for each compiled program module is stored in its statement number table. Therefore program locations can be referenced symbolically via statement number. The LINK process can combine different modules, each with its own set of statement numbers. Therefore, a statement number reference may require a module reference in the same format as that used in a symbolic reference. A statement number reference uses the following format:

```
source-statement-ref = [..module-name] # statement-number
```

The statement number is an integer value that specifies the number of the source statement. If the statement number does not have an explicit suffix, the default suffix is decimal. If more than one program module is currently loaded into ICE-88 memory, a module reference is required to distinguish the reference from identical reference numbers in other modules. Examples are:

```
#45
..TEST1 #12FH
```

The value returned is a pointer value that is the absolute address of the first instruction generated by the compiler for the source statement referenced by the statement number.

**Memory References**

References to memory specify the type of reference as well as the memory location (address) required. A memory reference uses the following format:

```
memory-ref = reference-type address
```

<table>
<thead>
<tr>
<th>reference-type</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>1-byte integer value at location &quot;address.&quot;</td>
</tr>
<tr>
<td>WORD</td>
<td>2-byte integer value with low byte at &quot;address&quot; and high byte at &quot;address&quot; + 1.</td>
</tr>
<tr>
<td>SINTEGER</td>
<td>Same as BYTE.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Same as WORD.</td>
</tr>
<tr>
<td>POINTER</td>
<td>4-byte pointer value located at &quot;address&quot; through &quot;address&quot; + 3.</td>
</tr>
</tbody>
</table>
When changing memory or referencing it in an expression, BYTE is equivalent to SINTEGER and WORD to INTEGER. However, when displaying memory, the format of the display is either unsigned (BYTE, WORD) or signed (SINTEGER, INTEGER). (See Display Memory command, Chapter 7.)

Examples:

BYTE 1000H
BYT 0100:0000H
WORD 101
INTEGER .ABLE
POINTER CS:IP

Typed Memory Reference

A typed memory reference employs the symbols contained in the ICE-88 symbol table to obtain both location and type of memory reference. Each symbol has one of the following types of memory references or has no type:

- BYTE 1-byte integer value
- WORD 2-byte integer value
- SINTEGER Same as BYTE
- INTEGER Same as WORD
- POINTER 4-byte pointer value

If a symbol has a memory reference type, the symbol represents a memory reference. If the symbol has no memory reference type, the symbol represents a label, procedure name, or a constant. If the source language translator generates type information in the object file, then the type values are loaded with the symbols in the ICE-88 emulator. The user may also specify memory reference type when defining symbols or when using the ICE-88 TYPE command.

A typed memory reference is executed with the following format:

\[ \text{typed-mem-ref} \equiv [!\text{module-name}] \text{symbol-name} \ldots \]

Example:

Assume symbol table ..SAM contains:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Type</th>
<th>Base Value</th>
<th>Displacement Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>.X</td>
<td>BYTE</td>
<td>100H</td>
<td>0</td>
</tr>
<tr>
<td>.Y</td>
<td>WORD</td>
<td>100H</td>
<td>0</td>
</tr>
<tr>
<td>.Z</td>
<td>POINTER</td>
<td>100H</td>
<td>0</td>
</tr>
</tbody>
</table>

Also assume:

<table>
<thead>
<tr>
<th>Memory Location</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000H</td>
<td>21</td>
</tr>
<tr>
<td>1001H</td>
<td>43</td>
</tr>
<tr>
<td>1002H</td>
<td>65</td>
</tr>
<tr>
<td>1003H</td>
<td>87</td>
</tr>
</tbody>
</table>

Therefore:

!!SAM !X = 21
!!SAM !Y = 4321
!!SAM !Z = 8765:4321
Port References

The ICE-88 emulator supports a maximum of 64K 8-bit or 32K 16-bit I/O ports. These ports are referenced in the following format:

\[
port-ref = \text{PORT address} :: \text{WPORT address}
\]

PORT references an 8-bit I/O port at location "address." WPORT references a 16-bit I/O port at location "address." The value of "address" must be an integer. The port is read or written immediately when referenced.

Examples:

- PORT 123
- PORT RDX
- WPOR T 1FFH

String Constants

Any one of the ASCII characters (ASCII codes 00H through 7FH) can be entered as a string constant by enclosing the character in single quotes. The operand value of a string constant is a 16-bit integer with the high-order bits set to 0, and the 7-bit ASCII code in the low-order seven bits. For example, the string constant "A" has the value 0000000001000001 (0041H).

In data communications usage, an ASCII-coded character consists of seven low-order data bits (bits 0-6), and a parity bit (bit 7). Thus another way to describe the operand value of an ASCII string constant is as a two-byte integer; the high byte is all zeros and the low byte contains the 8-bit ASCII value with the parity bit set to 0.

Table 5-2 gives the printing ASCII characters with their corresponding hexadecimal codes (codes 20H through 7EH). Note that some console keyboards output upper case ASCII characters only, or lack keys for some of the non-printing ASCII codes.
Table 5-2. ASCII Printing Characters and CODES (20H—7EH)

<table>
<thead>
<tr>
<th>Character</th>
<th>Hex Code</th>
<th>Character</th>
<th>Hex Code</th>
<th>Character</th>
<th>Hex Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Space (SP)</td>
<td>20</td>
<td><code>@</code></td>
<td>40</td>
<td><code>a</code></td>
<td>60</td>
</tr>
<tr>
<td>!</td>
<td>21</td>
<td><code>A</code></td>
<td>41</td>
<td><code>b</code></td>
<td>61</td>
</tr>
<tr>
<td>'</td>
<td>22</td>
<td><code>B</code></td>
<td>42</td>
<td><code>c</code></td>
<td>62</td>
</tr>
<tr>
<td>#</td>
<td>23</td>
<td><code>C</code></td>
<td>43</td>
<td><code>d</code></td>
<td>63</td>
</tr>
<tr>
<td>$</td>
<td>24</td>
<td><code>D</code></td>
<td>44</td>
<td><code>e</code></td>
<td>64</td>
</tr>
<tr>
<td>%</td>
<td>25</td>
<td><code>E</code></td>
<td>45</td>
<td><code>f</code></td>
<td>65</td>
</tr>
<tr>
<td>&amp;</td>
<td>26</td>
<td><code>F</code></td>
<td>46</td>
<td><code>g</code></td>
<td>66</td>
</tr>
<tr>
<td>'</td>
<td>27</td>
<td><code>G</code></td>
<td>47</td>
<td><code>h</code></td>
<td>67</td>
</tr>
<tr>
<td>(</td>
<td>28</td>
<td><code>H</code></td>
<td>48</td>
<td><code>i</code></td>
<td>68</td>
</tr>
<tr>
<td>)</td>
<td>29</td>
<td><code>I</code></td>
<td>49</td>
<td><code>j</code></td>
<td>69</td>
</tr>
<tr>
<td>+</td>
<td>2A</td>
<td><code>J</code></td>
<td>4A</td>
<td><code>k</code></td>
<td>6A</td>
</tr>
<tr>
<td>-</td>
<td>2B</td>
<td><code>K</code></td>
<td>4B</td>
<td><code>l</code></td>
<td>6B</td>
</tr>
<tr>
<td>.</td>
<td>2C</td>
<td><code>L</code></td>
<td>4C</td>
<td><code>m</code></td>
<td>6C</td>
</tr>
<tr>
<td>/</td>
<td>2D</td>
<td><code>M</code></td>
<td>4D</td>
<td><code>n</code></td>
<td>6D</td>
</tr>
<tr>
<td>0</td>
<td>30</td>
<td><code>P</code></td>
<td>50</td>
<td><code>p</code></td>
<td>70</td>
</tr>
<tr>
<td>1</td>
<td>31</td>
<td><code>Q</code></td>
<td>51</td>
<td><code>q</code></td>
<td>71</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td><code>R</code></td>
<td>52</td>
<td><code>r</code></td>
<td>72</td>
</tr>
<tr>
<td>3</td>
<td>33</td>
<td><code>S</code></td>
<td>53</td>
<td><code>s</code></td>
<td>73</td>
</tr>
<tr>
<td>4</td>
<td>34</td>
<td><code>T</code></td>
<td>54</td>
<td><code>t</code></td>
<td>74</td>
</tr>
<tr>
<td>5</td>
<td>35</td>
<td><code>U</code></td>
<td>55</td>
<td><code>u</code></td>
<td>75</td>
</tr>
<tr>
<td>6</td>
<td>36</td>
<td><code>V</code></td>
<td>56</td>
<td><code>v</code></td>
<td>76</td>
</tr>
<tr>
<td>7</td>
<td>37</td>
<td><code>W</code></td>
<td>57</td>
<td><code>w</code></td>
<td>77</td>
</tr>
<tr>
<td>8</td>
<td>38</td>
<td><code>X</code></td>
<td>58</td>
<td><code>x</code></td>
<td>78</td>
</tr>
<tr>
<td>9</td>
<td>39</td>
<td><code>Y</code></td>
<td>59</td>
<td><code>y</code></td>
<td>79</td>
</tr>
<tr>
<td>;</td>
<td>3A</td>
<td><code>Z</code></td>
<td>5A</td>
<td><code>z</code></td>
<td>7A</td>
</tr>
<tr>
<td>&lt;</td>
<td>3B</td>
<td><code>(</code></td>
<td>5B</td>
<td><code>(</code></td>
<td>7B</td>
</tr>
<tr>
<td>=</td>
<td>3C</td>
<td><code>)</code></td>
<td>5C</td>
<td><code>)</code></td>
<td>7C</td>
</tr>
<tr>
<td>&gt;</td>
<td>3D</td>
<td><code>^ (↑)</code></td>
<td>5D</td>
<td><code>^ (↑)</code></td>
<td>7D</td>
</tr>
<tr>
<td>?</td>
<td>3E</td>
<td><code>– (–)</code></td>
<td>5E</td>
<td><code>– (–)</code></td>
<td>7E</td>
</tr>
</tbody>
</table>
Parenthesized Expressions

(exp): an operand whose value is the value of the parenthesized expression, e.g.,
(1+2+3) = 6 (operand value).

Operators

An expression can contain any combination of unary and binary operators. Table
5-3 describes all the operators available to the ICE-88 emulator. The operators
are ranked in order of precedence from highest (1) to lowest (10). Other things
being equal, the operator with the highest precedence is evaluated first. The
operators are shown in the table as they are to be entered in expressions. The
class content-operators has too many details to fit the table; see table 5-5. The
table identifies each operator as unary or binary. A unary operator takes one
operand, and a binary operator takes two operands.

Classes of Operators

For discussion, the operators are classed as shown in table 5-4. Table 5-6
specifies the arithmetic and logical semantic rules for operators.

Arithmetic Operators

The ICE scanner distinguishes unary "+" and "-" from binary "+" and "-" by
context. Unary "+" is superfluous, since it is a no-operation.

A unary "-" applied to an integer means "2's complement modulo 65536." In
other words, (−N) evaluates to (65536 − N). As the ICE-88 emulator uses only
unsigned arithmetic, unary "-" does not apply to pointers. The unary "-" is also
used in the MOVE and PRINT commands (see MOVE and PRINT commands in
Chapter 6).

Binary "+" applies to pointer and integer values only and results in the arithmetic
sum of its two operands. In the case of the sum of two integers, the result is
treated modulo 65536 (any high-order bits after the sixteenth bit are dropped). In
the case of the sum of a pointer and an integer, the displacement value of the
pointer is summed with the integer modulo 65536 and the base value of the
pointer is unchanged.

Binary "-" applies to pointer and integer values only and results in the arithmetic
difference of the two operands. In the case of the difference of two integers, the
result is the 2's complement difference of the two integers; this result is also
treated modulo 65536, so that a "negative" result (−N) ends up as (65536 − N). An
integer may also be subtracted from a pointer. In this case, the result is the 2's
complement difference of the pointer displacement and the integer modulo 65536
and the base value of the pointer remains unchanged. The "-" can be used to ob-
tain the arithmetic difference of two pointers but only if they have the same base
value. In this case the result is the 2's complement difference of the displacements
modulo 65536 and the resulting base value is set equal to zero. An
error occurs if the base values of the pointers are not equal.
The operators "*", "/", "MOD", and "MASK" can be applied only to integer operands and return only integer results.

Binary "*" results in the multiplication of two integer operands, truncated to the low-order 16 bits.

Binary "/" causes the first integer operand to be divided by the second. The result is the integer quotient; the remainder, if any, is lost. Thus, (5/3) evaluates to (1).

Binary "MOD" returns the remainder after integer division as an integer result, and the quotient part of the division is lost. Thus, (5 MOD 3) evaluates to (2), the remainder of (5/3).

Binary "MASK" performs a bitwise logical AND on two integer operands. If either corresponding bit is a 1, or if both are 1's, the result has 1 in that bit; if both are 0, the result has 0 in that bit. MASK is identical to the boolean "AND" operator, except that MASK has higher precedence.

::, SEGMENT, OFFSET have the highest precedence of the arithmetic operators. Binary "*", "/", and "MOD" have equal precedence, lower than unary "-". Binary "+" and "-" have equal precedence, lower than "*", "/", and "MOD". "MASK" has lowest precedence of the arithmetic operators (see table 5-3).

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operator</th>
<th>Unary</th>
<th>Binary</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>: OFFSET</td>
<td>u</td>
<td>u</td>
<td>Base, displacement integer connector for a pointer (e.g., 1234:5678 or CS:IP).</td>
</tr>
<tr>
<td>1</td>
<td>OFFSET</td>
<td>u</td>
<td></td>
<td>Designates integer value that is the displacement of a pointer (e.g., OFFSET 1234:5678 is 5678).</td>
</tr>
<tr>
<td>1</td>
<td>SEGMENT</td>
<td>u</td>
<td></td>
<td>Designates integer value that is the base of a pointer (e.g., SEGMENT 1234:5678 is 1234).</td>
</tr>
<tr>
<td>2</td>
<td>+</td>
<td>u</td>
<td></td>
<td>Unary plus.</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>u</td>
<td></td>
<td>Unary minus, (-N) means (65536-N), the 2's complement of N, modulo 2^16.</td>
</tr>
<tr>
<td>3</td>
<td>*</td>
<td>b</td>
<td></td>
<td>Integer multiplication.</td>
</tr>
<tr>
<td>3</td>
<td>/ MOD</td>
<td>b</td>
<td></td>
<td>Integer division. The result is the integer quotient; the remainder (if any) is lost.</td>
</tr>
<tr>
<td>4</td>
<td>+</td>
<td>b</td>
<td></td>
<td>Addition.</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>b</td>
<td></td>
<td>Subtraction.</td>
</tr>
<tr>
<td>5</td>
<td>MASK</td>
<td>b</td>
<td></td>
<td>Bitwise AND. Higher precedence than identical operation AND (see below).</td>
</tr>
<tr>
<td>6</td>
<td>content-operator</td>
<td>u</td>
<td></td>
<td>Treats operand as memory or port address, returns the content of that address.</td>
</tr>
<tr>
<td>7</td>
<td>=</td>
<td>b</td>
<td></td>
<td>Is equal to. Result is either TRUE (FFFFH) or FALSE (0).</td>
</tr>
</tbody>
</table>
### Table 5-3. ICE™ Operators (Cont’d.)

<table>
<thead>
<tr>
<th>Precedence&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Operator</th>
<th>Unary Binary&lt;sup&gt;1&lt;/sup&gt;</th>
<th>Effect&lt;sup&gt;1&lt;/sup&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>&gt;</td>
<td>b</td>
<td>Is greater than. Result is TRUE or FALSE.</td>
</tr>
<tr>
<td></td>
<td>&lt;</td>
<td>b</td>
<td>Is less than. Result is TRUE or FALSE.</td>
</tr>
<tr>
<td></td>
<td>&lt;&gt;</td>
<td>b</td>
<td>Is not equal to. Result is TRUE or FALSE.</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
<td>b</td>
<td>Is greater than or equal to. Result is TRUE or FALSE.</td>
</tr>
<tr>
<td></td>
<td>&lt;=</td>
<td>b</td>
<td>Is less than or equal to. Result is TRUE or FALSE.</td>
</tr>
<tr>
<td>8</td>
<td>NOT</td>
<td>u</td>
<td>Unary Logical (1’s) complement. Bitwise 1 becomes 0, 0 becomes 1; TRUE becomes FALSE, FALSE becomes TRUE.</td>
</tr>
<tr>
<td>9</td>
<td>AND</td>
<td>b</td>
<td>Bitwise AND. If both corresponding bits are 1’s, result has 1 in that bit; else 0. TRUE AND TRUE yields a TRUE result; any other combination is FALSE.</td>
</tr>
<tr>
<td>10</td>
<td>OR</td>
<td>b</td>
<td>Bitwise inclusive OR. If either corresponding bit is a 1, result has 1 in that bit; else 0. If either operand is TRUE, result is TRUE; else FALSE.</td>
</tr>
<tr>
<td></td>
<td>XOR</td>
<td>b</td>
<td>Bitwise exclusive OR. If corresponding bits are different, result has 1 in that bit; else 0. If one operand is TRUE and the other is FALSE, result is TRUE; if both are TRUE or both are FALSE, result is FALSE.</td>
</tr>
</tbody>
</table>

**Notes:**

1. <sup>1</sup> = highest precedence (evaluated first), 10 = lowest precedence.
2. <sup>1</sup>u = unary, b = binary.
3. Refer to text for additional details.

4. `content-operator` is one of the tokens: BYTE, WORD, SINTEGER, INTEGER, POINTER, PORT, or WPORT.

### Table 5-4. Classes of Operators

<table>
<thead>
<tr>
<th>Class</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>(Numeric)</td>
<td></td>
</tr>
<tr>
<td>Arithmetic</td>
<td>+, −, SEGMENT, OFFSET,</td>
</tr>
<tr>
<td>unary</td>
<td>* , /, MOD, +, −, MASK, :</td>
</tr>
<tr>
<td>binary</td>
<td></td>
</tr>
<tr>
<td>Content</td>
<td></td>
</tr>
<tr>
<td>unary</td>
<td></td>
</tr>
<tr>
<td>binary</td>
<td>content-operators</td>
</tr>
<tr>
<td>(Boolean)</td>
<td></td>
</tr>
<tr>
<td>Relational</td>
<td>=, &gt;, &lt;, &lt;=, &gt;=, &lt;&gt;</td>
</tr>
<tr>
<td>binary</td>
<td></td>
</tr>
<tr>
<td>Logical</td>
<td>NOT</td>
</tr>
<tr>
<td>unary</td>
<td>AND, OR, XOR</td>
</tr>
<tr>
<td>binary</td>
<td></td>
</tr>
<tr>
<td>Unary</td>
<td>+, −, SEGMENT, OFFSET,</td>
</tr>
<tr>
<td>content-operators</td>
<td></td>
</tr>
<tr>
<td>Binary</td>
<td>* , /, MOD, +, −, MASK, :</td>
</tr>
<tr>
<td>relational-operators</td>
<td></td>
</tr>
<tr>
<td>AND, OR, XOR</td>
<td></td>
</tr>
</tbody>
</table>
Content Operators

Content operators are keywords that refer to the contents of memory locations and I/O ports. In expressions, they function as unary operators with precedence immediately below “MASK.” Table 5-5 summarizes the content operators for the ICE-88 emulator.

Table 5-5. Content Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Content Returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>1-byte integer value from the addressed location in user memory.</td>
</tr>
<tr>
<td>WORD</td>
<td>2-byte integer value from the addressed location in user memory.</td>
</tr>
<tr>
<td>SINTEGER</td>
<td>Same as BYTE.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>Same as WORD.</td>
</tr>
<tr>
<td>POINTER</td>
<td>4-byte pointer value from the addressed location in user memory.</td>
</tr>
<tr>
<td>PORT</td>
<td>1-byte value from addressed 8-bit I/O port.</td>
</tr>
<tr>
<td>WPORT</td>
<td>2-byte value from addressed 16-bit I/O port.</td>
</tr>
</tbody>
</table>

To be used in an expression, a content operator must precede a single operand that can be interpreted as a valid address. A partition of addresses (using a keyword such as TO or LENGTH) cannot be used in an expression. Furthermore, the address given must be accessible (not GUARDED) if it uses the memory map (see MAP commands in Chapter 7).

Relational Operators

A relational operator calls for a comparison of the values of its two operands. The six possible relational operations are shown in table 5-4. Each comparison is either true when the expression is evaluated, or it is false. The result is correspondingly TRUE (FFFFH) or FALSE (0).

Logical Operators

The “NOT” logical operator results in a 1’s complement of an operand; a 16-bit operand is assumed. The following are examples of “NOT” logical operations:

<table>
<thead>
<tr>
<th>Operand</th>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>NOT</td>
<td>FFFFH</td>
</tr>
<tr>
<td>1</td>
<td>NOT</td>
<td>FFFEH</td>
</tr>
<tr>
<td>11110110Y</td>
<td>NOT</td>
<td>1111111100001001Y</td>
</tr>
<tr>
<td>FFFFH</td>
<td>NOT</td>
<td>0</td>
</tr>
<tr>
<td>FFFEH</td>
<td>NOT</td>
<td>1</td>
</tr>
</tbody>
</table>
ANDing two operands results in the following values depending upon bit pair values:

<table>
<thead>
<tr>
<th>bit 1</th>
<th>bit 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Examples:

<table>
<thead>
<tr>
<th>Logical Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 AND 0</td>
<td>0</td>
</tr>
<tr>
<td>1010Y AND 1001Y</td>
<td>1000Y</td>
</tr>
<tr>
<td>FFFFH AND 0</td>
<td>0</td>
</tr>
<tr>
<td>FFFFH AND FFFFH</td>
<td>FFFFH</td>
</tr>
<tr>
<td>1 AND 0</td>
<td>0</td>
</tr>
</tbody>
</table>

ORing two operands results in the following values depending upon bit pair values:

<table>
<thead>
<tr>
<th>bit 1</th>
<th>bit 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Examples:

<table>
<thead>
<tr>
<th>Logical Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 OR 0</td>
<td>0</td>
</tr>
<tr>
<td>1 OR 0</td>
<td>1</td>
</tr>
<tr>
<td>1010Y OR 1001Y</td>
<td>1011Y</td>
</tr>
<tr>
<td>FFFFH OR 0</td>
<td>FFFFH</td>
</tr>
<tr>
<td>FFFFH OR FFFFH</td>
<td>FFFFH</td>
</tr>
</tbody>
</table>

The result of an “XOR” operation is as follows:

<table>
<thead>
<tr>
<th>bit 1</th>
<th>bit 2</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Examples:

<table>
<thead>
<tr>
<th>Logical Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 XOR 0</td>
<td>0</td>
</tr>
<tr>
<td>1 XOR 0</td>
<td>1</td>
</tr>
<tr>
<td>1010Y XOR 1001Y</td>
<td>11Y</td>
</tr>
<tr>
<td>FFFFH XOR 0</td>
<td>FFFFH</td>
</tr>
<tr>
<td>FFFFH XOR FFFFH</td>
<td>0</td>
</tr>
</tbody>
</table>
Arithmetic and Logical Semantic Rules

Table 5-6 provides a summary of the semantic rules that apply to arithmetic and logical operations. The table specifies the function performed by each type of arithmetic and logical operation, the input required, and the result of the operation (output).

Table 5-6. Arithmetic and Logical Semantic Rules

<table>
<thead>
<tr>
<th>Operation</th>
<th>Operators</th>
<th>Input</th>
<th>Output</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>logical</td>
<td>AND, OR, XOR, MASK</td>
<td>2 integers</td>
<td>integer</td>
<td>Bitwise conjunction or disjunction of integers.</td>
</tr>
<tr>
<td>not</td>
<td>NOT</td>
<td>integer</td>
<td>integer</td>
<td>One's complement of an integer.</td>
</tr>
<tr>
<td>relational</td>
<td>&lt;, &gt;, &lt;=, =&gt;, =</td>
<td>(1) 2 pointers, same base</td>
<td>integer</td>
<td>Logical test of relational expression. If the displacement integer values satisfy relational operation (true), then the output integer value = FFFFH. If the displacement integer values do not satisfy the relational operation (false), the output integer = 0. If the base values of the input pointers are not equal, an error occurs.</td>
</tr>
<tr>
<td>arithmetic</td>
<td>*, /, MOD</td>
<td>2 integers</td>
<td>integer</td>
<td>Unsigned product (*), quotient(/), or remainder(MOD) of two integers.</td>
</tr>
<tr>
<td>memory-content</td>
<td>BYTE, WORD, INTEGER, SINTEGER</td>
<td>pointer or integer</td>
<td>integer</td>
<td>Fetches content of memory location addressed by input value.</td>
</tr>
<tr>
<td>memory-content</td>
<td>POINTER</td>
<td>pointer or integer</td>
<td>pointer</td>
<td>Fetches content of memory location addressed by input value.</td>
</tr>
<tr>
<td>I/O-content</td>
<td>PORT, WPORT</td>
<td>integer</td>
<td>integer</td>
<td>Fetches content of I/O port (8-bit or 16-bit) addressed by input value.</td>
</tr>
<tr>
<td>+(binary)</td>
<td>+</td>
<td>(1) pointer, integer</td>
<td>integer</td>
<td>Sum of the displacement values, same base as the pointer. Sum of the integers.</td>
</tr>
<tr>
<td>(binary)</td>
<td>-</td>
<td>(1) pointer with base values</td>
<td>integer</td>
<td>Two's complement difference of displacement values. Error occurs if base values are unequal.</td>
</tr>
<tr>
<td>-(binary)</td>
<td>-</td>
<td>(2) pointer &amp; integer</td>
<td>pointer</td>
<td>Two’s complement difference of pointer displacement value and integer input, same base as the pointer.</td>
</tr>
<tr>
<td>-(unary)</td>
<td>-</td>
<td>integer</td>
<td>integer</td>
<td>Two’s complement of the input integer.</td>
</tr>
<tr>
<td>+(unary)</td>
<td>+</td>
<td>All types</td>
<td>same</td>
<td>No change.</td>
</tr>
<tr>
<td>override base/</td>
<td>:</td>
<td>(1) integer &amp; pointer</td>
<td>pointer</td>
<td>Replaces current base value of pointer with input integer value.</td>
</tr>
<tr>
<td>construct pointer</td>
<td></td>
<td>(2) 2 integers</td>
<td>pointer</td>
<td>Constructs new pointer with base value set to first input integer value and displacement set to second integer.</td>
</tr>
<tr>
<td>offset</td>
<td>OFFSET</td>
<td>pointer</td>
<td>integer</td>
<td>Generates integer whose value is the displacement value of the input pointer.</td>
</tr>
<tr>
<td>segment</td>
<td>SEGMENT</td>
<td>pointer</td>
<td>integer</td>
<td>Generates integer whose value is the base value of the input pointer.</td>
</tr>
</tbody>
</table>
How Expressions are Evaluated

This section provides a simple conceptual model of how the ICE-88 emulator evaluates an expression. The model involves a loop that scans the expression iteratively (figure 5-1). The loop terminates in either of two ways:

- When the expression resolves to a single numeric value.
- When a syntax error (or other error) occurs.

The ICE-88 emulator goes through the scan loop once for each operator in the expression. On each scan, the operator (unary or binary) that must be applied next is identified.

The next operator is always:

- the leftmost operator
- with highest precedence (table 5-3)
- that is enclosed in the innermost pair of parentheses.

If this next operator is unary, and has a numeric operand, the operation is performed on the operand to produce a numeric result. If the next operator is binary, and has a pair of operands, the operation is performed on the pair of operands to produce a numeric result. If the next operator does not have the required number of numeric operands, a syntax error results, and the loop terminates.

A pair of parentheses is “cleared” when it contains just a single numeric value; that is:

\[(\text{numeric-value}) \rightarrow \text{numeric-value}\]

After performing any operation, the numeric result becomes an operand for the next scan. Parentheses are cleared before the next scan begins.

“Case Studies” in Evaluating Expressions

Here are some representative cases of expressions showing how they are evaluated by the ICE-88 emulator. In some examples, the steps in evaluation are shown, but most show just the overall result. Table 5-7 summarizes the cases. The EVALUATE (EVA) command used in these examples performs the evaluation and displays the result in the four numeric radices (Y, Q, T, and H), plus the ASCII printing equivalent (if any) in single quotes. The examples in this section assume the initial conditions shown in table 5-8. This table also describes the special notation used in some of the examples. The examples also assume SUF-FIX = T; that is, any number without an explicit radix is decimal.
Figure 5-1. A Simple Model of Evaluation
Table 5-7. Conditions and Notation for Examples

<table>
<thead>
<tr>
<th>Conditions</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>All memory locations are accessible (none are GUARDED).</td>
<td></td>
</tr>
<tr>
<td>SUFFIX = T (implicit radix is decimal).</td>
<td></td>
</tr>
<tr>
<td>IP = 1000H</td>
<td></td>
</tr>
<tr>
<td>DEFINE .AA = 2000H</td>
<td></td>
</tr>
<tr>
<td>DEFINE .BB = FFFFH</td>
<td></td>
</tr>
<tr>
<td>BYTE 1000H = 3EH</td>
<td></td>
</tr>
<tr>
<td>BYTE 2000H = 23H</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Notation</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;&gt; has higher precedence than.</td>
<td></td>
</tr>
<tr>
<td>&gt;&gt;= has higher or equal precedence.</td>
<td></td>
</tr>
<tr>
<td>u1,u2,... unary operators</td>
<td></td>
</tr>
<tr>
<td>b1,b2,... binary operators</td>
<td></td>
</tr>
</tbody>
</table>

Case 1: EVALUATE operand

An expression can be composed of just a single operand, requiring at most a lookup to produce a numeric result.

Examples:

EVA 10 (input)
1010Y 12Q 10T AH " (display)

EVA IP (input)
10000000000000Y 1000Q 4096T 1000H " (display)

EVA .AA (input)
10000000000000Y 2000Q 8192T 2000H " (display)

EVA 1234H:5678H (input)
1234:5678H 179BBH (display)

Case 2: EVALUATE unary-operator operand

A unary operator with a single primary operand evaluates to a number.

Examples:

EVA -2 (input)
1111111111111110Y 177760 65534T FFFEH 't' (display)

EVA BYTE .AA (input)
100011Y 43Q 35T 23H '#' (display)

EVA NOT IP (input)
1101111111111111Y 167777Q 61439T EFFFH '0' (display)

Case 3: EVALUATE operand binary-operator operand

The binary operator is applied to its two operands to produce a numeric result.

Examples:

EVA 10 + 20 (input)
11110Y 36Q 30T 1EH "

EVA .AA > 10 (input)
111111111111111Y 177777Q 65535T FFFFH " (display)
Table 5-8. Representative Cases of Expressions

<table>
<thead>
<tr>
<th>Case</th>
<th>Expression</th>
<th>Precedence</th>
<th>Result of Lookup Plus One Scan</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>operand</td>
<td>None</td>
<td>number</td>
</tr>
<tr>
<td>2</td>
<td>unary-operator operand</td>
<td>Any</td>
<td>number</td>
</tr>
<tr>
<td>3</td>
<td>operand binary-operator operand</td>
<td>Any</td>
<td>number</td>
</tr>
<tr>
<td>4</td>
<td>operand b1 operand b2 operand</td>
<td>b1 &gt;= b2</td>
<td>number b2 number (case 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b2 &gt;= b1</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td>5</td>
<td>operand b1 (operand b2 operand)</td>
<td>b2 &gt;= b1</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b1 &gt;= b2</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td>6</td>
<td>u1 operand b1 operand</td>
<td>u1 &gt;= b1</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b1 &gt;= u1</td>
<td>u1 number (case 2)</td>
</tr>
<tr>
<td>7</td>
<td>operand b1 u1 operand</td>
<td>u1 &gt;= b1</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b1 &gt;= u1</td>
<td>ERROR (See case 8)</td>
</tr>
<tr>
<td>8</td>
<td>operand b1 (u1 operand)</td>
<td>u1 &gt;= b1</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b1 &gt;= u1</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td>9</td>
<td>u1 u2 operand</td>
<td>u2 &gt;= u1</td>
<td>u1 number (case 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>u1 &gt;= u2</td>
<td>ERROR (See case 10)</td>
</tr>
<tr>
<td>10</td>
<td>u1 (u2 operand)</td>
<td>u2 &gt;= u1</td>
<td>u1 number (case 2)</td>
</tr>
</tbody>
</table>

EVA .AA OR IP (input)
11000000000000Y 3000Q 12288T 3000H '0' (display)

EVA 0100H:0010H + .AA (input)
0100:2010H 03010H (display)

Case 4: EVALUATE operand b1 operand b2 operand

The binary operator with the highest precedence is evaluated first. If they have equal precedence, b1 (the leftmost) is evaluated first.

A. b1 >>= b2

Examples:

EVA 10 + .AA - IP (input)
1000000001010Y 10012Q 4106T 100AH ' ' (display)

EVA 10 * .AA - IP (input)
11000000000000Y 3000Q 12288T 3000H '0' (display)

EVA IP = .AA OR .BB (input)
111111111111111Y 177777Q 65535T FFFFH ' '

EVA 1 + 2 - 3 (input)
0Y 0Q 0T 0H ' ' (display)

EVA 3 * 2 + 1 (input)
111Y 7Q 7T 7H ' ' (display)
B. \( b_2 \gg b_1 \)

Examples:

\[
\begin{align*}
EVA & \ 2 + 3 \ * \ 4 \ (input) \\
& \ 1110Y \ 16Q \ 14T \ \text{EH} \quad \text{(display)}
\end{align*}
\]

\[
\begin{align*}
EVA \ .BB \ OR \ .AA \ AND \ IP \ (input) \\
& \ 1111111111111111Y \ 177777Q \ 65535T \ \text{FFFFH} \quad \text{(display)}
\end{align*}
\]

\[
\begin{align*}
EVA & \ 1 OR 2 AND 3 \ (input) \\
& \ 11Y \ 3Q \ 3T \ 3H \quad \text{(display)}
\end{align*}
\]

Case 4 also fits expressions of any length that use only binary operators. Here is an example showing the steps in the evaluation.

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Expression .BB OR IP = .AA AND AFAFH XOR .AA MOD 277</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Lookup FFFFH OR 1000H = 2000H AND AFAFH XOR 2000H MOD 277</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>MOD FFFFH OR 1000H = 2000H AND AFAFH XOR 9FH</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>= FFFFH OR 0 AND AFAFH XOR 9FH</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>AND FFFFH OR 0 XOR 9FH</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>OR FFFFH XOR 9FH</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>XOR FF60H</td>
<td></td>
</tr>
</tbody>
</table>

More examples:

\[
\begin{align*}
EVA & \ 2 \ XOR \ 3 \ MASK \ 41 \ MOD \ 33 \\
& \ 10Y \ 2Q \ 2T \ 2H \quad \text{''}
\end{align*}
\]

\[
\begin{align*}
EVA & \ 2 \ * \ 3 \ + \ 5 \ / \ 3 \ + \ 6 \\
& \ 1101Y \ 15Q \ 13T \ \text{DH} \quad \text{''}
\end{align*}
\]

\[
\begin{align*}
EVA & \ 2 \ + \ 3 \ * \ 5 \ + \ 7 \\
& \ 11000Y \ 30Q \ 24T \ 18H \quad \text{''}
\end{align*}
\]

Case 5: EVALUATE operand \( b_2 \) \((\text{operand} \ b_2 \ \text{operand})\)

Binary operator \( b_2 \) is evaluated first, even if it has lower precedence than \( b_1 \). Use parentheses when \( b_2 \) must be evaluated before \( b_1 \).

Examples:

\[
\begin{align*}
EVA & \ 2 \ * \ (3 \ + \ 5) \\
& \ 1000Y \ 20Q \ 16T \ 10H \quad \text{''}
\end{align*}
\]

\[
\begin{align*}
EVA \ .BB \ / \ (.AA \ \text{MASK} \ AFAFH) \\
& \ 111Y \ 7Q \ 7T \ 7H \quad \text{''}
\end{align*}
\]

This case can be generalized to include any number of binary operators and any arrangement of parentheses. For example:

<table>
<thead>
<tr>
<th>Step</th>
<th>Operation</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Expression 10 * (44 + (17 * 15 - 6) / 7)</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2nd* 10 * (44 + (255 - 6) / 7)</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>- 10 * (44 + (249) / 7)</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Clear () 10 * (44 + 249 / 7)</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>/ 10 * (44 + 35)</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>+ 10 * (79)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Clear () 10 * 79</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>1st* 790</td>
<td></td>
</tr>
</tbody>
</table>
Case 6: EVALUATE u1 operand b1 operand

Precedence decides which operator is evaluated first.

A. u1 >> b1

Examples:

EVA -10 + 22
1100Y 14Q 12T CH "

EVA BYTE .AA OR .BB
1111111111111111Y 177777Q 65535T FFFFH "

EVA NOT .BB AND AFAFH
0Y 0Q 0T 0H "

B. b1 >> u1

Examples:

EVA BYTE .AA - 1000H
111110Y 76Q 62T 3EH ‘>’

EVA NOT .BB/23
1111010111011101Y 172336Q 62686T F4DEH ‘T’

Case 7: EVALUATE operand b1 u1 operand

The unary operator must have higher precedence than the binary operator.

A. u1 >> b1 is valid.

Examples:

EVA 10 * -2
1111111111101100Y 177754Q 65516T FFECH ‘L’

EVA .AA AND NOT .BB
0Y 0Q 0T 0H "

B. b1 >> u1

This produces an error. The operator b1 must be evaluated next, and requires two numeric operands, but u1 operand has not yet been evaluated to a numeric result.

Examples:

EVA 10 + BYTE .AA
SYNTAX ERROR

EVA .AA MASK NOT .BB
SYNTAX ERROR
Case 8: EVALUATE \textit{operand} \textit{bl} (u1 \textit{operand})

Unary operator \textit{u1} is evaluated first, even if it has lower precedence than binary operator \textit{bl}. Parentheses must be used when \textit{u1} has lower precedence than \textit{bl}.

Examples:

\begin{verbatim}
EVA 10 + (BYTE .AA)  
101101Y 55Q 45T 2DH '–'

EVA .AA MASK (NOT .BB)  
0Y 0Q 0T 0H ''
\end{verbatim}

Case 9: EVALUATE \textit{u1} \textit{u2} \textit{operand}

Unary operator \textit{u2} must have higher precedence than \textit{u1} to evaluate without an error.

A. \textit{u1} \textit{>>} \textit{u1} is valid.

Examples:

\begin{verbatim}
EVA BYTE -F00H  
111110Y 76Q 62T 3EH '>'

EVA NOT BYTE .AA  
1111111110111100Y 177734Q 65500T FFDCH ''
\end{verbatim}

B. \textit{u1} \textit{>>>} \textit{u2}

Examples of this case shown below result in an error.

Examples:

\begin{verbatim}
EVA BYTE NOT .AA  
SYNTAX ERROR

EVA – BYTE .AA  
SYNTAX ERROR

EVA BYTE BYTE 1000H  
SYNTAX ERROR

EVA – – 5  
SYNTAX ERROR
\end{verbatim}

Case 10: EVALUATE \textit{u1} (\textit{u2} \textit{operand})

Unary operator \textit{u2} is evaluated first, even if it has lower precedence than \textit{u1}. Parentheses must be used when \textit{u2} has lower precedence than \textit{u1}.

Examples:

\begin{verbatim}
EVA BYTE (NOT .AA)  
111101Y 75Q 61T 3DH '='

EVA – (BYTE .AA)  
1111111110111101Y 177735Q 65501T FFDCH ']'''
\end{verbatim}
Expressions ICE-88

EVA BYTE (BYTE 1000H)
11111110Y 376Q 254T FEH ‘t’

EVA - (-5)
101Y 5Q 5T 5H ‘’

Two other "cases" can be diagrammed as:

\[
\text{operand b1 b2 operand} \\
\text{operand u1 b2 operand}
\]

Both forms produce an error no matter which operator has higher precedence, and no arrangement of parentheses can resolve the error.

These examples show the basic ways to control evaluation with and without parentheses. Parentheses must be used when two operators are concatenated and the second operator has lower precedence than the first.

Command Contexts

All expressions produce numeric values as results. The interpretation or use of the result depends upon the command that contains the expression. The term numeric-expression means an expression in a numeric command context. Numeric command contexts treat the result as an numeric value; all bits are significant.

The term boolean-expression means an expression in a boolean command context. Only integer values may be used in boolean contexts. Boolean command contexts test only the least significant bit (LSB) of the result, to obtain a TRUE or FALSE value. The result of a boolean expression is TRUE if its LSB is 1, FALSE if its LSB is 0. Thus, any number can have a boolean interpretation.

The BOOL command can be used instead of the EVALUATE command to display the evaluation of an expression as TRUE or FALSE.

A boolean expression uses relational and logical operators to manipulate TRUE/FALSE values. When a relational operator is evaluated, the result is always either 0 (FALSE) or FFFFH (TRUE). These results can have a numeric interpretation, but relational operators have limited usefulness in numeric contexts.

When logical operators are applied to TRUE/FALSE values, the results are also boolean. Specifically:

\[
\begin{align*}
\text{NOT:} & \quad \text{NOT FALSE} \rightarrow \text{TRUE} \\
& \quad \text{NOT TRUE} \rightarrow \text{FALSE} \\
\text{AND:} & \quad \text{TRUE AND TRUE} \rightarrow \text{TRUE} \\
& \quad \text{TRUE AND FALSE} \rightarrow \text{FALSE} \\
& \quad \text{FALSE AND TRUE} \rightarrow \text{FALSE} \\
& \quad \text{FALSE AND FALSE} \rightarrow \text{FALSE} \\
\text{OR:} & \quad \text{TRUE OR TRUE} \rightarrow \text{TRUE} \\
& \quad \text{TRUE OR FALSE} \rightarrow \text{TRUE} \\
& \quad \text{FALSE OR TRUE} \rightarrow \text{TRUE} \\
& \quad \text{FALSE OR FALSE} \rightarrow \text{FALSE}
\end{align*}
\]
XOR: TRUE XOR TRUE → FALSE
   TRUE XOR FALSE → TRUE
   FALSE XOR TRUE → TRUE
   FALSE XOR FALSE → FALSE

In addition to numeric and boolean contexts, there are several other contexts that control the interpretation or use of a number or expression. These contexts are summarized in table 5-9 for reference.

Table 5-9. Command Contexts

<table>
<thead>
<tr>
<th>Type of Entry</th>
<th>Contexts</th>
<th>Interpretation</th>
<th>Limitations</th>
<th>Examples of Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric expression</td>
<td>Set and change commands,</td>
<td>16-bit unsigned number; bit size</td>
<td>All operands and operators allowed.</td>
<td>IP = AA*256T + 10FFFH</td>
</tr>
<tr>
<td></td>
<td>etc.</td>
<td>may be reduced to fit destination.</td>
<td>Numeric constant without suffix is</td>
<td></td>
</tr>
<tr>
<td>Boolean expression</td>
<td>BOOL, IF, UNTIL, WHILE</td>
<td>LSB = 0 → FALSE</td>
<td>All operands and operators allowed.</td>
<td>AA AND .BB AND NOT .CC</td>
</tr>
<tr>
<td>Address</td>
<td>FROM, content-operator,</td>
<td>LSB = 1 → TRUE</td>
<td>Numeric constants without suffix are</td>
<td></td>
</tr>
<tr>
<td></td>
<td>partition, SAVE</td>
<td></td>
<td>interpreted in current default radix.</td>
<td></td>
</tr>
<tr>
<td>Decimal number</td>
<td>statement-number, MOVE,</td>
<td>positive number</td>
<td>Only arithmetic operators are allowed</td>
<td>GO FROM .BB + 10</td>
</tr>
<tr>
<td></td>
<td>PRINT</td>
<td></td>
<td>outside of the outermost parentheses.</td>
<td></td>
</tr>
</tbody>
</table>

All constants without suffix are decimal.
Chapter 6 contains discussions, examples, and syntax summaries for each of the ICE-88 emulation and trace control commands.

The following brief outline of Chapter 6 shows how the emulation and trace control commands have been classified.

Emulation Control Commands
- Set Breakpoint Register Command
- Set Tracepoint Register Command
- GO Command
- GR Command
- STEP Command
- Display Emulation Register Command
- Set CLOCK Command
- Display CLOCK
- Set RWTIMEOUT Command
- Display RWTIMEOUT Command
- ENABLE/DISABLE RDY Command

Trace Control Commands
- Set TRACE Display Command
- ENABLE/DISABLE TRACE Command
- Display TRACE
- MOVE, OLDEST, and newest Commands
- PRINT Command

Emulation Control Commands
The ICE-88 emulator contains an 8088 as the emulation processor. During emulation, this processor executes the instructions in the user program that have been mapped and loaded into the ICE-88 system. The operations of the user system can be monitored through the 8088 processor signals. The commands in this section allow you to specify the starting address where emulation is to begin, and to specify and display the software or hardware conditions for halting emulation and returning control to the console for further commands.
The commands in this section are as follows:

<table>
<thead>
<tr>
<th>COMMAND</th>
<th>PURPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Breakpoint-Register</td>
<td>Set match condition for halting emulation.</td>
</tr>
<tr>
<td>Set Tracepoint-Register</td>
<td>Set match condition for starting or halting trace data collection.</td>
</tr>
<tr>
<td>GO command</td>
<td>Begin real-time emulation.</td>
</tr>
<tr>
<td>GR command</td>
<td>Enable or set and enable breakpoint registers to halt emulation.</td>
</tr>
<tr>
<td>STEP command</td>
<td>Execute single-step emulation.</td>
</tr>
<tr>
<td>Display Emulation Register</td>
<td>Display GO-register, breakpoint and tracepoint register settings.</td>
</tr>
<tr>
<td>Set CLOCK</td>
<td>Designate system clock.</td>
</tr>
<tr>
<td>Display CLOCK</td>
<td>Display clock setting.</td>
</tr>
<tr>
<td>Set RWTIMEOUT</td>
<td>Enable or disable halting of emulation and error message on memory access timeout.</td>
</tr>
<tr>
<td>Display RWTIMEOUT</td>
<td>Display current setting of memory access timeout.</td>
</tr>
<tr>
<td>ENABLE/DISABLE RDY</td>
<td>Enable or disable user ready signal for memory access.</td>
</tr>
</tbody>
</table>

**Discussion**

The emulation control commands tell the ICE-88 emulator where to start emulation and when to halt emulation. After the ICE-88 emulator has been loaded by ISIS-II, the following initialization is executed:

- The GO-register (GR) is set to FOREVER. The setting of GR identifies the combination of factors that are enabled to halt emulation. The setting FOREVER means no factors are enabled.
- Both breakpoint registers (BR0 and BR1) are set to don’t care and initially disabled.

To initialize for emulation, you map the locations in prototype and ICE-supplied memory that are to be accessible to the ICE-88 emulator, and load your program code into mapped locations. After the code has been loaded, the ICE-88 emulator initializes for emulation as follows:

- The instruction pointer (IP) and code segment register (CS) are loaded with the address of the first executable instruction in your program.

Now you can begin emulation by entering the command GO, followed by a carriage return. At the command GO, the following occurs:

- Emulation begins with the instruction at the address that is in the IP and CS; this is the first executable instruction in your program.
• The message EMULATION BEGUN is displayed at the console.
• Emulation continues until you press the ESC key, or until a fatal error occurs. (See Appendix B for error messages.)

Now if you press the ESC key, the following happens.
• The ICE-88 emulator completes executing the current instruction.
• Emulation halts; the IP and CS contain the address of the next instruction to be executed.
• The message EMULATION TERMINATED, CS:IP=bbbb:ddddH is displayed. The value displayed is the address of the next instruction to be executed.
• The message PROCESSING ABORTED is displayed, acknowledging the user abort (ESC key).

This is the simplest case of starting and stopping emulation. When the GO-register is set to FOREVER, you can enter the command GO to start emulation at the current CS:IP address, and press the ESC key to halt emulation.

Instead of starting wherever the CS:IP happens to be, you may specify the starting address you want for each GO command. There are two ways to do this. First, you can set the CS:IP directly to any desired address with commands of the form CS = expr, IP = expr, then enter the GO command to start emulation at that address. Second, you can specify the starting address as part of the GO command; this form of the GO command is as follows:

GO [FROM address]

The meta-term address means the following type of entry.

numeric-expression A numeric expression is evaluated to give the address (see Chapter 5). (Table 5-9 specifies restrictions.)

For example, to start emulation with the instruction at memory location 3000H, you could enter:

CS = 0
IP = 3000H
GO

Or, you can enter:

GO FROM 3000H

The effect is the same either way.

The following form of the GO command is also valid.

GO [FROM address] FOREVER

This form of the GO command enables you to optionally select the starting address and to disable the factors that halt emulation. For example, to start emulation with the instruction at memory location 3000H and to set the GO-register to FOREVER, you can enter:

GO FROM 3000H FOREVER

The effect of this command is to start emulation with the instruction at location 3000H. Emulation will stop only when you abort processing.
The ICE-88 emulator has two breakpoint registers, BR0 and BR1. Each of these registers can be set to hold a "match condition" that can be used to halt real-time emulation when the register is enabled. A second form of the GO command can be used to both load and enable breakpoint registers. This form of the GO command is:

\[
\text{GO [FROM } address \text{] [TILL } match-condition \text{ ] [OR \{ AND \} match-condition]}\]

This command loads the match-conditions into the breakpoint registers and enables the registers to halt emulation on the desired set of system conditions contained in these match-conditions. Match conditions are of two types:

\[
\text{match-condition} = \begin{cases} \text{execution-match-condition} \\ \text{non-execution-match-condition} \end{cases}
\]

The breakpoint registers may be set to contain either type of match-condition.

**Execution Match Condition**

An execution-match-condition consists of a single 20-bit field plus the keyword EXECUTED, where each address bit can take any one of three values: 0, 1, or "don't care." An execution match condition is examined when the 8088 CPU executes an instruction byte, that is, when the byte is fetched from the 8088 instruction queue. The condition "matches" when the executed instruction byte was obtained from a memory location whose 20-bit address matches the contents of the selected breakpoint register.

\[
\text{execution-match-condition} = \begin{cases} \text{address EXECUTED} \\ \text{masked-const EXECUTED} \end{cases}
\]

Entering an address causes all 20 bits of the match condition to be loaded with 0 and 1 bit values. The address contains a base and displacement (e.g., .X or 50:3000H); note that a single constant is evaluated modulo 65536 (e.g., 12345H is the same as 2345H—use 1234:5H to get all 20 bits). Entering a masked-constant causes the 20 bit field to contain 0, 1, or "don't care" values. The "don't care" values are ignored. The masked-constant can be 20 bits in length.

The following examples illustrate the use of this form of the GO command. The examples assume that the initial contents of the breakpoint registers are as shown below:

- BR0 = XXXXXH (all bits set to "don't care")
- BR1 = XXXXXH (all bits set to "don't care")

Also, in these examples, the address of .START is 0000:0002H, .DELAY is at 0000:00DEH, and .DISPLAY is at 0000:0098H. Each example will list a GO command followed by the contents of the breakpoint registers as set by the command.

1. Go from .START until the first instruction in .DELAY is executed.

\[
\text{GO FROM .START TILL .DELAY EXECUTED}
\]

- BR0 = 000DEH E (000DEH is the 20-bit address of .DELAY, the last E specifies "EXECUTED")
- BR1 = XXXXXH (BR1 is unchanged)
ICE-88 Emulation and Trace Control Commands

This command loads BR0 with the given match condition: "DELAY EXECUTED".

2. Go from START until address 0200H is executed.

   GO FROM .START TILL 0200H EXEC
   BR0 = 00200H E
   BR1 = XXXXH

   This command loads the numeric address and "executed" status into BR0 and leaves BR1 unchanged.

3. Go from START until address 100:0200H is executed.

   GO FROM .START TILL 100:0200H EXEC
   BR0 = 01200H E
   BR1 = XXXXH

   The pointer address 100:0200H and "executed" status are loaded into BR0 and BR1 is unchanged.

4. Go from .START until an address in an address range specified by a masked constant is executed.

   GO FROM .START TILL 10XXH EXECUTED
   BR0 = 010XXH E
   BR1 = XXXXH

   The masked constant address loaded into BR0 specifies a range of addresses: 01000H through 010FFH. BR0 remains unchanged.

5. Load two execution match conditions, one in each breakpoint register, and "OR" the conditions.

   GO FROM .START TILL .DELAY EXEC OR .DISPLAY EXEC
   BR0 = 000DEH E (Halt when .DELAY is executed.)
   BR1 = 00098H E (Halt when .DISPLAY is executed.)

   This command sets emulation to halt when either the instruction located at location 000DEH (.DELAY) or location 00098H (.DISPLAY) is executed.

6. Load two execution match conditions and "AND" them.

   GO FROM .START TILL .DELAY EXEC AND .DISPLAY EXEC
   BR0 = 000DEH E (Halt when .DELAY is executed.)
   BR1 = 00098H E (Halt when .DISPLAY is executed.)

   ERR AE:INVALID "AND" IN GO-REG (Error message generated by this command)

This command attempts to set emulation to halt when the instruction located at location 000DEH (.DELAY) "AND" the instruction located at location 00098H (.DISPLAY) are executed. Execution of two separate instructions cannot occur at the same time. Therefore an error message is generated by this command and the command is not executed.
Non-Execution Match Condition

The *non-execution-match-condition* must contain one or more of four types of fields: a set of addresses, a list of bus status types, a set of data values, and a segment register designation. A *non-execution-match-condition* matches whenever a breakpoint that contains a set of one or more of the above fields matches corresponding state values in the user system during real-time or single step emulation.

\[
\text{non-execution-match-condition} = \{ \begin{align*}
\text{address-match-range} \\
\text{match-status-list} \\
\text{data-match-range} \\
\text{segment-register-usage}
\end{align*} \} \ldots
\]

Address-match-range, match-status-list, data-match-range, and segment-register-usage must be used in the order shown. At least one of these fields must be entered in a given command to establish a non-execution match condition.

Address Match Range

An *address-match-range* may consist of a single address or masked constant, an "unlimited" range of addresses, or a set of match partitions. If an "unlimited" range of addresses are to be entered, an address value modified by the mnemonic UP or DOWN is entered. UP implies any address value equal to or greater than the stated address. DOWN implies any address value equal to or less than the stated address. The match partition may be any of three types: partitions, memory references, and/or typed memory references. If the *address-match range* contains more than one partition, all partitions must have the same base and their displacements must lie within a 1K-byte range. If there is only a single partition, it must lie within a 1K-byte range to be contained in a single breakpoint register. If the partition does not lie within a 1K range, two registers are required to hold the partition. Therefore an *address-match range* can be defined as:

\[
\text{address-match-range} = \begin{cases}
\text{address :: masked-const} \\
\text{address UP :: address DOWN} \\
\text{partition} \\
\text{OBJECT memory-reference} \\
\text{OBJECT typed-memory-reference}
\end{cases}, \ldots
\]

By expanding the definition of partitions and memory reference to their component parts, the definition of *address-match range* becomes:

\[
\text{address-match-range} = \begin{cases}
\text{address :: masked-const} \\
\text{address UP :: address DOWN} \\
\text{address TO address} \\
\text{address LENGTH length} \\
\text{OBJECT BYTE address} \\
\text{OBJECT WORD address} \\
\text{OBJECT SINTEGER address} \\
\text{OBJECT INTEGER address} \\
\text{OBJECT POINTER address} \\
\text{OBJECT typed-memory-ref}
\end{cases}
\]
For example, an \textit{address-match-range} of a single address would be 3000H, whereas using the masked constant 30XXH would result in a match range of 3000H through 30FFH. Two examples of the use of partitions in a match range are:

\begin{verbatim}
4000 TO 4100
\end{verbatim}

and

\begin{verbatim}
4000 LENGTH 101
\end{verbatim}

Both of these partition specifications result in the range of addresses 4000 through 4100.

A sequence of discontinuous addresses can be specified by:

\begin{verbatim}
OBJECT BYTE 4000, OBJECT WORD 3188, OBJECT !!MOD1 !SYMSAM, ...
\end{verbatim}

This would result in a string of discontinuous match addresses, the third address in the above string being specified by a typed memory reference. \textsc{object} indicates a partition beginning at the low address of the memory object and whose length is the length of the object. In the above example, "OBJECT BYTE 4000" specifies a one-byte partition whose address is 4000. "OBJECT WORD 3188" specifies a two-byte partition starting at 3188 and ending at 3189. "OBJECT .. MOD1 .SYMSAM" specifies a partition starting at the address given in the symbol table for !!MOD1 !SYMSAM and whose length is specified by the memory type of symbol .SYMSAM. For example if .SYMSAM is type WORD, the partition will be two bytes in length.

\textbf{Match Status List}

The \textit{match-status-list} field matches whenever the 8088 bus status is any of those listed in the match status list:

\begin{verbatim}
READ match on memory read other than an instruction fetch.
WRITTEN match on memory write.
INPUT match on an I/O read.
OUTPUT match on an I/O write.
FETCHED match on a memory read into the execution queue.
HALT match on 8088 halt.
ACKNOWLEDGE match on 8088 interrupt acknowledge.
\end{verbatim}

A match-status-list may consist of one or more of the above bus status types and they may be listed in any order:

\begin{verbatim}
match-status-list \equiv [READ
  WRITTEN
  INPUT
  OUTPUT
  FETCHED
  HALT
  ACKNOWLEDGE]...
\end{verbatim}

An example of a match-status-list would be:

\begin{verbatim}
FETCHED, READ, HALT, WRITTEN, ACKNOWLEDGE
\end{verbatim}
Data Match Range

The data-match-range field can be used to specify data values. The syntax is similar to address-match-range, except the OBJECT form is not allowed. In this case, the values specified by address will be treated as data values and used to match against values on the 8088 address/data lines at data time.

\[
data-match-range = VALUE \begin{array}{c} \text{address} :: \text{masked-const} \\ \text{address UP} :: \text{address DOWN} \\ \text{[partition]}, \ldots \end{array}\]

Data values must be integers. The set of values which match this field is the same as for address-match-range. Since the 8088 deals only with 8-bit data values, only the low order 8 bits in the data values are significant.

Segment Register Usage

The segment-register-usage field is used to specify one of the four segment registers. A match occurs whenever the segment register used in an effective address calculation is the one specified in the segment-register-usage field. Segment register usage occurs at data time.

\[
\text{segment-register-usage} = \begin{array}{c} \text{USING SS} \\ \text{USING CS} \\ \text{USING DS} \\ \text{USING ES} \end{array}\]

Match Condition Restrictions

Figure 6-1 illustrates a detailed specification of the non-execution-match-condition. The following examples illustrate a set of restrictions that must be observed in the use of match conditions in emulation commands.

Data values, bus status and segment register usage come out of the 8088 at data-time. Address values and bus status are available at address-time. The following restrictions apply to match conditions.

- Breakpoint register BR0 cannot contain data-time values and breakpoint register BR1 contain address-time values if the two registers are ANDed (the reverse is permissible).
- Neither BR0 nor BR1 may contain an execution-match-condition if they are to be ANDED.

A warning message is issued after a Set BR command or a Set GR command if the command results in either of the above conditions. An Error is issued on a GO command if the command results in either of the above conditions.

- If a match-condition specifies both address and data or segment register usage, the match condition requires both breakpoint registers; hence this match-condition cannot OR/AND with another match-condition.

All partitions in a multi-partition match condition must have the same base value. For example, the following command generates the error message shown.

```
GO FROM .START TILL 0:0000 LEN 2, 100:1000H READ
ERR AD:DIFFERING BASIS (error message)
```
Figure 6-1. Non-Execution Match Condition

Note: (address-match-range), (match-status-list), (data-match-range), and (segment-register-usage) must be used in the order shown. At least one of these fields must be entered in a given command and no field may be repeated in the command.
All the displacement values must be within a 1K-byte range to be contained in a single breakpoint register. Displacements which exceed 1K must be contained in a single partition.

The following match condition cannot be contained in one breakpoint register.

```
GO FROM .START TILL 0:0000 LENGTH 2048T WRITTEN
```

The above command would require both breakpoint registers as the partition is 2K-bytes in length.

The following command would generate an error as two partitions are specified and the displacements exceed 1K.

```
GO FROM .START TILL 0, 2048T WRITTEN
```

The following command would generate an error as the partition requires both breakpoint registers. Therefore the partition "BEGIN" cannot be entered.

```
GO FROM .START TILL 0 LEN 2048T, .BEGIN W
```

Segment register usage can only be used in conjunction with a single match value as illustrated in the example below.

```
GO FROM .DELAY TILL .MAINTIME READ USING DS
```

The last form of the GO command is in the following format:

```
GO [FROM address] [TILL break-reg [AND break-reg]]
```

where `break-reg` references breakpoint register BR0, BR1 or BR. This command form is used when the breakpoint registers have been set prior to the entering of this command. The command enables the referenced breakpoint register but does not set its contents. The Set Breakpoint command, GR command, or a previous GO command must be used to set the required breakpoint registers. Care is required in ANDing two breakpoint registers in this command. Only two non-execution-match-conditions can be "anded." An error results if either of the conditions described below occur.

- BR0 contains data values or segment register usage and BR1 contains address values.
- Either of the breakpoint registers contains an execution-match-condition.

### Setting the Go-Register

To enable either (or both using BR) of the breakpoint registers as a halt condition, you can use a set GR command of the form:

```
GR = halt-go-condition
```

The meta-term `halt-go-condition` means any of three exclusive types of halt conditions:

```
halt-go-condition ≡ \{ \text{FOREVER} \} \cup \{ \text{TILL break-reg \{ AND break-reg \}} \} \cup \{ \text{TILL match-cond \{ AND match-cond \}} \}
```

6-10
Using the FOREVER condition in the Go-Register command:

\[ GR = \text{FOREVER} \]

would disable both breakpoint registers.

The following command would enable BR1:

\[ GR = \text{TILL BR1} \]

Both registers can be enabled and ORed with the following command:

\[ GR = \text{TILL BR1 OR BR0} \]

BR0 would be loaded with a match condition and enabled with the following command:

\[ GR = \text{TILL 3000H WRITTEN, FETCHED} \]

The following command would load BR0 with the first match condition and BR1 with the second stated match condition:

\[ GR = \text{TILL 3000H WRITTEN, FETCHED OR INPUT VALUE 01 USING DS} \]

BR0 would contain the match condition 3000H WRITTEN, FETCHED and BR1 would contain the match condition INPUT VALUE 01 USING DS and both registers would be enabled and ORed.

The following command would require both breakpoint registers to contain the match condition:

\[ GR = \text{TILL .DELAY FETCHED OR READ VALUE .MAINTIME USING DS} \]

BR0 would contain the match condition .DELAY FETCHED and BR1 would contain the match condition READ VALUE .MAINTIME USING DS.

The following command would require both breakpoint registers to contain match conditions that are "anded":

\[ GR = \text{TILL !SIDETIME READ AND VALUE 8} \]

BR0 would contain the condition .SIDETIME READ and BR1 would contain the match condition VALUE 8. These conditions are ANDED.

### Setting Tracepoint Registers

The ICE-88 emulator has two tracepoint registers, ONTRACE and OFFTRACE. A tracepoint register may only contain a non-execution match condition. Also the match range may only contain an address, masked constant, or data, and segment register usage may not be used with an address condition. For example, the following commands are valid:

\[ \text{ONT} = 3000H \text{ WRITTEN, FETCHED} \]
\[ \text{OFF} = \text{INPUT VALUE 01 USING DS} \]
However, the commands:

\[
\text{OFFTRACE} = 3000H \ \text{EXECUTED} \\
\text{ONTRACE} = 3000H \ \text{READ USING CS}
\]

are invalid as \text{EXECUTED} is invalid in a tracepoint, and the segment register CS is specified with an address condition.

**Command Signal Timeout**

When the 8088 accesses INTELLEC- or DISK-mapped memory, a command signal timer starts counting. If the access is not completed before it times out, the ICE-88 emulator will cause the READ and WRITE command signals to go inactive to the user system. The \text{RWTIMEOUT} commands are used to set and display the current setting of the command signal timer.

**Emulation Timer**

An emulation timer is enabled when emulation is running. The timer can be used to determine how long it takes the ICE-88 emulator to emulate a given segment of code. The timer is a 2-MHz clock (i.e., counts are intervals of 500 ns), derived from the crystal on the Control board.

The timer starts when the GO command is entered, starting emulation. The timer stops counting where a maximum count of approximately 33 minutes is reached. \text{TIMER} continues counting modulo 65536.

The timer is reset to 0 (before starting to count) when the GO command is entered with a FROM clause or when CS,IP is changed or when ENABLE/DISABLE TRACE. If you want to reset the timer without changing the current program counter, enter a command such as GO FROM CS:IP.

After emulation halts, you can display the value of the timer in the current output radix. The display command \text{TIMER} displays the low 16 bits of the timer value; the command \text{HTIMER} displays the high 16 bits of the timer value. The tokens \text{TIMER} and \text{HTIMER} can also be used as keyword references in commands and expressions.

With the timer, you can measure the real elapsed time required to emulate a given code sequence. The elapsed time can then be compared to the calculated time based on the number of clock states in each instruction and the speed of the system clock. Note that code mapped to user runs at real-time; the timer value for code mapped to prototype memory is the real-time value.
Set Breakpoint Register Command

(1) \( \text{break-reg} = \text{address} \text{EXECUTED}::\text{masked-const} \text{EXECUTED} \)

(2) \( \text{break-reg} = [\text{address-match-range}][\text{match-status-list}][\text{data-match-range}][\text{seg-reg-usage}] \)

NOTE

Form (2) requires that the address-match-range, match-status-list, data-match-range, and seg-reg-usage fields be used in the order shown. At least one field is required in a given command and no field may be repeated in the command. Restriction: the fields selected must fit in one breakpoint register.

Examples:

\[
\begin{align*}
\text{BR0} &= 1XXXH \text{EXECUTED} \\
\text{BR1} &= 3000H \text{UP WRITTEN} \\
\text{BR} &= 3000H \text{TO} 30FFH \text{READ} \\
\text{BR} &= 3000H \text{LENGTH} \text{FEH, OBJECT !VAR WRITTEN}
\end{align*}
\]

\( \text{break-reg} \)

The name of one of the breakpoint registers (BR0, BR1) or BR to set both registers to the same match condition.

\( = \)

The assignment operator.

\( \text{address} \)

The address of the memory location or I/O port, or a data value.

\( \text{masked-const} \)

A masked constant used to define a range of memory locations or data values.

\( \text{EXECUTED} \)

Denotes that the match condition is the execution (CPU fetch of the instruction byte from the instruction queue) of the instruction byte whose address is given by address or masked-const.

\( \text{address-match-range} \)

A set of one or more addresses. (See page 6-6.)

\( \text{match-status-list} \)

A set of bus status conditions to be used as match parameters. (See page 6-7.)

\( \text{data-match-range} \)

A set of data values to be used as match parameters. (See page 6-8.)

\( \text{seg-reg-usage} \)

A specification of one of the segment registers to be used as a match parameter. (See page 6-8.)
Set Tracepoint Register Command

\[
\begin{array}{|c|c|c|c|}
\hline
\text{trace-reg} & \text{address} & \text{READ} & \text{VALUE \ address} \\
\text{masked-const} & \text{WRITTEN} & \text{INPUT} & \text{USING \ SS} \\
& \text{OUTPUT} & \text{FETCHED} & \text{USING \ CS} \\
& \text{HALT} & \text{ACKNOWLEDGE} & \text{USING \ DS} \\
\hline
\end{array}
\]

\(\text{address-match-range}\) \hspace{1cm} \text{(data-match-range)} \hspace{1cm} \text{(segment-register-usage)}

\(\text{match-status-list}\) \hspace{1cm} \text{USING \ ES}

\text{NOTE}

The \text{address-match-range}, \text{match-status-list}, \text{data-match-range}, and \text{segment-register-usage} fields must be used in a command in the order shown. At least one field is required in a given command and no field may be repeated in the command. A \text{segment-register-usage} field or \text{data-match-range} may not be used with an address condition (\text{address-match-range} field); because you cannot mix address-time fields with data-time fields.

Examples:

\begin{align*}
\text{ONTRACE} &= 2340 \text{ READ, ACKNOWLEDGE} \\
\text{OFFTRACE} &= \text{INPUT, OUTPUT VALUE 1234H} \\
\text{ONTRACE} &= \text{!X FETCHED} \\
\text{ONTRACE} &= R,W \text{ VALUE 40XX USING ES} \\
\text{OFFTRACE} &= \text{USING ES}
\end{align*}

\(\text{trace-reg}\) \hspace{1cm} \text{The name of one of the tracepoint registers, ONTRACE or OFFTRACE.}

\(=\) \hspace{1cm} \text{The assignment operator.}

\(\text{address}\) \hspace{1cm} \text{The address of the memory location or I/O port, or a data value (see Data Match Range).}

\(\text{masked-const}\) \hspace{1cm} \text{A masked constant used to define a range of memory locations or data values.}

\(\text{match-status-list}\) \hspace{1cm} \text{See Match Status List. (See page 6-7.)}

\(\text{data-match-range}\) \hspace{1cm} \text{See Data-Match Range. (See page 6-8.)}

\(\text{segment-register-usage}\) \hspace{1cm} \text{See Segment Register Usage. (See page 6-8.)}
Go Command

\[
\text{GO} \left\{ \begin{array}{c}
\text{FROM address} \\
\text{FOREVER} \\
\text{TILL \ break-reg \ [\{AND\break-reg\}\{OR\break-reg\}\{AND\\\text{break-reg}\}\{OR\\text{break-reg}\}]}
\end{array} \right.
\]

Examples:

GO
GO FROM 3000H
GO FROM .START TILL BR0
GO FROM 3000H TILL 3000H EXECUTED
GO TILL INPUT VALUE 10
GO FROM 1000H TILL 3000H TO 30FFH READ USING DS
GO FROM 3000H TILL OBJECT POINTER .START READ

GO Command

Command keyword that starts emulation, subject to the current start and halt conditions.

FROM
Keyword introducing a starting address.

deadress
The address of the memory location of the first instruction to emulate, i.e., the start address.

FOREVER
Disables all breakpoint conditions; emulation can be stopped only by user aborting processing.

TILL
A keyword introducing one or more match or halt conditions.

break-reg
One of the breakpoint registers (BR0, BR1), or BR to set both registers to the same match setting.

match-cond
One of the following forms of breakpoint register settings.

1. execution-match-condition. (See page 6-4.)

2. non-execution-match-condition. (See page 6-6.)

NOTES
The ICE-88 emulator cannot enter GO or STEP with the 8088 Trap Flag (TFL) set. Therefore a warning message will be issued whenever GO or STEP commands are executed with TFL = 1, and TFL will be set to 0.

If either breakpoint register contains a match range other than a single match-value and the breakpoint register has changed since the last GO command, the message “LOADING RANGE BREAKPOINTS” is issued and, it takes approximately 10 seconds to load breakpoints and hardware before emulation begins.
Set GO-Register (GR) Command

\[
GR = \left\{ \begin{array}{l}
\text{FOREVER} \\
\text{TILL break-reg} \left[ \begin{array}{l}
\text{AND} \\
\text{OR}
\end{array} \right] break-reg \\
\text{TILL match-cond} \left[ \begin{array}{l}
\text{AND} \\
\text{OR}
\end{array} \right] match-cond
\end{array} \right.
\]

Examples:

- \( GR = \text{FOREVER} \)
- \( GR = \text{TILL BR1} \)
- \( GR = \text{TILL BR0 OR BR1} \)
- \( GR = \text{TILL OBJECT !ABLE1} \)
- \( GR = \text{TILL OBJECT POINTER 0123 READ, WRITTEN VALUE 30 USING DS} \)

<table>
<thead>
<tr>
<th>GR</th>
<th>Command keyword referring to the GO-register (halting conditions for emulation).</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>The assignment operator.</td>
</tr>
<tr>
<td>FOREVER</td>
<td>Disables all breakpoint conditions; emulation can be stopped only by user aborting processing.</td>
</tr>
<tr>
<td>TILL</td>
<td>A keyword introducing one or more match or halt conditions.</td>
</tr>
<tr>
<td>break-reg</td>
<td>One of the breakpoint registers, BR0 or BR1 (or BR to denote both breakpoint registers) that is to be enabled.</td>
</tr>
<tr>
<td>match-cond</td>
<td>One of the following forms of breakpoint register settings:</td>
</tr>
</tbody>
</table>

1. execution-match-condition. (See page 6-4.)
2. non-execution-match-condition. (See page 6-6.)
STEP Command

STEP [FROM address]

Examples:

STEP
STEP FROM 1FFH
STEP FROM ..MOD.GO
STEP FROM !PTR
STEP FROM #123 + 10
STEP FROM CS:(WORD .X) ;SHORT JUMP INDIRECT THROUGH .X

STEP
A command keyword that causes the ICE-88 emulator
to execute a single step of emulation.

FROM
A function keyword introducing the address where a
single step of emulation is to be executed.

address
See address. (See page 6-14.)

The STEP command causes the ICE-88 emulator to execute one single step of
emulation. If FROM address is not included in the command, the emulation step
is executed from the current address. If FROM address is included in the com-
mand, the value of the address is loaded into the CS and IP and the step is ex-
ecuted from this location.

NOTE
The STEP command is very useful in repeat loops and macros (see
Chapter 8), where terminating condition can be given (UNTIL or
WHILE) and system status and values can be displayed after each step.
However, the user is cautioned that a hardware reinitialization occurs
intermittently with a reset timeout when the RESET pin is pulsed during
a repeat of the STEP command.
Display Emulation Register Command

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GR</td>
<td>A command keyword that causes the content of the GO-register (factors enabled to halt emulation) to be displayed.</td>
</tr>
<tr>
<td>break-reg</td>
<td>One of the breakpoint register keywords BR0 or BR1, to obtain a display of the register setting, or the keyword BR to cause the display of the settings of both breakpoint registers.</td>
</tr>
<tr>
<td>trace-reg</td>
<td>One of the tracepoint register keywords ONTRACE or OFFTRACE to command the display of the content of the designated register.</td>
</tr>
<tr>
<td>ABSOLUTE</td>
<td>Display all addresses as 20-bit numbers (this is the default).</td>
</tr>
<tr>
<td>BASE</td>
<td>Display all addresses in base and displacement format (e.g., 0000:1000H). If no expr is given, display with the base that was used to set the register.</td>
</tr>
<tr>
<td>expr</td>
<td>An integer value that specifies that all addresses are to be displayed as their displacement from (expr )*16. An error occurs if an address needs a displacement of less than 0 or greater than 65535 from the base (expr ). Typically expr will be a segment register name; thus “BR0 BASE CS” displays the displacements of the addresses in BR0 using the current code segment register. If no expr is given, use the base that the register was set with.</td>
</tr>
</tbody>
</table>

NOTE

Data values are always displayed as 8-bit numbers, masked-constants as 8-bit, 16-bit or 20-bit strings with Xs (in hexadecimal if possible, or else in binary).

Internal to the ICE-88 emulator, match addresses are stored as 20-bit numbers. Thus “GO TILL 20:8 R” breaks whenever 208H is read, even if it is read as 10:108H.
Set CLOCK Command

CLOCK = [INTERNAL] [EXTERNAL]

Examples:

CLOCK = INTERNAL
CLOCK = EXTERNAL

CLOCK
This command keyword enables the user to designate the type of clock being used in the system: user-provided clock or ICE-provided clock.

= The assignment operator.

INTERNAL Designates that the ICE-provided clock is being selected. This is necessary whenever the cable is not plugged into user system. When clock is set to Internal, the ICE-88 emulator is operated in stand-alone mode. The Socket Protector should be mounted on user cable in this mode of operation.

EXTERNAL Designates the clock to be user supplied. This is necessary whenever the cable is plugged into a user system.

Display CLOCK Command

CLOCK

Examples:

CLOCK

CLOCK A command keyword that causes the display of the clock setting.
Set RWTIMEOUT Command

\[
\text{RWTIMEOUT} = \begin{cases} 
\text{INFINITE} \\
\text{expr-10 [ERROR]} \\
\text{expr-10 NOERROR}
\end{cases}
\]

Examples:

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RWTIMEOUT = INFINITE</td>
<td>:DISABLE RWTIMEOUT</td>
</tr>
<tr>
<td>RWTIMEOUT = 500 ERROR</td>
<td>:SET TIMEOUT TO HALT EMULATION W/ REPORT</td>
</tr>
<tr>
<td>RWTIMEOUT = 500</td>
<td>:HALT EMULATION WITH ERROR REPORT</td>
</tr>
<tr>
<td>RWTIMEOUT = 1500 NOERROR</td>
<td>SET TIMER BUT Do NOT HALT EMULATION WHEN IT TIMES OUT</td>
</tr>
<tr>
<td>RWTIMEOUT = 1500</td>
<td>:SET TIMER BUT DO NOT HALT EMULATION WHEN IT TIMES OUT</td>
</tr>
</tbody>
</table>

RWTIMEOUT

A command keyword denoting that a command signal timeout function is to be set.

= Denotes that the command is a set signal timeout command.

INFINITE Sets command signal timeout to "infinite" effectively disabling the timeout.

expr-10 An integer value that specifies the timeout value in microseconds. The integer value must be greater than 0 and less than 32K, and the default suffix when evaluating expr-10 is decimal.

ERROR Specifies that error is to be reported whenever command signal times out.

NOERROR Specifies that command signal timeout is not to halt emulation.

Display RWTIMEOUT Command

RWTIMEOUT

Causes the current setting of the command signal timeout to be displayed.
ENABLE/DISABLE RDY Command

1. ENABLE RDY
2. DISABLE RDY

ENABLE: A command keyword denoting that an ICE-88 element is to be enabled.

DISABLE: A command keyword denoting that an ICE-88 element is to be disabled.

RDY: A reference keyword specifying the user ready signal for memory access.

The ICE-88 emulator allows the user to enable and disable the user ready signal. If RDY is enabled, the ready signal to the 8088 is a local ready (generated by the ICE-88 emulator) AND user ready; otherwise ready to the 8088 is either the local ready when mapped to local memory or user ready when mapped to user memory. RDY is initially enabled.

NOTE

Must disable RDY if clock is INTERNAL and you are using INTELLEC memory or DISK memory. When emulating in the user memory with DISABLE RDY invoked, the user ready pin must be active to continue emulation.
Trace Control Commands

The ICE-88 emulator can record program execution through the collection of trace data in a trace buffer during real-time and single-step emulation. The commands in this section allow you to specify the conditions for enabling and disabling trace data collection during emulation and to control the display of trace data.

The commands in this section are as follows:

<table>
<thead>
<tr>
<th>COMMAND</th>
<th>PURPOSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set TRACE Display Mode</td>
<td>Establishes trace data that will be displayed as frames or instructions.</td>
</tr>
<tr>
<td>ENABLE/DISABLE TRACE</td>
<td>Enables or disables the collection of trace data.</td>
</tr>
<tr>
<td>Display TRACE Mode</td>
<td>Causes the display of the current display mode.</td>
</tr>
<tr>
<td>MOVE, OLDEST, NEWEST</td>
<td>Set trace buffer pointer to entry to be displayed.</td>
</tr>
<tr>
<td>PRINT</td>
<td>Display one or more entries from the trace buffer.</td>
</tr>
</tbody>
</table>

Discussion

The unit of emulation is the instruction. During real-time and single-step emulation, the ICE-88 emulator traces program execution twice per 8088 bus cycle: first when the address signals are valid and then when the data signals are valid. It also traces each CPU clock cycle during which the execution queue is active.

The ICE-88 emulator contains a trace buffer used to collect trace data (frames) during real-time and single-step emulation. The trace buffer holds a total of 1023 frames or approximately 300 bus cycles of typical trace information. Each entry in the buffer is a frame, and is either half a bus cycle or contains queue status, or both. Each frame contains:

<table>
<thead>
<tr>
<th>bit-size</th>
<th>purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>Address/data</td>
</tr>
<tr>
<td>3</td>
<td>Bus status (S0, S1, S2)</td>
</tr>
<tr>
<td>2</td>
<td>Queue Status (QS0, QS1)</td>
</tr>
<tr>
<td>3</td>
<td>Queue depth</td>
</tr>
<tr>
<td>2</td>
<td>Frame type indicator: address, data, or queue status</td>
</tr>
<tr>
<td>1</td>
<td>Start/stop trace marker for conditional trace</td>
</tr>
</tbody>
</table>

Trace is initially unconditionally on and the buffer is initially empty. The buffer is cleared whenever the user changes the IP or CS, either by a FROM clause on a GO or STEP command or by a Change command. Otherwise new trace data is appended to the end of existing trace data and the most recent 1023 frames are retained in the buffer. Similarly, the TIMER and HTIMER registers are reset to zero each time the user changes the IP or CS register. Also, whenever the user issues an enable/disable trace command, the trace buffer is cleared to empty and TIMER and HTIMER are reset to zero when the user next enters emulation.
The user can control the collection of trace data using the tracepoint registers. The enable/disable trace command enables trace conditionally and unconditionally or disables trace unconditionally:

**ENABLE TRACE**

Turns trace on unconditionally during subsequent emulations.

**ENABLE TRACE CONDITIONALLY**

Trace will be turned on whenever the ONTRACE register matches and turned off whenever the OFFTRACE register matches.

NOW ON indicates that trace is turned on for the beginning of the next emulation; NOW OFF indicates it is off; if neither is present the trace is left in its current state.

**DISABLE TRACE**

Turns trace off unconditionally during subsequent emulations.

The trace display command allows the user to examine collected trace data displayed in one of two modes: as “raw” data or disassembled with instructions appearing as 8086 assembler mnemonics.

Instructions in the trace buffer are counted by occurrences of queue status indicating “first instruction byte out of queue” (i.e., QS0=1 and QS1=0). Since the 8088 defines instruction prefix bytes as well as the first non-prefix byte as “first instruction bytes,” an 8088 instruction with one prefix byte counts as two instructions when using the MOVE or PRINT commands. However, if a PRINT command prints the requested number of instructions and ends up after a prefix byte but before the non-prefix instruction, it completes printing the entire non-prefix instruction. When the user switches from frames to instructions mode, if the buffer pointer is not at the oldest or newest frame, then the pointer is moved to a “first byte out of queue” frame if it is not already pointing at one before beginning to MOVE or PRINT the requested number of instructions.

**Trace Display Mode**

The trace display mode controls the type of an entry to be displayed or located in the trace buffer. An entry can be a frame, or an instruction. The initial trace display mode is INSTRUCTION. To set the trace display mode, use one of the following commands.

```
TRACE = FRAME
TRACE = INSTRUCTION
```

To display an entry from the buffer, move the pointer to the desired entry and enter a PRINT command. However, it is not necessary to move the pointer if you use a PRINT ALL or PRINT-decimal command.

**Moving the Buffer Pointer**

The pointer movement commands are MOVE, OLDEST, and NEWEST.

The command OLDEST (followed by carriage return) moves the pointer to the top of the buffer, in any trace display mode. The NEWEST command moves the pointer to the bottom of the buffer (i.e., after the last instruction or frame). “Top” refers to the oldest trace data, “bottom” refers to the newest trace data.
The MOVE command has the following form:

\[ \text{MOVE} \left[ [ + \, :: \, - ] \, \text{decimal} \right] \]

The meta-term `decimal` means any numeric quantity; if no explicit input-radix is given, the ICE-88 emulator assumes decimal radix. The value of `decimal` is the number of entries between the current pointer position and the desired position. Movement in a plus (+) direction is toward the bottom (newest point) of the buffer; if neither (+) nor (-) is entered, a forward movement is assumed as the default. Movement in a minus (-) direction is toward the top (oldest point) of the buffer. The size of the move does not count the entry under the pointer when the MOVE command is given.

For example, assuming FRAME mode, if the pointer is pointing at frame 100 and you issue the command "MOVE 10", the pointer is moved to point to frame 110. Under the same initial conditions, if you issue the command "MOVE -10", the pointer is moved to point to frame 90. If `decimal-number` is larger than the number of entries between the current pointer location and the bottom (for “+”) or top (for “-”), the pointer is moved only to the bottom or top, respectively. In short, you cannot move the pointer outside the range of buffer locations.

If the MOVE command has no number following it, “MOVE 1” is executed.

The trace display mode in effect controls the size of each move. Under FRAME mode, the command MOVE 10 moves down ten frames; under instruction, the same command moves down ten instructions.

**Displaying Trace Data**

The PRINT command displays one or more entries from the buffer. This command has the form:

\[ \text{PRINT} \left[ [ + \, :: \, - ] \, \text{decimal} \right] \]

With (+) or no sign, `decimal` entries lower (toward the bottom) than the current pointer position are displayed. With (-), decimal entries above (toward the top) the current pointer position are displayed. The command PRINT without a `decimal` modifier is equivalent to PRINT 1 (one entry is displayed).

The PRINT command displays the number of entries requested, then moves the pointer to point to the next entry just past the last one displayed. As an illustration, the commands:

\[
\begin{align*}
\text{OLDEST} \\
\text{PRINT 10} \\
\text{PRINT 10}
\end{align*}
\]

are equivalent to the commands

\[
\begin{align*}
\text{OLDEST} \\
\text{PRINT 20}
\end{align*}
\]

The command PRINT ALL displays the entire trace buffer; PRINT ALL is equivalent to the commands:

\[
\begin{align*}
\text{OLDEST} \\
\text{PRINT 1023}
\end{align*}
\]
TRACE Display Formats

Display of Trace Data in Frames Mode

The display has one frame per line. The header at the top of display has the following format (one line of display shown also):

```
FRAME ADDR STS QSTS QDEPTH DMUX MARK
0000: 0002CH F N 0 A 0
```

How to interpret the Frames mode display:

**Header entry Meaning**

**FRAME**  Frame number; decimal number from 0000 to 1022. The colon separates the frame number from the next entry (ADDR).

**ADDR**  The 20-bit address in Hexadecimal radix (five digits plus suffix H) when DMUX = A (address frame). When DMUX = D (data frame), the last 2 digits (8 bits) of this number are data, and the first digit is status: S6, S5, S4, S3 (MSB to LSB). Bits S4 and S3 are the segment register used in effective address calculation:

```
S4  S3  Segment Register
0  0  ES
0  1  SS
1  0  CS or none
1  1  DS
```

**STS**  A one-character display of processor action, as follows:

- A  Interrupt Acknowledge
- F  Instruction Fetch
- H  Halt
- I  Input
- O  Output
- R  Read (Memory)
- W  Write (Memory)
- ?  Passive State

STS is valid on ADDR and DATA frames only (DMUX = A or D).

**QSTS**  Queue status; a one-character display, as follows:

- E  Empty the queue
- F  First byte of opcode executed out of queue
- N  Nothing coming out of queue
- S  Subsequent byte of opcode executed out of queue

**QDEPTH**  Number of bytes in queue (decimal number). Valid on ADDR frames only (DMUX = A).

**DMUX**  Type of frame; a one-character display as follows:

- A  Address
- D  Data
- Q  Queue
- S  Stop emulation

**MARK**  1 if trace was turned off before this frame or if emulation broke before this frame.
Display of Trace Data in Instructions Mode

The display shows the disassembled instruction mnemonic and any operands, and any succeeding cycles. Each instruction combines several frames of trace data. Machine cycles after the instruction fetch are displayed four cycles per display line, using as many lines as necessary.

First, we discuss the header and the instruction display. Display of cycles is discussed later on. The headers apply to the first line of the display entry—the line with the frame numbers. The Instructions mode header has the following format (two instructions are also shown):

<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>PREFIX</th>
<th>MNEMONIC</th>
<th>OPERANDS</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0006</td>
<td>000E7H</td>
<td>DEC</td>
<td>CL</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0010</td>
<td>000E9H</td>
<td>MOV</td>
<td>WORD PTR [0101H],BX</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00101H-W-</td>
<td>2CH-DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00102H-W-</td>
<td>00H-DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Header entry Meaning**

**FRAME**  The (decimal) number of the frame where the first byte (or prefix) of the instruction came out of the 8088 execution queue.

**ADDR**  Address of first byte (or prefix) of instruction; 20-bit number in Hexadecimal radix (five Hex digits plus suffix H).

**PREFIX**  Prefix other than segment-override (LOCK, REPE, REPNE) if specified in assembly language else blank.

**MNEMONIC**  MCS-86 assembler mnemonic for the instruction.

**OPERANDS**  Zero, one, or two operands separated by commas. The formats for the operand fields are discussed below.

**COMMENTS**  The word “;SHORT” for a JMP or CALL instruction to an address within the same segment of field bytes that contains the instruction’s address, or the word “;LONG” for a branch to a different segment, or the characters “;?” for an opcode value that does not correspond to a valid instruction.

**Operand fields**

1. **Registers:** the MCS-86 register identifiers are displayed:
   
   RAL, RAH, RBL, RBH, RCL, RCH, RDL, RDH, 
   RAX, RBX, RCX, RDX

   **Example (comments field omitted):**

<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>PREFIX</th>
<th>MNEMONIC</th>
<th>OPERANDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0003</td>
<td>00206H</td>
<td>MOV</td>
<td>AL, BYTE PTR [0000H]</td>
<td></td>
</tr>
<tr>
<td>00200H-R-</td>
<td>34H-DS</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

2. **Memory operands** have the following display format:

   ```
   [CS:DS:][BYTE][WORD][DWORD][PTR][BP][DI][SI][xxxxH][+xxH][-xxH]
   ```
Example: the display ES:BYTE PTR [BX] [SI] [ + 01H]
represents the operand BYTE ES:(BX + SI + 1)

More examples showing memory operand display.

<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>PREFIX</th>
<th>MNEMONIC</th>
<th>OPERANDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000:</td>
<td>FF380H</td>
<td></td>
<td>ADD</td>
<td>ES:BYTE PTR [BX] [SI], AL</td>
</tr>
<tr>
<td>0025:</td>
<td>FF480H</td>
<td></td>
<td>ADD</td>
<td>ES:BYTE PTR [BX] [SI] [ + 01H], AL</td>
</tr>
<tr>
<td>0050:</td>
<td>FF580H</td>
<td></td>
<td>MOV</td>
<td>AL, BYTE PTR [0001H]</td>
</tr>
</tbody>
</table>

Notes on the memory operand format:

- The first field is the segment register field. It is only displayed if the
instruction has a segment-override prefix.
- In the second field, an entry “? PTR” means that the type of the pointer
cannot be determined from the context.
  Example: LEA AX,? PTR [34A0H]
- The base register (BX, BP) and index register (DI, SI) fields are not
displayed for direct memory operands. When these fields are displayed, they
are enclosed in brackets (shown as “[” and “]” in the format given earlier).
- The last field is either a 16-bit unsigned (word) number, or a signed 8-bit
(byte) number. The entry is displayed enclosed in brackets.
- At least one of the last three fields (base register, index register, number) is
displayed for any memory operand.

3. Immediate data is displayed as a byte or word number, without brackets.
   Example:

<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>PREFIX</th>
<th>MNEMONIC</th>
<th>OPERANDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0932:</td>
<td>FF391H</td>
<td></td>
<td>TEST</td>
<td>AL, 07H</td>
</tr>
</tbody>
</table>

4. Labels for the JUMP and CALL instructions:
   - Within 128 bytes of current address—$ ± xxH
   Example:

<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>MNEMONIC</th>
<th>OPERANDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0934:</td>
<td>FF393H</td>
<td>JE</td>
<td>$-06H</td>
</tr>
</tbody>
</table>

   - Within same 64K segment as current address—$ + xxxxH
   Example:

<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>MNEMONIC</th>
<th>OPERANDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000:</td>
<td>FF00H</td>
<td>JMP</td>
<td>$+1005H</td>
</tr>
</tbody>
</table>

   - To a different segment—base:displacement
   Example:

<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>MNEMONIC</th>
<th>OPERANDS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0978:</td>
<td>FFF0H</td>
<td>JMP</td>
<td>FF0:0096H</td>
</tr>
</tbody>
</table>

Note: the first two labels represent “SHORT” (intra-segment) branches, the third
is a “LONG” (inter-segment) branch.

Display of Cycles in Instruction Mode. After the instruction mnemonic and
operands are displayed, the display shows succeeding cycles performed by the
current instruction. Four cycles are shown per line of display; the display uses as
many lines as needed to show all cycles.
The general format for cycles display is:

\[ \text{address-status-data-segment} \]

Examples:

- **Read/Write:**
  
  - 12345H-R-34H-DS  
    (8-bit read of data 34H from address 12345H using DS)
  
  - 45100H-W-70H-SS  
    (8-bit write of data 70H to address 45100H using SS)

- **Input/Output:** (no segment register; 16-bit address)
  
  - FFOOH-I-01H  
    (8-bit input of data 01H from port FFOOH)
  
  - FFDAH-O-34H  
    (8-bit output of data 34H to port FFDAH)

- **Interrupt Acknowledge:** no address field, “A” for “acknowledge” status, 8-bit interrupt type.

Example:

<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>PREFIX</th>
<th>MNEMONIC</th>
<th>OPERANDS</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0971:</td>
<td>FF391H</td>
<td>TEST</td>
<td>AL, 07H</td>
<td>; SHORT</td>
<td></td>
</tr>
<tr>
<td>0977:</td>
<td>FF393H</td>
<td>JE</td>
<td>$06H</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0986:</td>
<td>FF38DH</td>
<td>MOV</td>
<td>DX, FFEAH</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0995:</td>
<td>FF390H</td>
<td>IN</td>
<td>AL, DX</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note that:

a. The I cycle is part of the IN instruction; the rest of the cycles are the interrupt.

b. The “A” cycle is traced twice; ignore the first one.

c. Interrupt is type 0FFH

d. The five cycles after the “A” cycle are as follows:
   - Read IP of interrupt vector
   - Read CS of interrupt vector
   - Write flags to stack
   - Write old CS to stack
   - Write old IP to stack

- *Fetch* cycles do not appear as cycles; they are used to display the opcode mnemonic and operands.

- *Halt* cycles never appear as cycles; they appear as the mnemonic HLT.

**Gaps in Trace in Instruction Mode.** In Instruction mode, a gap in trace data is shown as three asterisks (**`). A gap in trace is produced by tracepoints or by buffer overflow.

A gap in trace data also is reflected by a MARK = 1 in Frames mode.
Extended Example of Trace Displays

The following example (from SDK-86 Monitor) shows most of the features of trace displays discussed in this section.

*ONTRACE=FF00:96
*OFFTRACE=FFFF:0
*ONT
*OFFT
OFFT=FFFF0H A,I,O,H,F,R,W
*ENABLE TRACE CONDITIONALLY NOW ON
*GO TILL FF00:9F EXECUTED
EMULATION BEGUN
EMULATION TERMINATED, CS:IP=FF00:00A1H
*BUF
BUF=03FDH
*P-20

<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>PREFIX</th>
<th>MNEMONIC</th>
<th>OPERANDS</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0966:</td>
<td>FF09CH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>0977:</td>
<td>FF09CH</td>
<td>F</td>
<td>S</td>
<td>1</td>
<td>Q</td>
</tr>
<tr>
<td>0988:</td>
<td>FF08CH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>0999:</td>
<td>FF02CH</td>
<td>F</td>
<td>R</td>
<td>N</td>
<td>1</td>
</tr>
<tr>
<td>1000:</td>
<td>FF007H</td>
<td>R</td>
<td>N</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>1001:</td>
<td>FF093H</td>
<td>R</td>
<td>N</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1002:</td>
<td>FF000H</td>
<td>R</td>
<td>N</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>1003:</td>
<td>FF09DH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1004:</td>
<td>FF09EH</td>
<td>F</td>
<td>S</td>
<td>1</td>
<td>Q</td>
</tr>
<tr>
<td>1005:</td>
<td>FF09EH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>1006:</td>
<td>FF09FH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1007:</td>
<td>FF09FH</td>
<td>F</td>
<td>S</td>
<td>1</td>
<td>Q</td>
</tr>
<tr>
<td>1008:</td>
<td>FF08BH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>D</td>
</tr>
<tr>
<td>1009:</td>
<td>FF08BH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1010:</td>
<td>FF08CH</td>
<td>F</td>
<td>F</td>
<td>1</td>
<td>Q</td>
</tr>
<tr>
<td>1011:</td>
<td>FF0A8H</td>
<td>F</td>
<td>F</td>
<td>1</td>
<td>Q</td>
</tr>
<tr>
<td>1012:</td>
<td>FF0A8H</td>
<td>F</td>
<td>F</td>
<td>1</td>
<td>Q</td>
</tr>
<tr>
<td>1013:</td>
<td>FF0ECH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>D</td>
</tr>
</tbody>
</table>

*TRANCE TURNED OFF AT FFFF:0, BACK ON AT FF00:96, BREAK AT FF00:9F
*BRO
BRO=FF07EH E
*ONT
*OFFT
OFFT=FFFF0H A,I,O,H,F,R,W
*TRA=FRA
*P-25

<table>
<thead>
<tr>
<th>FRAME</th>
<th>ADDR</th>
<th>STS</th>
<th>QSTS</th>
<th>QDEPTH</th>
<th>DMUX</th>
<th>MARK</th>
</tr>
</thead>
<tbody>
<tr>
<td>0966:</td>
<td>FF09CH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>0977:</td>
<td>FF09CH</td>
<td>F</td>
<td>S</td>
<td>1</td>
<td>Q</td>
<td>0</td>
</tr>
<tr>
<td>0988:</td>
<td>FF08CH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>0999:</td>
<td>FF02CH</td>
<td>F</td>
<td>R</td>
<td>N</td>
<td>1</td>
<td>A</td>
</tr>
<tr>
<td>1000:</td>
<td>FF007H</td>
<td>R</td>
<td>N</td>
<td>1</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>1001:</td>
<td>FF093H</td>
<td>R</td>
<td>N</td>
<td>1</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>1002:</td>
<td>FF000H</td>
<td>R</td>
<td>N</td>
<td>1</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>1003:</td>
<td>FF09DH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>1004:</td>
<td>FF09EH</td>
<td>F</td>
<td>S</td>
<td>1</td>
<td>Q</td>
<td>0</td>
</tr>
<tr>
<td>1005:</td>
<td>FF09EH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>1006:</td>
<td>FF09FH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>1007:</td>
<td>FF09FH</td>
<td>F</td>
<td>S</td>
<td>1</td>
<td>Q</td>
<td>0</td>
</tr>
<tr>
<td>1008:</td>
<td>FF08BH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>1009:</td>
<td>FF08BH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>1010:</td>
<td>FF08CH</td>
<td>F</td>
<td>F</td>
<td>1</td>
<td>Q</td>
<td>0</td>
</tr>
<tr>
<td>1011:</td>
<td>FF0A8H</td>
<td>F</td>
<td>F</td>
<td>1</td>
<td>Q</td>
<td>0</td>
</tr>
<tr>
<td>1012:</td>
<td>FF0A8H</td>
<td>F</td>
<td>F</td>
<td>1</td>
<td>Q</td>
<td>0</td>
</tr>
<tr>
<td>1013:</td>
<td>FF0ECH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Command</td>
<td>Mode</td>
<td>QNT</td>
<td>CNT</td>
<td>Command</td>
<td>Mode</td>
</tr>
<tr>
<td>---</td>
<td>-----------</td>
<td>------</td>
<td>-----</td>
<td>-----</td>
<td>-----------</td>
<td>------</td>
</tr>
<tr>
<td>1014</td>
<td>FF0A1H</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>1015</td>
<td>2F0A9H</td>
<td>F</td>
<td>S</td>
<td>1</td>
<td>Q</td>
<td>0</td>
</tr>
<tr>
<td>1016</td>
<td>2F02EH</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>1017</td>
<td>FF0A2H</td>
<td>F</td>
<td>N</td>
<td>1</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>1018</td>
<td>2F08EH</td>
<td>F</td>
<td>N</td>
<td>2</td>
<td>D</td>
<td>0</td>
</tr>
<tr>
<td>1019</td>
<td>FF0A3H</td>
<td>F</td>
<td>N</td>
<td>2</td>
<td>A</td>
<td>0</td>
</tr>
<tr>
<td>1020</td>
<td>2F01EH</td>
<td>F</td>
<td>N</td>
<td>3</td>
<td>D</td>
<td>0</td>
</tr>
</tbody>
</table>
Set TRACE Display Mode Command

\[
\text{TRACE} = \begin{cases} 
\text{FRAME} \\
\text{INSTRUCTION}
\end{cases}
\]

Examples:

\[
\begin{align*}
\text{TRACE} &= \text{FRAME} \\
\text{TRACE} &= \text{INSTRUCTION}
\end{align*}
\]

<table>
<thead>
<tr>
<th>TRACE</th>
<th>A command keyword indicating that the mode of display for trace data is to be set.</th>
</tr>
</thead>
<tbody>
<tr>
<td>FRAME</td>
<td>A function keyword indicating that data in the trace buffer is to be displayed frame by frame.</td>
</tr>
<tr>
<td>INSTRUCTION</td>
<td>A function keyword indicating that data in the trace buffer is to be displayed by instruction. Each instruction is equivalent to one or more machine cycles.</td>
</tr>
</tbody>
</table>

In the FRAME mode, trace data is displayed one frame per line, with fields for frame number, address/data, bus status, queue status, queue depth, type of frame (address, data or queue) and start/stop trace marker.

In the INSTRUCTION mode, trace is disassembled with instructions appearing as 8086 assembler mnemonics. All other cycle data other than instruction fetches, the address, status and data of the cycle are displayed. Memory fetches into the execution queue and queue activity are not shown explicitly. Instead, they are used to find the instruction bytes that were executed when the instruction is taken from the queue. Whenever it is impossible to disassemble frames, immediately before or after a frame with the START/STOP trace marker set, the gap is indicated by a line containing three asterisks ("***"). In either mode, status appears as "F," "R," "W," "I," "O," "H," or "A" corresponding to the match status(es) set in the tracepoint register, and addresses are displayed as 20-bit numbers is the displacement of the address from that base.
**ENABLE/DISABLE TRACE Command**

1. **ENABLE TRACE** [CONDITIONALLY [NOW ON NOW OFF]]
2. **DISABLE TRACE**

**Examples:**

- ENABLE TRACE
- ENABLE TRACE CONDITIONALLY
- ENABLE TRACE CONDITIONALLY NOW ON
- ENABLE TRACE CONDITIONALLY NOW OFF
- DISABLE TRACE

**ENABLE**
A command keyword that causes trace data collection to be conditionally or unconditionally enabled.

**DISABLE**
A command keyword that causes trace data collection to be disabled.

**CONDITIONALLY**
A command modifier that specifies that trace will be turned on whenever the ONTRACE register matches and turned off whenever the OFFTRACE register matches.

**NOW ON**
Indicates that trace is turned on for the beginning of the next emulation (see Note).

**NOW OFF**
Indicates that trace is turned off for the beginning of the next emulation (see Note).

**TRACE**
Command modifier denoting that trace is to be enabled/disabled.

**NOTE**

If **ENABLE TRACE CONDITIONALLY**, the tracepoints will inadvertently match and turn trace on or off when entering emulation if the tracepoint is set to match on a Fetch at address 00008 or 00009, and when exiting emulation if the tracepoint is set to match on a Read at address 00008 or 00009. Conditional trace should not be set-up ONTRACE/OFFTRACE tracepoints at memory locations 00008 or 00009 as the ICE-88 emulator uses these two memory locations when emulation is broken.

If neither **NOW ON** or **NOW OFF** is selected (i.e., **ENABLE TRACE CONDITIONALLY**), trace is left in its current state.
Display TRACE Command

TRACE

Example:

TRACE

TRACE

A command keyword that, if entered from the keyboard as a single token, causes the current TRACE mode (FRA for FRAME or INS for INSTRUCTIONS) to be displayed.

MOVE, OLDEST, and NEWEST Commands

MOVE [[+::−]decimal]

OLDEST

NEWEST

Example:

MOVE
MOVE +6
MOVE −11
OLDEST
NEWEST

MOVE

A command keyword that moves the buffer pointer one or more entries forward (toward the most recent entries) or backward (toward the earliest entries). An entry is a frame or instruction, depending on the TRACE mode in effect.

+ A unary operator specifying a forward movement. Plus is the default.

− A unary operator specifying a backward movement.

decimal A number, evaluated in decimal radix (if no explicit suffix is given), that gives the number of entries to be included in the MOVE.

OLDEST A command keyword that moves the pointer to the earliest entry in the buffer.

NEAREST A command keyword that moves the pointer to the latest entry in the buffer.
PRINT Command

1. PRINT ALL
2. PRINT \([+\ :-\]\)decimal\]

Example:

```
PRINT
PRINT ALL
PRINT +5
PRINT 5
PRINT -10
```

PRINT A command keyword calling for a display of one or more entries from the trace data buffer. The entries are displayed as frames or instructions, depending on the current trace mode.

ALL A function keyword indicating that the entire trace buffer contents are to be displayed.

+ A unary operator directing the display of decimal entries below (entered later than) the current buffer pointer location. See DISCUSSION (page 6-24) for details. Plus is the default.

− A unary operator directing the display of decimal-number entries above (entered earlier than) the current buffer pointer location. See DISCUSSION (page 6-24) for details.

decimal A numeric constant, evaluated in decimal suffix, giving the number of entries to be displayed.
Chapter 7 contains discussions, examples and syntax summaries for each of the ICE-88 interrogation and utility commands.

The following brief outline of Chapter 7 shows how the interrogation and utility commands have been classified.

Utility Commands Involving ISIS-II
- ICE88 Command
- EXIT Command
- LOAD Command
- SAVE Command
- LIST Command

Number Bases and Radix Commands
- Set or Display Console Input Radix Commands
- Set or Display Console Output Radix Commands

Hardware Register Commands
- Set Register Command
- RESET HARDWARE Command

Memory Mapping Commands
- MAP DISK Command
- MAP INTELLEC Command
- Set MAP Status Command
- Display MAP Status Command
- RESET MAP Command

Set Memory and Port Content Commands
- Set Memory Command
- Set Input/Output Port Command

Symbol Table and Statement Number Table Commands
- DEFINE Symbol Command
- Display Symbols Command
- Display Statement Numbers Command
- Display Modules Command
- Change Symbol Command
- REMOVE Symbols Command
- REMOVE Modules Command
- TYPE Command
- Set DOMAIN Command
- RESET DOMAIN Command

Display Commands
- Display Processor and Status Registers Command
- Display Memory Command
- Display I/O Command
- Display STACK Command
- Display Boolean Command
- Display NESTING Command
- Evaluate Command
Utility Commands Involving ISIS-II

The Intel Systems Implementation Supervisor (ISIS-II) is the diskette operating system for the Intellec Microcomputer Development System. The ICE-88 emulator runs under ISIS-II control, and can call upon ISIS-II for file management function.

The following commands are included in this section:

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICE88</td>
<td>Load ICE-88 program from diskette.</td>
</tr>
<tr>
<td>EXIT</td>
<td>Return control to ISIS-II.</td>
</tr>
<tr>
<td>LOAD</td>
<td>Load user program into memory accessed by ICE-88 emulator.</td>
</tr>
<tr>
<td>SAVE</td>
<td>Copy user program from memory onto diskette.</td>
</tr>
<tr>
<td>LIST</td>
<td>Copy ICE-88 emulation output to printer or file.</td>
</tr>
</tbody>
</table>

Discussion

ICE-88 commands can use ISIS-II *pathnames* to direct ISIS-II to a desired diskette file or other output device.

For diskette files, the format of *pathname* is as follows.

`:drive:filename`

The entry *:drive:* stands for one of the references to ISIS disk drives, as follows.

`:F0:` drive 0  
`:F1:` drive 1  
`:F2:` drive 2  
`:F3:` drive 3  
`:F4:` Single-density or double density drives on a double-density system  
`:F5:`  
`:F6:`  
`:F7:`

The entry *filename* must follow the second colon (after *drive*) without any intervening spaces. A filename has the following components.

*identifier [extension]*

The entry *identifier* is a name assigned by the user, and is made up of one to six alphanumeric characters. The *extension* is an optional part of the filename, consisting of one to three alphanumeric characters preceded by a single period. The extension must be used if it is present in the directory listing of the file on the diskette. If used, the extension follows the identifier without any spaces. Some extensions (e.g., .BAK) are assigned by system processors; others can be assigned at the desire of the user. An extension provides a second level of file identification; it can be used to identify different versions of the same program, or to give supplemental information about the file (e.g., author, data, version).

Fully compiled or assembled programs ready to run (emulate) do not have system-assigned extensions, although they may have extensions assigned by the user.

For devices other than diskette files, the format of *pathname* is as follows.

`:device:`
The following devices are commonly accessed in ICE-88 commands.

:Device: Output Device

:LP: Line printer
:HP: High-speed paper tape punch
:TO: Teletypewriter printer
:CO: Console display

For more information on ISIS-II filenames and device codes, refer to the ISIS-II System User’s Guide.

The ICE88 command, entered after an ISIS-II prompt, directs ISIS-II to load the ICE-88 program from the specified diskette drive, into a reserved area in Intellec memory. The ICE-88 emulator begins operation as soon as it is loaded, initializing its hardware and program variables, and signaling readiness to accept ICE-88 commands by displaying an asterisk prompt.

NOTE
Inspect disk containing ICE-88 program prior to loading into the disk drive. If the diskette contains a write protect tab, remove the tab to write-enable the disk. If the ICE-88 program is loaded from a disk containing a write-protect tab, an ISIS ERR 24 (write protect) will result.

The EXIT command ends the emulation session and returns control to ISIS-II. The command issues a hardware reset before exiting.

The LOAD command loads the object code from the named file and drive into the areas of memory specified by the memory map. Modules are loaded in the order of their appearance in the source file. The modules names, symbols, and statement numbers are placed in reserved areas of Intellec memory. Symbols and statement numbers are grouped into tables by module, in the order in which they appear. Both a base value and a displacement value are loaded for all symbols and statement numbers. Any symbol that has no type information is given no type specification in the symbol table. If no exclude modifiers are included in the command, module names are loaded into the ICE-88 module table in the order in which they appear following any module names already in the table, symbols and their types (if present) local to each module are loaded into that module’s symbol table in the order in which they appear, and statement numbers local to each module are loaded into that module’s statement table (for PL/M 86 programs) in the order in which they appear.

The command can include one or more exclude modifiers to control what is to be loaded. If NOCODE is included, the program code is omitted from the load, e.g., it is already in ROM. If NO LINE is included, the program statement number table is not loaded. If NOSYMBOL is included, the program symbol table is not loaded. Any combination of one, two, or three exclusion modifiers may be included, although the command with all three modifiers represents a “null” command. No modifier may be named twice in the same load command.

The SAVE command copies the user program currently loaded from memory onto the specified file and drive. If the diskette installed on the given drive does not have the named file in its directory, ISIS-II creates the file and opens it for write. If the named file does exist on the diskette, the file is overwritten and the previous contents are lost. If no explicit drive number is given, drive 0 is assumed.
The command can include one or more modifiers to control what is to be saved. If NOSYMBOL is included, the symbol table is not copied from memory to diskette. If NOLINE is included, the statement number table (for PL/M-86 programs) is not saved. The modifiers NOCODE and partition are mutually exclusive: if one is used, the other may not be included. If NOCODE is included, the program object code is not copied to diskette. If partitions are included, only the code stored in the memory addresses in the partitions (ranges of addresses) are saved. If neither NOCODE nor Partition appears in the command, any code between the lower and highest addresses in each 1K segment that has been previously loaded is saved. If no code has been loaded, no code is saved. When more than one modifier is used, separate them with spaces. No modifier may be used twice in the same SAVE command. The SAVE operation does not alter the program code, symbol table, or statement number table in memory.

The LIST command saves a record of the emulation session, including high-volume data such as trace data, on a hard-copy device or on a diskette file in addition to sending it to the console. Only one device or file other than the console can be specified (active) at a given time.

The initial device is :CO:, output to the console. Other devices that can be specified are a line printer (:LP:), high-speed paper tape punch (:HP:), and teletypewriter printer (:TO:).

Instead of a hard-copy device, a diskette file can be specified. If the output is to a diskette file, the file is opened when the LIST command is invoked, and output is stored from the beginning of the file, writing over any existing data. Specifying a new file or device in a later LIST command closes any existing open file and avoids over-writing any more data. Specifying the same file in a later LIST causes the delete of the file and starting over.

When LIST is in effect (with a device or file other than :CO:), all output from the ICE·88 emulator, including system prompts, commands, and error messages, is sent both to the named device or file and to the console display. To restore output to the console only (no other device), use the command LIST :CO:.
ICE88 Command

[:drive:]ICE88 [WORKFILES(:alt. drive:)]

Examples:

:F1:ICE88
:F1:ICE88 WORKFILES(:F2:)

:drive: The number of the diskette drive containing the ICE-88 software diskette. The number is preceded by the letter F, and enclosed in colons. This drive is also the default drive for the ICE temporary workfile.

ICE88 The name of the ICE-88 program file under ISIS-II. The filename follows the second colon without any intervening spaces.

WORKFILES Control keyword specifying that an alternate disk drive is to be used for the ICE temporary workfile.

:alt. drive: The number of the diskette drive containing the diskette where the temporary workfile is to be stored. The number is preceded by the letter F, and enclosed in colons.

NOTE
Inspect the diskette containing the ICE-88 program prior to loading into the diskette drive. If the WORKFILES control is specified, the diskette containing the ICE-88 program may be write protected. If the WORKFILES control is not specified, the diskette containing the ICE-88 program must not be write protected or an ISIS ERR24 (write protect) will result. In this case, if the diskette contains a write protect tab, remove the tab to write-enable the diskette.
EXIT Command

EXIT

Example:

EXIT

EXIT  A command keyword that returns control from the ICE-88 emulator to ISIS-II. The command issues a hardware reset before exiting, and leaves the file used for DISK-mapped memory intact.
LOAD Command

```
LOAD[:drive:]filename [NOCODE NOSYMBOL NOLINE]
```

Examples:

```
LOAD :F0:TEST.VR1
LOAD :F1:MYPROG NOLINE
LOAD :F2:COUNT. ONE NOCODE NOLINE
LOAD :F3:NEWCOD NOSYMBOL
```

LOAD: A command keyword that loads the software on the given file and drive into the combination of prototype and Intellec memory specified by a previous MAP command.

drive: The diskette drive (:F0:, :F1:, :F2:, or :F3:) that contains the target file. If no drive is given, :F0: (drive 0) is the default.

filename: The name of the desired program as compiled or assembled, linked, and located. The filename follows the second colon with no intervening spaces.

NOCODE: A modifier specifying that program code is not to be loaded.

NOSYMBOL: A modifier specifying that the program symbol table is not to be loaded.

NOLINE: A modifier specifying that the program line number table (for PL/M-86 programs) is not to be loaded.
SAVE Command

SAVE [[:drive:]filename NOCODE:[:partition [,partition ]...] NOSYMBOL NOLINE

Examples:

SAVE :F1:TEST
SAVE :F0:MYPROG 0800 TO 0FFF NOLINE
SAVE :F2:COUNT.TWO NOLINE NOSYMBOL
SAVE :F3:NEWSYM NOCODE NOLINE
SAVE :F1:TEST #1 TO #50,..SUBR #1 TO ..SUBR #20

SAVE The command keyword that directs the ICE-88 emulator to write the designated software elements to the indicated file and drive.

:drive: The diskette drive (:F0:, :F1:, :F2:, :F3:) holding the diskette that is to contain the saved software. If no explicit drive number is given, drive 0 is the default.

filename The name of the file that is to receive the saved information. The name of the file, including the extension if present, must follow the rules for naming files under ISIS-II. The filename immediately follows the second colon. If the filename does not exist on the designated diskette, ISIS-II creates the file and opens it for write. If it does exist, the current file is destroyed.

NOCODE A modifier specifying that program code is not to be saved.

partition A construct specifying a range of one or more contiguous locations in memory; the contents of the specified locations are saved, but code in other locations is not copied.

NOSYMBOL A modifier specifying that the symbol table is not to be saved to diskette.

NOLINE A modifier specifying that the line number table (for PL/M-86 programs) is not to be saved.
LIST Command

(a) LIST :device:
(b) LIST [:drive]filename

Examples:

LIST :LP:
LIST :CO:
LIST :F1:ICEFIL

LIST: The command keyword directing all ICE-88 emulator output to be sent to the specified device or file, and to the console.

:device: An ISIS-II device code, indicating a hard-copy output device to receive the output.

:drive: The diskette drive holding the diskette on which output is to be written. If no explicit drive is given, drive 0 is assumed.

filename The name of the file on the target diskette. The filename immediately follows the second colon, without intervening spaces.
Number Bases and Radix Commands

ICE-88 commands and displays involve several different number bases (*radixes*). This section describes the various radices used by the ICE-88 emulator and the commands used to control some of them. Most radices are set by the ICE-88 emulator and cannot be changed, but others are under your control.

This section gives details on the following commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUFFIX</td>
<td>Set or display console input radix.</td>
</tr>
<tr>
<td>BASE</td>
<td>Set or display console output radix.</td>
</tr>
</tbody>
</table>

**Discussion**

The commands given in detail in this section refer to the radices used for console input and console output.

**Console Input Radixes; SUFFIX Command**

Any number entered from the console can include an *explicit* input radix. An explicit input radix consists of one of the following alphabet characters appended directly to the number as entered.

<table>
<thead>
<tr>
<th>Explicit Radix</th>
<th>Example</th>
<th>Radix Specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>1001Y</td>
<td>Binary (base 2)</td>
</tr>
<tr>
<td>Q, O</td>
<td>11Q</td>
<td>Octal (base 8)</td>
</tr>
<tr>
<td>T</td>
<td>9T</td>
<td>Decimal (base 10)</td>
</tr>
<tr>
<td>H</td>
<td>9H</td>
<td>Hexadecimal (base 16)</td>
</tr>
<tr>
<td>K</td>
<td>3K</td>
<td>Multiple of 1024 decimal</td>
</tr>
</tbody>
</table>

The *implicit* input radix is the base used by the ICE-88 emulator to interpret numbers entered from the console without an explicit radix.

To display the current implicit input radix, enter the command token SUFFIX followed by a carriage return. The implicit input radix can be Y, Q, T, or H, as defined earlier. The initial implicit radix is hexadecimal.

You can change the implicit input radix by entering a command with the form SUFFIX = *suffix*, where *suffix* is any of the characters Y, Q, O, T, or H. This SUFFIX command can be used where several numbers are to be entered in the same radix.

Note that K (multiple of 1024) cannot be specified as an implicit input radix.

For some kinds of entries from the console, the implicit input radix is always decimal (T). Entries with implicit decimal radix are:

- Numbers entered after MOVE and PRINT keywords.
- Program statement numbers.
- Value in COUNT command.
- Timeout value in RW_TIMEOUT
- Segment numbers in MAP.

An explicit radix always takes precedence over the implicit radix. If the digits in the number entered cannot be interpreted in either the explicit or the implicit radix, an error message is displayed.
Console Output Radixes; BASE Command

Numeric information such as memory and register contents, and data, is displayed in the current console output radix. The console output radix can be one of the following.

<table>
<thead>
<tr>
<th>Output Radix</th>
<th>Radix Specified</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Binary</td>
</tr>
<tr>
<td>Q, O</td>
<td>Octal</td>
</tr>
<tr>
<td>T</td>
<td>Decimal</td>
</tr>
<tr>
<td>H</td>
<td>Hexadecimal</td>
</tr>
<tr>
<td>ASCII</td>
<td>ASCII character for each byte</td>
</tr>
</tbody>
</table>

The initial output radix is hexadecimal (H).

To display the current console output radix, enter the command token BASE followed by carriage return. The display consists of a single character, Y, Q, T, H, or A (for ASCII).

You can change the console output radix by entering a command with the form BASE = base, where base is one of the single characters Y, Q, O, T, or H, or the token ASCII. Once the radix is set with a BASE command, it stays in effect until another BASE command is entered.
Set or Display Console Input Radix Commands

SUFFIX
SUFFIX = Y::Q::O::T::H

Examples:

SUFFIX
SUFFIX = Y

SUFFIX: A command keyword referring to the implicit console input radix. The token SUFFIX alone displays the current setting (Y, Q, T, or H).

= The assignment operator, indicating that the new setting is to follow.

Y Binary radix.
Q, O Octal radix.
T Decimal radix.
H Hexadecimal radix.
Set or Display Console Output Radix Commands

BASE

BASE = Y::Q::O::T::H::ASCII

Examples:

BASE

BASE = Q

BASE = A command keyword referring to the system console output radix. The token BASE alone displays the current setting (Y, Q, T, H, or A).

= The assignment operator, indicating that the new setting is to follow.

Y Binary radix.

Q, O Octal radix.

T Decimal radix.

H Hexadecimal radix.

ASCII Each byte represented by its corresponding ASCII character, without separators.
Hardware Register Commands

This section presents the keywords used in the ICE-88 emulator to refer to the following types of hardware registers and signals.

- 8088 Processor Register
- 8088 Status Flags
- ICE-88 Status Registers
- 8088 Pin Signals

The following commands that refer to hardware registers and signals are discussed in this section.

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Register</td>
<td>Set (change) the contents of any of the writeable 8088 registers.</td>
</tr>
<tr>
<td>RESET HARDWARE</td>
<td>Reset ICE-88 hardware to initial state.</td>
</tr>
</tbody>
</table>

Discussion

Tables 7-1 through 7-6 show the tokens used to refer to any 8088 8-bit register, 16-bit register or pin signal.

- Meta-term: Class of tokens
  - general-register: 16-bit and 8-bit work registers
  - pointer-register: 16-bit pointer registers
  - index-register: 16-bit index registers
  - segment-register: 16-bit segment reference registers
  - status-register: 8- and 16-bit status registers
  - pin-reference: 8088 pin signals
  - flag-reference: 8088 status flags

Table 7-1. 8088 General Registers

<table>
<thead>
<tr>
<th>Reference</th>
<th>8088 Register and Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>RAX</td>
<td>16-bit Accumulator</td>
</tr>
<tr>
<td>RAH</td>
<td>High 8 bits of Accumulator</td>
</tr>
<tr>
<td>RAL</td>
<td>Low 8 bits of Accumulator</td>
</tr>
<tr>
<td>RBX</td>
<td>16-bit Base Register</td>
</tr>
<tr>
<td>RBH</td>
<td>High 8 bits of Base Register</td>
</tr>
<tr>
<td>RBL</td>
<td>Low 8 bits of Base Register</td>
</tr>
<tr>
<td>RCX</td>
<td>16-bit Count Register</td>
</tr>
<tr>
<td>RCH</td>
<td>High 8 bits of Count Register</td>
</tr>
<tr>
<td>RCL</td>
<td>Low 8 bits of Count Register</td>
</tr>
<tr>
<td>RDX</td>
<td>16-bit Data Register</td>
</tr>
<tr>
<td>RDH</td>
<td>High 8 bits of Data Register</td>
</tr>
<tr>
<td>RDL</td>
<td>Low 8 bits of Data Register</td>
</tr>
</tbody>
</table>
### Table 7-2. Pointer Registers

<table>
<thead>
<tr>
<th>Reference</th>
<th>8088 Register and Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SP</td>
<td>16-bit Stack Pointer</td>
</tr>
<tr>
<td>BP</td>
<td>16-bit Base Pointer</td>
</tr>
</tbody>
</table>

### Table 7-3. Index Registers

<table>
<thead>
<tr>
<th>Reference</th>
<th>8088 Register and Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SI</td>
<td>16-bit Source Index</td>
</tr>
<tr>
<td>DI</td>
<td>16-bit Destination Register</td>
</tr>
</tbody>
</table>

### Table 7-4. Segment Registers

<table>
<thead>
<tr>
<th>Reference</th>
<th>8088 Register and Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS</td>
<td>16-bit Code Segment Register</td>
</tr>
<tr>
<td>DS</td>
<td>16-bit Data Segment Register</td>
</tr>
<tr>
<td>SS</td>
<td>16-bit Stack Segment Register</td>
</tr>
<tr>
<td>ES</td>
<td>16-bit Extra Segment Register</td>
</tr>
</tbody>
</table>

### Table 7-5. Status Registers

<table>
<thead>
<tr>
<th>Reference</th>
<th>Register and Interpretation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IP</td>
<td>16-bit Instruction Pointer Register</td>
</tr>
<tr>
<td>RF</td>
<td>16-bit Flag Register</td>
</tr>
<tr>
<td>CAUSE (Read only)</td>
<td>8-bit Cause of last break in emulation</td>
</tr>
<tr>
<td>OPCODE (Read only)</td>
<td>8-bit Previous opcode executed</td>
</tr>
<tr>
<td>PIP (Read only)</td>
<td>16-bit Previous Instruction Pointer Register</td>
</tr>
<tr>
<td>TIMER (Read only)</td>
<td>Low 16 bits of emulation timer</td>
</tr>
<tr>
<td>HTIMER (Read only)</td>
<td>High 16 bits of emulation timer</td>
</tr>
<tr>
<td>BUFFERSIZE (Read only)</td>
<td>16-bit trace buffer size Register</td>
</tr>
<tr>
<td>LOWER (Read only)</td>
<td>16-bit Intellec address register</td>
</tr>
<tr>
<td>UPPER (Read only)</td>
<td>16-bit Intellec address register (displayed in decimal only)</td>
</tr>
</tbody>
</table>
Table 7-6. Pin References

<table>
<thead>
<tr>
<th>reference</th>
<th>8088 Pin (Read only)</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDY</td>
<td>READY</td>
</tr>
<tr>
<td>NMI</td>
<td>NMI</td>
</tr>
<tr>
<td>TEST</td>
<td>TEST</td>
</tr>
<tr>
<td>HOLD</td>
<td>HOLD</td>
</tr>
<tr>
<td>RST</td>
<td>RESET</td>
</tr>
<tr>
<td>MN</td>
<td>MN/MN (minimum/maximum configuration)</td>
</tr>
<tr>
<td>IR</td>
<td>INTR</td>
</tr>
</tbody>
</table>

Table 7-7. Flag References

<table>
<thead>
<tr>
<th>reference</th>
<th>Flag</th>
</tr>
</thead>
<tbody>
<tr>
<td>AFL</td>
<td>Auxiliary-carry out of low byte to high byte</td>
</tr>
<tr>
<td>CFL</td>
<td>Carry or borrow out of high byte</td>
</tr>
<tr>
<td>DFL</td>
<td>Direction of string manipulation instruction</td>
</tr>
<tr>
<td>IFL</td>
<td>Interrupt-enable (external)</td>
</tr>
<tr>
<td>OFL</td>
<td>Overflow flag in signed arithmetic</td>
</tr>
<tr>
<td>PFL</td>
<td>Parity</td>
</tr>
<tr>
<td>SFL</td>
<td>Sign of the result of an operation</td>
</tr>
<tr>
<td>TFL</td>
<td>Trap used to place processor in single step mode for debug</td>
</tr>
<tr>
<td>ZFL</td>
<td>Zero indicates a zero value result of an operation</td>
</tr>
</tbody>
</table>

To set (change) the content of one of the processor registers, use a command with the form:

\[
\text{reference} = \text{expr}
\]

Where \( \text{expr} \) is a numerical constant or numerical expression giving the desired new contents. Each of the registers that can be changed with this command has a definite size (16, 8, or 1 bits). If the new contents represents fewer bits than the destination register, the bits are right-justified in the register, and the remaining bits are set to zero. In other words the ICE-88 emulator assumes that the quantity represents the lowest-order bits, and sets any unspecified high-order bits to zero.

The RESET HARDWARE command is used to restore the ICE-88 hardware to the initial program load condition. One use for this command might be to reset the hardware when reconfiguring. The EXIT command includes the RESET HARDWARE function.
When a RESET HARDWARE command is issued, the ICE-88 emulator attempts to
reset the hardware without disturbing any internal controls or user specified setup
(breakpoints, map, or enable/disable trace conditionally). If this attempt is suc-
cessful, the ICE-88 emulator returns a prompt to the user immediately:

*CLOCK = INTERNAL
*RESET HARDWARE ;This reset will not encounter any problems.
* ;Prompt appears immediately, no internal setup
* ;affected.

The user may now continue his processing.

If, however, this attempt is unsuccessful, the ICE-88 emulator will warn the user
that it is beginning to reinitialize the hardware. It then attempts to reset the hard-
ware by using initialization procedures and then restoring internal controls and
setup to the pre-reset state (trace, however, cannot be restored to its previous state,
and will be enabled unconditionally). Although no internal registers or flags are
changed, the integrity of user memory, ICE memory, and current disk-mapped
memory cannot be guaranteed. If this attempt to reset the hardware is successful,
the ICE-88 emulator will return a warning to the user that the hardware has been
initialized followed by a prompt. The user may then examine memory for integrity,
reload memory or continue processing.

NOTE

If the above reinitialization is successful, the ICE-88 emulator may require
up to approximately 30 seconds to fully restore the map.

If the ICE-88 emulator detects an error during this attempt, it will return an error
message and a prompt to the user. No warning that the hardware has been
reinitialized will be returned. The user should attempt to resolve the error condition
and MUST reissue the RESET HARDWARE command. The ICE-88 emulator will
then go through the reinitialization as described above. The following example
illustrates the procedure required to respond to an error condition:

*CLOCK = EXTERNAL ;This command forces an error condition.
ERR 40:NO USER CLOCK ;Error response.
*RESET HARDWARE ;Not able to reset, tries to reinitialize.
WARN C8:REINITIALIZING - FAULT ;Reports an error and no warning that hardware
ERR 40:NO USER CLOCK was reinitialized; thus, reinitialization was not able
to complete. The user must clear the error
* ;condition and then reset the hardware again in
* ;order to get the hardware into a known state.
* ;
*CLO = INT ;Clears the error condition, must reset again.
*RESET HARDWARE ;This reset must clear the incomplete reset from
* ;above.

WARN C8:REINITIALIZING - FAULT ;Note that the user is informed that the hardware
WARN C6: HARDWARE REINITIALIZED ;has been reinitialized and that the user specified
* ;setup (map, breakpoints, etc.) has been restored.
* ;Memory contents (ICE memory, user memory,
* ;and current disk-mapped memory) cannot be
* ;guaranteed. The user must verify the contents or
* ;reload his code and data.
Set Register Command

\[ \text{reference} = \text{contents} \]

Examples:

\[
\begin{align*}
\text{RAX} &= 0000H \\
\text{IP} &= F23AH \\
\text{IP} &= \text{IP} + 1 \\
\text{RDL} &= \text{FFH} \\
\text{CS} &= \text{WORD .SAM}
\end{align*}
\]

- **reference**: The keyword name of any of the writeable registers, as follows:
  - general registers (see table 7-1).
  - pointer registers (see table 7-2).
  - index registers (see table 7-3).
  - segment registers (see table 7-4).
  - status registers (see table 7-5).

- **=**: The assignment operator.

- **contents**: A numeric expression.
RESET HARDWARE Command

RESET HARDWARE

Example:

    RESET HARDWARE

RESET       A command keyword restoring its object to a reset condition.

HARDWARE    A function keyword restoring ICE-88 hardware to the reset condition that occurs after the initial ICE-88 emulator invocation.
Memory Mapping Commands

The commands in this section control ICE-88 memory map. The ICE-88 emulator uses the map to identify what user memory is installed and what types and sizes of memory are being “borrowed” from the ICE-88 emulator for testing purposes.

This section gives details on the following commands.

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAP DISK</td>
<td>Declare which disk file is available for mapping to.</td>
</tr>
<tr>
<td>MAP INTELLEC</td>
<td>Declare which physical Intellec segments are available for mapping memory to.</td>
</tr>
<tr>
<td>Set MAP Status</td>
<td>Assign up to 1024 1K segments of memory locations to USER, ICE, INTELLEC, DISK, or GUARDED status.</td>
</tr>
<tr>
<td>Display MAP Status</td>
<td>Display status of one or more memory segments.</td>
</tr>
<tr>
<td>RESET MAP</td>
<td>Restore the memory map to its initial condition, all GUARDED.</td>
</tr>
</tbody>
</table>

Discussion

A maximum of 1M byte (megabyte or 1,048,576 bytes) locations are accessible with the 20-bit addressing scheme used by the 8088 processor. The MAP commands allow this 1M logical address space to be mapped in 1K-byte segments (on 1K boundaries) to any of (1) physical memory in the user’s system, (2) either of two 1K-byte segments of ICE-88 “real-time” memory, (3) a random-access disk file, (4) any 1K-byte segment in expansion Intellec memory (addresses at 64K or above), or (5) as guarded (i.e., the logical addresses do not physically exist).

The ICE-88 module supplies 2K bytes of high-speed static memory which may be mapped for “real-time” execution. The speed of this memory will allow near real-time operation. The 2K bytes may be mapped in 1K-byte segments into appropriate address space within the 8088’s 1M byte address space.

Disk-based memory and expansion Intellec memory both provide substantially slower execution speeds since all accesses to these memories require additional processing by the ICE-88 emulator. The first 64K of Intellec memory is reserved for system and ICE-88 software. Therefore, any memory assigned to the user must reside in expansion Intellec memory above 64K.

The MAP commands are used to declare, set, display, and reset the ICE-88 memory mapping. The ICE-88 emulator has 1M logical addresses divided into 1024 logical address segments, each starting on a 1K boundary and representing 1K bytes of memory. Each segment is addressed by a decimal segment number, n. The value of n is an integer value between 0 and 1023. For any given segment n, that segment contains addresses nk through nK + 1023. For example, the lowest logical segment in memory space contains addresses 0 through 1023 (0K through 0K + 1023). In a like manner, logical segment 10 would contain addresses 10240 through 11263 (10K through 10K + 1023). All partitions used in MAP commands must contain segment numbers whose values lie between 0 and 1023.
If the diskette is to be used for user memory, the MAP DISK command must be used to declare the disk file to be used. The syntax of the MAP DISK command is:

\[
\text{MAP DISK} = [\text{:drive:}] \text{name}
\]

The command opens the ISIS-II file specified by \text{name}, checks how many physical segments will fit on the diskette, and reports this number to you. MAP DISK may only be declared initially and once following each RESET MAP command.

If Intellec memory is to be used for user memory, the MAP INTELLEC command must be used to specify the physical memory segments in extended Intellec memory to be used by user programs. The syntax of the MAP INTELLEC command is:

\[
\text{MAP INTELLEC} = \text{partition} [, \text{partition}] ...
\]

where \text{partition} is defined as:

\[
\text{partition} = \begin{cases} 
\text{physical-segment-number} & \text{TO physical-segment-number} \\
\text{physical-segment-number} & \text{LENGTH physical-segment-length} 
\end{cases}
\]

This command declares physical memory segments in expanded Intellec memory and checks that the memory physically exists by writing to it and reading back. An error occurs if any \text{partitions} extend below 64K as extended Intellec memory exists only at addresses 64K or above. Therefore the range of \text{physical-segment-number} values is 64-1023 (inclusive), the range of \text{physical-segment-length} values is 1-960 (inclusive). A warning is issued if the memory does not exist. INTELLEC declarations are cumulative between RESET MAP commands. Logical memory segments may not be set to Intellec or disk until the associated MAP INTELLEC or MAP DISK commands have been entered.

\textbf{NOTE}

When mapping to Intellec memory, the monitor prom circuitry does not decode the four high order bits of a 20-bit address. Therefore, addresses of the 20-bit form XXXX 1111 1XXX XXXX XXXX (where X is don't care) will be overshadowed by the monitor and should not be used as addresses to be mapped.

The Set MAP Status command is then used to map logical memory segments to physical segments. The syntax of the Set MAP Status command is:

\[
\text{MAP partition = GUARDED USER \{NOVERIFY\} ICE \{physical-segment-number\} \{NOVERIFY\} INTELLEC \{physical-segment-number\} \{NOVERIFY\} DISK \{physical-segment-number\} \{NOVERIFY\}}
\]

where

\[
\text{partition} = \begin{cases} 
\text{logical-segment-number} & \text{TO logical-segment-number} \\
\text{logical-segment-number} & \text{LENGTH logical-segment-length} 
\end{cases}
\]
The Set MAP Status command sets the memory map by assigning logical segments to physical addresses in memory:

**USER**

The logical segments specified are set to exist in the user's memory at the same physical addresses as those specified in the logical segments.

**ICE**

The logical segments specified are set to exist in ICE-88 "real-time" memory. The partition specified may only contain a maximum of two segments and the physical-segment-number value must be 0 or 1, if one is given.

**INTELLEC**

The logical segments specified in the partition are set to exist in extended Intellec memory starting in the physical segment specified by the physical-segment-number in the command, if one is given.

**DISK**

The logical segments specified in the partition are set to exist on the disk file starting at the physical segment specified by the physical-segment-number in the command, if one is given.

**GUARDED**

All accesses to memory addresses in the segments specified by the input parameter, partition are error conditions. All memory is initially GUARDED.

**NOVERIFY**

Specifies that write-verification will not be performed when using the ICE-88 Change command to change the contents of memory. This is useful when memory-mapped I/O is being used. Whenever a logical memory segment is mapped USER, ICE, INTELLEC, or DISK, it is write-verified unless explicitly NOVERIFY.

When mapping to ICE memory, INTELLEC memory or DISK memory, if a physical-segment-number is given in the command, the first logical segment specified in the partition is set to the physical segment specified by the physical-segment-number and subsequent logical segments are set at subsequent physical segments. If more logical segments are specified than physical segments available, logical segment will be assigned to available physical segments and an error message is displayed indicating that the remaining logical segments are unassigned. If physical-segment-number is omitted in the command, the ICE emulator assigns from unassigned physical segments in the designated memory. If Intellec or disk memory is specified, an error occurs if the resulting physical segment is not one of those previously declared by a MAP INTELLEC or MAP DISK command. An error also occurs if there are insufficient physical segments unassigned to map the specified logical segments.

The Display MAP Status command displays the current setting of the map for the unguarded memory segments specified by partition, if given. If a partition is not given, all the unguarded segments for the entire range from 0 to 1023 are displayed. The display is to the following format. The segments are displayed, four segments per line, each of the form:

```
logical-segment = type physical-segment [N]
```

where:

- **logical-segment** ≡ four decimal digits with T suffix
- **type** ≡ ICE :: USE :: INT :: DIS
- **physical-segment** ≡ four decimal digits with T suffix for ICE, INT, DIS and four blanks for USE
Example 1.

MAP 0 TO 3

Display:

0000T = USE 0001T = ICE 0002T = INT 0064T 0003 = DIS 0000T
0004T = DIS 0001T = ICE 0002T = INT 0064T 0003 = DIS 0000T

0000T = USE 0001T = ICE 0002T = INT 0064T 0003 = DIS 0000T
0004T = DIS 0001T = ICE 0002T = INT 0064T 0003 = DIS 0000T

1023T = DIS

The RESET MAP sets the memory map to its initial condition, all GUARDED, and "undeclares" the DISK and INTELLEC memory available. Multiple logical segments may be mapped to the same physical segment. Reassignment of a logical segment to a different physical segment causes the physical segment originally mapped to become "guarded," or unused, and reassignable, providing no other logical segment is currently mapped to it.
MAP DISK Command

MAP DISK = [:drive:]filename

Examples:

MAP DISK = :F0: MYPROG
MAP DISK = :F1: TEST1
MAP DIS = TEST2

MAP A command keyword referring to some operation on the ICE-88 memory map.

DISK A command modifier that specifies that an ISIS-II disk file is to be opened and available for mapping user memory to.

:drive: The disk drive (:F0:, :F1:, :F2:, or :F3:) that contains the target file. If no drive is specified, :F0: (drive 0) is the default.

filename The name of the desired disk file that is to be opened. The filename follows the second colon with no intervening spaces.

= The assignment operator.
MAP INTELLEC Command

\[
\text{MAP INTELLEC} = \begin{cases} \text{segmentno} \mid \text{segmentno} \to \text{segmentno} \\ \text{segmentno \ length segmentlen} \end{cases} \quad \begin{cases} \text{segmentno} \mid \text{segmentno} \to \text{segmentno} \\ \text{segmentno \ length segmentlen} \end{cases} \ldots
\]

Examples:

- MAP INTELLEC = 73
- MAP INTELLEC = 100 TO 123
- MAP INT = 100 LENGTH 23
- MAP INT = 65, 68 TO 76, 100 LEN 23

MAP
A command keyword referring to some operation on the ICE-88 memory map.

INTELLEC
A command modifier specifying Intellec memory.

\textit{segmentno}
A segment number (64 to 1023) that specifies a physical memory segment in Intellec memory. It is used to map Intellec memory in one of the following ways:

- As the address of a single physical segment in Intellec memory.
- As the address of the first physical segment in Intellec memory of a partition of physical segments being mapped.
- As the address of the last physical segment in Intellec memory of a partition of physical segments being mapped.

TO
A connector keyword that denotes that a segment number is to follow that defines the upper bound of a memory partition.

LENGTH
A connector keyword that denotes that a segment length value is to follow.

\textit{segmentlen}
A segment length value (1 to 960) that defines the length of a partition of memory.

= The assignment operator.

NOTE
All mapping to Intellec memory is contingent upon the amount of expanded Intellec memory a user has in his system.
## Set MAP Status Command

```
MAP {logsegmentno [TO logsegmentno] } = {GUARDED
USER [NOVERIFY]
ICE [physegmentno] [NOVERIFY]
INTELLEC [physegmentno] [NOVERIFY]
DISK [physegmentno] [NOVERIFY]}
```

Examples:

- `MAP 457 = ICE 0 NOVERIFY`
- `MAP 100 TO 200 = USER`
- `MAP 201 LEN 62 = INT 65 NOV`
- `MAP 263 LENGTH 87 = DISK 100`
- `MAP 351 TO 400 = DIS NOV`
- `MAP 401 TO 456 = GUARDED`
- `MAP 458 TO 1023 GUARD`

<table>
<thead>
<tr>
<th>MAP</th>
<th>A command keyword referring to some operation on the ICE-88 memory map.</th>
</tr>
</thead>
<tbody>
<tr>
<td>logsegmentno</td>
<td>A logical segment number (0 to 1023) that specifies a segment in logical address space. It is used to set the ICE-88 map in one of the following ways:</td>
</tr>
<tr>
<td>TO</td>
<td>A connector keyword that denotes that a logical segment number is to follow that defines the upper bound of a partition of logical segments.</td>
</tr>
<tr>
<td>LENGTH</td>
<td>A connector keyword that denotes that a logical segment length value is to follow defining a partition of logical segments.</td>
</tr>
<tr>
<td>segmentlen</td>
<td>A segment length [1 to 1024] that defines the length of partition of memory.</td>
</tr>
<tr>
<td>=</td>
<td>The assignment operator</td>
</tr>
<tr>
<td>GUARDED</td>
<td>The initial state of all memory segments. Any reference to a guarded address causes an error message. In the emulation mode accesses to a guarded location will cause emulation to terminate upon completion of the current instruction. In the interrogation mode, no access to the given location will be made.</td>
</tr>
<tr>
<td>USER</td>
<td>Refers to locations in user prototype memory.</td>
</tr>
<tr>
<td>ICE</td>
<td>Refers to locations in ICE-88 memory.</td>
</tr>
<tr>
<td>INTELLEC</td>
<td>Refers to locations in Intellec memory.</td>
</tr>
</tbody>
</table>
DISK  Refers to locations in diskette memory.

physegmentno  A physical segment number (0 to 1023) that specifies a physical segment of memory locations. Intellec expanded memory physical segment numbers are limited to a range of 1-960. If present in the command, the first logical segment in the partition is set equal to the value of physegmentno and the subsequent logical segments are set equal to the subsequent physical segment number values. If no physegmentno is entered, the ICE-88 emulator assigns physical segments from those declared but not yet mapped to.

NOVERIFY  A function keyword that suppresses the normal read-after-write verification of data loaded into the designated memory.
Display MAP Status Command

MAP { logsegmentno [TO logsegmentno] 
    { logsegmentno LENGTH logsegmentlen } }

Examples:

MAP
MAP 0 TO 100
MAP 123 LENGTH 200
MAP 300 LEN 400

MAP A command keyword referring to some operation on the ICE-88 memory map.

logsegmentno A segment number (0 to 1023) that specifies a segment of addresses in logical address space. It is used in the following ways:
• As the address of a single segment of logical addresses.
• As the address of the first logical segment of a partition of logical segments.
• As the address of the last logical segment of a partition of logical segments.

TO A connector keyword that denotes that segment number is to follow that defines the upper bound of a partition of logical segments.

LENGTH A connector keyword that denotes that a segment length value that defines the length of a partition of logical segments.

logsegmentlen A segment length [1 to 1024] that defines the length of a partition of memory.
RESET MAP Command

RESET MAP

Example:

RESET MAP

RESET  A command keyword that restores its object to its initial state, as after an initial ICE-88 invocation.

MAP  As the object of RESET, the token MAP causes the memory map to be set to its initial condition, all GUARDED. The available DISK and INTELLEC memory is deleted from the map.
Set Memory and Port Content Commands

The commands in this section set new values or change the current content stored in designated memory locations or input/output ports. The commands discussed in this section are as follows. The purpose of each command is indicated by its title.

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>Set Memory Contents</td>
</tr>
<tr>
<td>Set Input/Output Port Contents</td>
</tr>
</tbody>
</table>

Discussion

Memory Content References

A memory content reference has the form:

```
memory-type address
[ !mod-name ] !symbol-name ...
```

The meta-term `memory-type` means one of the following "content-of" modifiers for memory locations.

- **BYTE**
  - The content of a single byte (8-bit) memory location. The `address` following `BYTE` is treated as a logical address; the physical address whose content is referenced is determined by look-up in the ICE-88 memory map (see Memory and I/O Port Mapping Commands).

- **WORD**
  - The content of two adjacent bytes in memory. The most significant byte is located in the high address of the address pair; the least significant byte is stored in the low address of the pair. The `address` following `WORD` is treated as a logical address; the ICE-88 memory map is consulted to find the physical address whose content is referenced.

- **SINTEGER**
  - The same as `BYTE` except when displaying.

- **INTEGER**
  - The same as `WORD` except when displaying.

- **POINTER**
  - The content of four adjacent bytes in memory, interpreted as a base and displacement. The displacement is located at the low 2 bytes of the 4 and the base is at the high 2 bytes. The `address` following `POINTER` is treated as a logical address; the ICE-88 memory map is consulted to find the physical address whose content is referenced.
The meta-term *address* means one of the following types of entries.

**numeric-expression** The forms for numeric expressions are presented in Chapter 5. The result obtained when the expression is evaluated becomes an address modulo 64K.

**(mem-type address)** A memory content reference with a form such as BYTE (WORD 1000) represents an indirect reference. The content of the address or address-pair inside the parentheses is treated as the address for the *mem-type* outside the parentheses.

To obtain the content of bytes, words, or pointers in a range of addresses, use the form:

*memory-type partition*

A *partition* can be a single address, or one of the following types of constructs.

- **address TO address**
- **address LENGTH number-of-bytes** (for BYTE and SINTEGER)
- **address LENGTH number-of-words** (for WORD and INTEGER)
- **address LENGTH number-of-double-words** (for POINTER)

The first form of *partition* uses the keyword **TO**. The address on the right of the keyword **TO** must be greater than or equal to the one on the left. With BYTE and SINTEGER, this form allows you to access the content of each byte location in the range; the range includes both the first and last address in the partition. With WORD or INTEGER, the first address is treated as the low address of the first address pair in the range; subsequent pairs of addresses are accessed until the second address is reached. If the second address is the low address of a pair, the word formed from the content of that address and the next consecutive higher address is accessed; if the second address is not the low address of a pair (that is, if it turns out to be the high address of a pair already accessed in the range), the access ends after the last complete pair has been accessed. Word-length accesses can begin on either an even-numbered or an odd-numbered address. With POINTER, the first address is treated as the low address of the first address quadruple in the range; subsequent quadruples of addresses are accessed until the second address is reached. If the second address is the low address of a quadruple, the pointer formed from the content of that address and the next three consecutive higher addresses is accessed; if the second address is not the low address of the quadruple, the access ends after the last quadruple has been accessed. Pointer length accesses can begin on either an even-numbered or an odd-numbered access.

The second, third, and fourth forms use the keyword **LENGTH**. The *address* preceding the keyword **LENGTH** is the starting address in the range, as with the first form (using **TO**). The number or expression following the keyword **LENGTH** gives the number of addresses (when the controlling *memory-type* is BYTE or SINTEGER), the number of address pairs (for WORD or INTEGER) or the number of addresses quadruples (for POINTER) (Must be an integer value).
Interrogation and Utility Commands

Setting Memory Contents

To assign a new content to a byte, sinteger, word, integer, pointer or reallocation, use a command with the form:

\[
\begin{align*}
\text{mem-type} & \quad \text{address} \\
& \quad [!!\text{mod-name}] \{\text{symbol-name} \ldots\} = \text{new-content}
\end{align*}
\]

The meta-terms \textit{mem-type} and \textit{address} represent the types of entries discussed earlier in this section.

The meta-term \textit{new-content} represents one of the following types of entries (for single addresses, setting the content of a range of addresses will be discussed later on).

\textit{numeric-expression} \quad A numeric expression evaluated by the ICE-88 emulator to a single number.

If \textit{mem-type} is a \textit{POINTER}, \textit{new-content} must be a pointer value. Otherwise, \textit{new-content} must be an integer value.

When a single byte address is to be set, the ICE-88 emulator treats the \textit{new-content} as an 8-bit quantity. If \textit{new-content} has more than eight bits, the least significant eight bits in the quantity are used as the new content; and the other (higher) bits are lost. If \textit{new-content} has fewer than eight bits, the bit values in the quantity are right-justified (placed in the low-order bits in the address), and the remaining (high) bits in the location are set to zeroes.

Here are some examples of setting byte contents. The first line of each example shows the command that sets the new contents; the second line gives a command that produces a display of the contents just set; the third line shows the resulting display. The output radix is assumed to be H (hexadecimal).

*BYTE 1000H = FFH
*BYTE 1000H
B0 00:1000H=FFH

*BYTE 1010H = RAL + 1
*BYTE 1010H
B0 00:1010H=F1H

*BYTE 1020H = FF11H
*BYTE 1020H
B0 00:1020H=11H

*BYTE 1030H = 1Y
*BYTE 1030H
B0 00:1030H=01H

*BYTE 1040H = 'A'
*BYTE 1040H
B0 00:1040H=41H

*BYTE 1050H = BYTE 1000H
*BYTE 1050H
B0 00:1050H=FFH

You can change the radix used to display the contents, using the Set Output Radix (BASE) command.
When a single word address is to be set, the ICE-88 emulator treats the *new-content* as a pair of bytes. The least significant byte is loaded into the low address in the pair, and the most significant byte is loaded into the high address in the pair. If *new-content* has fewer than 16 bits, the bit values are loaded starting with the low address, and right-justified. The remaining (high) bits in the address pair are set to zeroes. The following examples demonstrate some of the possibilities for setting address pairs.

*WORD 1000H = 1122H
*WORD 1000H
WOR 0000:1000H=1122H

*WORD 1010H = FFH
*WORD 1010H
WOR 0000:1010H=00FFH

*WORD 1030H = WORD 1000H
*WORD 1030H
WOR 0000:1030H=1122H

When a single pointer is to be set, the ICE-88 emulator treats the *new-content* as a displacement and base. The least significant byte is loaded into the low address in the quadruple, and most significant byte is loaded into the high address. The following examples demonstrate some of the possibilities for setting pointers.

*POINTER 1000H = 1122:3344H
*POINTER 10000H
POI0000:1000H=1122:3344H

*POINTER 100:0 = 1122:3344H
*POINTER 0000:1000H
POI0000:1000H=1122:3344H

A command of the form *BYTE X =BYTE Y* copies the content of the address *Y* to the content of address *X* where addresses *X* and *Y* are either integer or pointer values as shown in the examples above. A command of the form *WORD X = WORD Y* copies the content of address *Y* to location *X*, and the content of address *(Y + 1)* to location *(X + 1)*. A command of the form *POINTER X = POINTER Y* copies the content of location *Y* through *(Y + 3)* to locations *X* through *(X + 3)* respectively. A command of the form *BYTE X = WORD Y* copies the content of address *Y* to location *X*; the content of location *(X + 1)* is unchanged. A command of the form *WORD X = BYTE Y* copies the content of address *Y* to location *X*; the content of location *(X + 1)* is set to a byte of zeroes. *POINTER X = BYTE Y* copies *(Y)* to *(X)* zeroes in *(X) + 1* to *(X) + 3*.

The commands used to set a range of addresses differ in some details.

One way to set a range of addresses is with the command of the form:

```
mem-type address =list of new-content values
```

With this form the *address* on the left side of the equals sign gives the starting location and the number of values in the list to the right of the equals sign tells the ICE-88 emulator how many consecutive addresses to load. Consecutive locations are changed to the values of the *new-contents* in left-to-right order. The list of new content values can consist of expressions, multi-character strings and memory partitions.
Here are some examples showing the use of this form of the set memory contents command.

\[
\begin{align*}
\text{*BYTE} & \quad 1000H = 11H, 22H, 33H, 44H, 55H, 66H, 'AB' \\
\text{*BYT} & \quad 1000H \text{ LEN 8T} \\
& \quad \text{BYT} 0000:1000H=11H 22H 33H 44H 55H 66H 41H 42H \\
\text{*WOR} & \quad 200:0H = FFFFH, \text{WORD} 100:0H \\
& \quad \text{WOR} 0200:0000H=FFFFH 2211H \\
\text{*POI} & \quad 4000:20H = 1122:3344H, \text{WOR} 1002H, \text{BYT} 1002 \\
& \quad \text{POI} 4000:20H \\
& \quad \text{POI} 4000:0020H=1122:3344 0000:4433 0000:0033
\end{align*}
\]

To set a range of addresses all to the same new value, use a command of the form:

\[
\text{mem-type partition } = \text{new-content}
\]

The forms of partition are discussed above in this section. All addresses in the partition are set to the single new-content value. The following examples show some of the possible results obtained with this command form.

\[
\begin{align*}
\text{*BYTE} & \quad 1000H \text{ TO } 1005H = \text{FFH} \\
\text{*BYTE} & \quad 1000H \text{ LEN 5H} \\
& \quad \text{BYT} 0000:1000H=\text{FFH FFH FFH FFH FFH} \\
\text{*WORD} & \quad 200:0 \text{ LENGTH 6T = AAOOH} \\
& \quad \text{WOR} 0200:0000H=\text{AAOOH AAOOH AAOOH AAOOH AAOOH AAOOH AAOOH AAOOH} \\
\text{*POINTER} & \quad 3000H \text{ LEN 3T = 1234:5678H} \\
& \quad \text{POI} 0:3000H \text{ TO } 0:300BH \\
& \quad \text{WOR} 0000:3000H=1234:5678 1234:5678 1234:5678
\end{align*}
\]

The last form of the set memory contents command sets the contents of each address in a range (partition) to the corresponding new-content in a list of values. This form is as follows.

\[
\text{mem-type partition } = \text{list of new-content values}
\]

This form combines the two forms discussed above. The list of new-content values can consist of expressions, multi-character strings and memory partitions.

If the number of locations in the partition is equal to the number of values in the new-content list, the addressed locations are set to the corresponding values in the list, in left-to-right order.

If the number of locations in the range is greater than the number of new values in the list, the locations are filled with the values from left-to-right, repeating the values in left-to-right order as necessary to fill all the locations. The maximum number of new content bytes that can be repeated is 128. With more than 128 bytes, the data is transferred but not repeated, and an error message is displayed.

If the number of new values in the list is greater than the number of locations in the partition, the lowest location receives the first value, and successive locations in the range receive values in left-to-right order until all locations in the range have received values. The excess values are then detected by the ICE-88 emulator as an error condition, and an error message is displayed. The excess values are lost.
Here are a few examples showing this form of command.

* BYT 1000H TO 1004H = 'ABCDE'
* BYT 1000H LEN 5T
  BYT 0000:1000H=41H 42H 43H 44H 45H

*WORD 200:0H LENGTH 6T= 1122H, 'AB'
*WOR 200:0H TO 200:AH
  WOR 0220:0000H=1122H 0041H 0042H 1122H 0041H 0042H

*BYTE 1000H TO 1002H = 11H, 22H, 33H, FFH
(an error message such as EXCESSIVE DATA is displayed)
*BYT 100:0 LEN 4T
  BYT 1000:0000=11H 22H 33H 44H

In the third example, note that the byte at location 1003H retains the value set in the first example in the group of examples (44H rather than the FFH given in the command.)

Port Content References

The set port contents commands parallel those of the set memory contents commands with exception of port addressing. There are a total of 65536 8-bit ports available for input/output in the ICE-88 emulator. The ports are referenced by the mnemonic PORT. These ports can be referenced as 16-bit ports by the mnemonic WPORT. Each port is referenced by a port-number that is an integer value in the range of 0 through 65535.

To assign a new content to an 8-bit or 16-bit port, use the following command with the form:

\[ \text{port-type port-number} = \text{new-content} \]

The meta-terms \text{port-type} and \text{port-number} are used to specify individual ports.

\begin{itemize}
  \item \text{port-type} Defines the port type and size:
    \begin{itemize}
      \item \text{PORT} An 8-bit port.
      \item \text{WPORT} A 16-bit port.
    \end{itemize}
  \item \text{port-number} An integer value specifying a specific port.
\end{itemize}

The following are examples of the use of this command form.

PORT 123 = FFH
WPORT 123 = FFFFH

One way of setting a range of ports is with the command of the form:

\[ \text{port-type port-number} = \text{list of new-content values} \]

With this form the \text{port-number} on the left side of the equals sign gives the starting port location and the list to the right of the equals sign tells the ICE-88 emulator how many consecutive ports to load. Consecutive ports are changed to the values of the \text{new-contents} in left-to-right order.
Here are some examples showing the use of this form.

PORT 1000H = 11, 22, 33H, 44H, 55H, 66H
WPORT 1000H = 1122H, 3344H, 5566H

To set a range of ports all to the same new value, use a command of the form:

\[ \text{port-type partition} = \text{new-content} \]

The following are examples of this form:

PORT 1000H TO 1005 = FFH
WPORT 2000H LENGTH 20H = FFFFH

The last form of the set port contents sets the contents of each port in a range (partition) to the corresponding new-content in a list of values. This form is as follows:

\[ \text{port-type partition} = \text{list of new-content values} \]

If the number of ports in the partition is equal to the number of values in the new-content list, the addressed ports are set to the corresponding values in the list, in left-to-right order.

If the number of ports in the range is greater than the number of new values in the list, the ports are filled with the new values from left-to-right, repeating the values in left-to-right order as necessary to fill all the ports.

If the number of new values in the list is greater than the number of ports in the partition, the lowest numbered port receives the first value, and successive ports in the range receive values in left-to-right order until all ports in the range have received values. The excess values are then detected by the ICE-88 emulator as an error condition, and an error message is displayed. The excess values are lost.

Here are a few examples showing this form of command.

PORT 1000 TO 1005H = 'ABCDE'
WPORT 2000H LEN 6T = 1122H, 'AB'
PORT 1000H TO 1002H = 11H, 22H, 33H, FFH
(an error message such as EXCESSIVE DATA is displayed)
Set Memory Command

```
mem-type partition =new-content [, new-content ] ...
typed-mem-ref = numeric-expression
```

Examples:

```
BYTE 0800H = FFH
BYTE 7000H LENGTH 16T = 00H
BYT 0800 TO 0805 = 12H, 34H, 56H, 78H, 9AH, BCH
WORD 70FFH = IP
WORD 7000H = PIP + 1
POINTER 8000H = ABCD:1234H
BYTE 0800H = 'ABCDE'
POI 7000H = POI 4000H LEN 20H
BYT #56 = FAH
!VAR = 75
!X = !X + 1
!!MODA !PTR = SS:SP
```

**mem-type**
One of the six memory "content-of" modifiers BYTE, WORD, SINTEGER, INTEGER, or POINTER.

**partition**
One or more contiguous locations in memory.

**new-content**
One of the following types of entries, to be used as the new contents of the memory location:

- **numeric-expression**
- **'string'**
- **mem-type partition**
- **port-type partition**

**typed-mem-ref**
See page 5-6.
Set Input/Output Port Contents Command

\[
\text{port-type partition} = \text{new-content}, \text{new-content}...
\]

Examples:

\begin{verbatim}
PORT 0800H = FFH
PORT 7000H LENGTH 16T = 00H
POR 0880 TO 0885 = 12H, 34H, 56H, 78H, 9A, BCH
WPORT 70FFH = IP
WPO 7000H = PIP + 1
PORT 0800H = 'ABCDE'
PORT 56 = FAH
\end{verbatim}

- **port-type**
  One of the two port "content-of" modifiers PORT or WPORT.

- **partition**
  One or more contiguous ports (must be integers).

- **new-content**
  One of the following types of entries, to be used as the new contents of the port:
  - *numeric-expression*
  - "string"
  - *mem-type partition*
  - *port-type partition*
Symbol Table and Statement Number Table Commands

The ICE-88 emulator maintains a symbol table and source program statement number table to allow you to refer to memory addresses and other values by using symbolic references and statement references in the ICE-88 commands.

This section gives details on the following commands.

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>DEFINE Symbol</td>
</tr>
<tr>
<td>Display Symbols</td>
</tr>
<tr>
<td>Display Statement Numbers</td>
</tr>
<tr>
<td>Display Modules</td>
</tr>
<tr>
<td>Change Symbol</td>
</tr>
<tr>
<td>REMOVE Symbols</td>
</tr>
<tr>
<td>REMOVE Module</td>
</tr>
<tr>
<td>TYPE</td>
</tr>
<tr>
<td>Set DOMAIN</td>
</tr>
<tr>
<td>RESET DOMAIN</td>
</tr>
</tbody>
</table>

Discussion

The ICE-88 symbol table receives symbols from two sources; the symbol table associated with the user program can be copied to the symbol table when the program is loaded, and the user can define additional symbols for use during the emulation session.

Corresponding to each symbol in the table is a number that you can interpret and use either as an address or as a numeric value (variable or constant). The next few paragraphs discuss the kinds of symbols that can appear in the table, and the interpretation of the corresponding symbol table quantity (address or value).

Instruction and statement labels are loaded with the program code. The symbol table gives the address of the instruction corresponding to the label.

A program variable is a symbol for a quantity that can have its value changed as a result of an instruction in the program. Program variables are LOADed with the program code. The symbol table gives the address where the variable value is stored.

A program constant is a symbol for a label set to a constant value (for example, using the assembler directive EQU or SET). Program constants are loaded into the symbol table when the program code is loaded. The symbol table gives the constant value associated with the symbol.

A module name is the label of a simple DO block that is not nested in any other block (for PL/M-86), or a label that is the object of a NAME directive (in 8086 assembly language). If no NAME directive is given, the module name is the same as the source file name without the extension. For example, if the source file name is "*:F1:MYPROG.A88," the module name will be "MYPROG."
A module name itself does not have a corresponding address value in the symbol table. However, symbols contained in a module are considered to be "local" to that module; the ICE-88 emulator thus allows you to reference multiple occurrences of the same symbol name in different modules, by using the module name as a modifier in the symbolic reference.

The symbol table is organized to preserve the modular structure present in the program. Initially (before any code is loaded), the symbol table consists of one "unnamed" module. Any symbols defined without a specific module name are stored in the unnamed module in the order they were defined. The unnamed module is always the first module in the symbol table. Following the unnamed module, named modules are stored in the symbol table in the order that the modules were loaded into the ICE-88 emulator. Symbols local to each named module are stored in the order they appear in the module.

In addition to the symbols stored when the program code is loaded, you can use the DEFINE Symbol command to define new symbols for your use during the emulation session. The rules for user-defined symbols are as follows.

The name of the new symbol (symbol-name) can be defined with a maximum of 122 characters. However, the ICE-88 emulator truncates each symbol-name to the first 31 characters. Thus, to be different, two symbols must be unique in the first 31 characters.

The first character in the new symbol-name must be an alphabetic character, or one of the three characters @ or ? or underscore (_). The remaining characters after the first can be these characters or numeric digits.

You can specify the module that is to contain the new symbol you define. Symbols defined without a module are placed in the unnamed module at the head of the table, in the order they were defined. Symbols defined with an existing module name are placed in that module's section of the table; the module named must already exist in the table.

The new symbol name cannot duplicate a symbol name already present in the module specified. You can, however, have two or more symbols of the same name in different modules.

When you define a new symbol, you also specify the value corresponding to it in the table. You can treat the value you assign as an address or as a numeric value for use other than addressing.

The DEFINE Symbol command has the following form.

\[
\text{DEFINE symbolic-reference} = \text{address::value} \ [\text{OF memory-type}] 
\]

The forms of symbolic-reference are shown in Table 7-8. The meaning of each form is as follows. Not all forms can be used in a DEFINE Symbol command.

A simple symbolic-reference has the form .symbol-name. the ICE-88 emulator searches for this form of reference starting with the first symbol in the unnamed module. If the symbol is not in the unnamed module, the ICE-88 emulator searches through the named modules in the order they were loaded, and takes the first occurrence of the symbol in the first (earliest) module that contains it.

When you define a symbol without a module, it is placed in the unnamed module.
The *symbolic-reference* can include a *module-reference*. The module reference immediately precedes the symbol name; the *module-name* is identified by a prefix consisting of a double period (..). When you define a symbol with a module reference, the symbol is added to the symbols under that module. A later reference to a symbol with a module name restricts the search to that module.

### Table 7-8. Symbolic References and Statement References

<table>
<thead>
<tr>
<th>Type of Reference</th>
<th>Meta-notation</th>
<th>Example</th>
<th>Display</th>
<th>DEFINE</th>
<th>Change</th>
<th>REMOVE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symbolic</td>
<td>.symbol-name</td>
<td>.ABC</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Symbolic</td>
<td>..module.symbol-name</td>
<td>..MAIN.DEF</td>
<td>YES, if module is already present in table.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Symbolic</td>
<td>.symbol-name.symbol-name</td>
<td>XX.YY</td>
<td>YES</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>Statement</td>
<td>#statement-number</td>
<td>#56</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>Statement</td>
<td>..module#stmt-number</td>
<td>..MAIN#44</td>
<td>YES</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>

The meta-term *address/value* as used in the DEFINE Symbol command means one of the forms of numeric expressions given in Chapter 4.

The meta-term *memory-type* one of the following memory types: BYTE, WORD, SINTEGER, INTEGER, or POINTER.

Once a symbol has been defined or loaded, any reference to that symbol is equivalent to supplying its corresponding address or value.

To display the value from the symbol table corresponding to any symbol, enter the appropriate symbolic reference followed by a carriage return. The ICE-88 emulator displays the symbol table value on the next line.

To display the entire symbol table, enter the command SYMBOL followed by a carriage return. Symbols are displayed module by module, starting with the unnamed module. The *address/value* corresponding to each symbol is also displayed.

Example 1.

```
*.SAM
.SAM=0200:1FE2H OF INT  (command)
.SAM=0200:1FE2H OF INT  (display)
```

Example 2.

```
*.MYPROG.SAM
.SAM=0200:1FE2H OF INT  (command)
.SAM=0200:1FE2H OF INT  (display)
```
The ICE-88 emulator also maintains a statement number table for user programs written in PL/M-86 source code. The statement numbers are assigned by the PL/M-86 compiler. Corresponding to each source statement number in the table is the address of the first instruction generated by that source statement.

Table 7-8 shows the forms used to refer to statement number in the ICE-88 emulator. The simplest form is the statement-number prefixed by a number sign (#). A module-reference can precede the statement reference, since the statement number table preserves any modular structure in the program. Thus, two modules compiled separately can have the same statement numbers; the module reference tells the ICE-88 emulator which statement number to use.

To display the address corresponding to a statement-number, enter the appropriate statement number reference followed by a carriage return.

The ICE-88 emulator does not allow you to change the address corresponding to any existing statement number, to define any new statement numbers, or to delete (REMOVE) any statement numbers.

To display the value from the statement number table of any statement number, enter the appropriate statement number reference followed by a carriage return. The ICE-88 emulator displays the statement number value on the next line.

Example 1.

*#1
#1=0800:0050H

Example 2.

*.MAIN #2
#2=0800:0057H

To display the addresses of all the statement numbers in the statement number table, enter the keyword LINE. The ICE-88 emulator displays all the statement number addresses starting on the line following the command.

Example.

*LINE
MODULE ..MAIN
#1=0800:0050H
#2=0800:0057H
MODULE ..SUBR
#1=1140:0012H
#2=1140:0037H
#3=1140:00DFH
To display the names of all the modules currently in the ICE-88 module table, enter the keyword MODULE. The ICE-88 emulator displays the names of all the modules currently in the table.

Example.

```
*MODULE
MODULE ..MAIN
MODULE ..SUBR
```

You can change the address/value corresponding to an existing symbol by entering a command of the form:

```
symbolic-reference=address::value [OF memory-type ]
```

Any of the three forms of symbolic-reference shown in table 7-8 can be used to identify the symbol whose value is to be changed. The symbol must already exist as referenced.

The forms of address::value are discussed earlier in this section. Any of these forms may be used to change the value of an existing symbol.

Where multiple occurrences of the same symbol name exist in the table, the rules for table search given earlier determine which of the several instances of the symbol is to receive the new address value .

To delete one or more symbols from the table, use a command of the form:

```
REMOVE list of symbolic-references
```

The symbolic-references in the list are separated by commas. The ICE-88 emulator searches the table for each reference using the search rules given earlier, deleting the first occurrence of each symbol name that fits the type of reference given.

Note that deleting a symbol from the ICE-88 symbol table makes that symbol inaccessible to the ICE-88 emulator but does not affect the program code.

To delete the entire symbol table and the statement number table, enter the command REMOVE SYMBOL.

To delete one or more modules, enter the following command:

```
REMOVE MODULE module-name [, module-name ]...
```

Removing a module removes all symbols and statement numbers in the module, but does not affect object code.

Example 1.

```
REMOVE MODULE ..MAIN
```

Example 2.

```
REM MOD ..MAIN, ..SUBR
```
The TYPE command allows you to assign or change the memory type of any symbol in the symbol table. The TYPE command is entered in the following format:

```
TYPE symbolic-reference = memory-type
```

The referenced symbol is assigned the memory type entered in the command.

Example.

```
TYPE .MYPROG .SAM = WORD
```

The symbol .SAM now is memory type WORD.

The Set DOMAIN command establishes a specified module as the default module for statement numbers. The RESET DOMAIN command establishes the first module in the module table after the unnamed module as the default for statement numbers, if there is a module other than the unnamed module; otherwise it establishes the unnamed module as the default. This is the initial domain.

Setting the domain should be especially useful for avoiding having to use module names on statement numbers from a particular module while debugging that portion of the program.
DEFINE Symbol Command

DEFINE \textsl{[module-name]} \textsl{symbol} = \textsl{expression} \textsl{[OF memory-type]}

Examples:

\begin{verbatim}
DEFINE ..MAIN .BEGIN = F3H OF BYTE
DEF .CAR = 0000:0F00H
DEF .VAR = 123T OF WOR
DEF .ENT1 = .VAR + 10 OF WOR
DEF .CAT2 = 0700:0050H OF POI
DEF ..SUBRA .CAT2 = 0000:00F0H OF POI
\end{verbatim}

\begin{itemize}
\item \textbf{DEFINE} A command keyword that tells the ICE-88 emulator to enter the new symbol in the appropriate module table and assign the symbol the initial value given.
\item \textit{module-name} A sequence of contiguous alphanumeric characters, prefixed by a pair of periods (..) that references a program module.
\item \textit{symbol} A sequence of contiguous alphanumeric characters, prefixed by a period (.) that references a location in a symbol table.
\item \textbf{=} The assignment operator.
\item \textit{expression} A numeric expression.
\item \textbf{OF} A command modifier keyword denoting that a specification of memory type is to follow.
\item \textit{memory-type} A specification of the memory type of the symbol: BYTE, WORD, SINTEGER, INTEGER, or POINTER. If omitted, symbol has no type.
\end{itemize}
Display Symbols Command

SYMBOL

[module-name] symbol [symbol]...

Examples:

SYMBOL
.ABC
..MAIN.DEF
..XX.YY
.SUBR.ABC.DEF

SYMBOL A command keyword that tells the ICE-88 emulator to display the entire ICE-88 symbol table, module by module.

module-name A sequence of continuous alphanumeric characters, prefixed by a period (..) that references a program module.

symbol A sequence of contiguous alphanumeric characters, prefixed by a period (.) that references a location in a symbol table.
Display Statement Numbers Command

LINE

[module-name] #decimal-10

Examples:

LINE
#54
..MAIN#44

LINE A command keyword that tells the ICE-88 emulator to display all statement numbers and associated absolute addresses currently in the statement number table.

module-name A sequence of contiguous alphanumeric characters, prefixed by a pair of periods (...) that reference a program module.

# The "number" sign designating the reference as a statement number.

decimal-10 The (source) statement number (a numeric constant). The default suffix is always decimal.
Display Modules Command

MODULE

Example;

MODULE

MODULE A command keyword that tells the ICE-88 emulator to display the names of all the modules currently in the module table.
Change Symbol Command

\[
\text{[module-name ] symbol [symbol ]... = expression [OF memory-type ]}
\]

Examples:

\[
\begin{align*}
.\text{ABC} &= 2000H \\
.\text{MAIN.DEF} &= \text{AAFFH OF WOR} \\
.\text{SUBR.PARM} &= .\text{ABC} + 10 \\
.\text{TEMP} &= .\text{ABC} + .\text{MAIN.DEF OF WORD}
\end{align*}
\]

\[
\begin{array}{ll}
\text{module-name} & \text{A sequence of contiguous alphanumeric characters, prefixed by a pair of periods (..), that references a program module.} \\
\text{symbol} & \text{A sequence of alphanumeric characters, prefixed by a period (.) that references a location in the symbol table.} \\
= & \text{The assignment operator.} \\
\text{expression} & \text{A numeric expression.} \\
\text{OF} & \text{A command modifier keyword denoting that a specification of memory type is to follow.} \\
\text{memory-type} & \text{A specification of the memory type of the changed symbol: BYTE, WORD, SINTEGER, INTEGER, or POINTER. If omitted, do not change symbol's type.}
\end{array}
\]
REMOVE Symbols Command

REMOVE \([module-name] \) symbol \([symbol]\)\,[\,|\,\([module-name] \) symbol \([symbol]\)\,\]...\)

REMOVE SYMBOL

REMOVE MODULE \([module-name]\)...

Examples:

REMOVE .ABC
REMOVE ..MAIN.DEF
REMOVE .HIJ.,PARM1, ..MAIN.TWO,.CARS,.CARS1..SUBR.XX
REMOVE SYMBOL
REMOVE MODULE ..MAIN,..SUBR,..CALC

REMOVE A command keyword causing the symbols that follow to be deleted from the ICE-88 symbol table.

module-name A sequence of alphanumeric characters, prefixed by a pair of periods (.) that references a program module.

symbol A sequence of alphanumeric characters, prefixed by a period (.) that references a location in the symbol table.

SYMBOL A command modifier that tells the ICE-88 emulator to delete the entire current symbol table.

MODULE A command modifier that tells the ICE-88 emulator to delete all the symbols and lines of the named module from the tables but does not affect object code.
TYPE Command

TYPE [module-name] symbol [symbol]... = memory-type

Examples:

TYPE
TYPE .ABC = POINTER
TYPE ..MAIN.DEF = WOR
TYPE ..SUBR.PARM.XX.YY = BYT

TYPE
module-name
symbol
=
memory-type

A command keyword that allows the user to assign or change the memory type of any symbol in the symbol table.

A sequence of alphanumeric characters, prefixed by a pair of periods (..) that references a program module.

A sequence of alphanumeric characters prefixed by a period (.) that references a location in the symbol table.

The assignment operator.

A specification of the memory type to be assigned to the referenced symbol: BYTE, WORD, SINTEGER, INTEGER, or POINTER.
**Set DOMAIN Command**

`DOMAIN module-name`

Example:

`DOMAIN ..MAIN`

**DOMAIN** A command keyword that causes the ICE-88 emulator to establish the named module as the default module for statement numbers.

**module-name** A sequence of alphanumeric characters prefixed by a pair of periods (..) that references a program module.
RESET DOMAIN Command

RESET DOMAIN

Example:

RESET DOMAIN

RESET          A command keyword restoring its object to a reset condition.

DOMAIN         A modifier keyword that causes the ICE-88 emulator to establish the first module after the unnamed module as the default module for statement numbers. If there are no named modules, the unnamed module is established as the default module.
Display Commands

This section presents the ICE-88 commands that allow you to reference and display the following systems elements.

- 8088 Processor Registers
- ICE-88 Status Registers
- 8088 Pin Signals
- Memory
- Ports
- Status Flags

The following commands are discussed in this section.

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Display Processor and Status Registers</td>
<td>Display the current contents of any of the 8088 processor registers and ICE-88 status registers.</td>
</tr>
<tr>
<td>Display Memory</td>
<td>Display the contents of a range of memory.</td>
</tr>
<tr>
<td>Display I/O</td>
<td>Display the contents of a range of I/O locations.</td>
</tr>
<tr>
<td>Display STACK</td>
<td>Display the contents of the user's stack.</td>
</tr>
<tr>
<td>Display Boolean</td>
<td>Display boolean values.</td>
</tr>
<tr>
<td>Display NESTING</td>
<td>Display procedure starting and return addresses.</td>
</tr>
<tr>
<td>EVALUATE</td>
<td>Display numeric constant or expression in all five possible output radices.</td>
</tr>
</tbody>
</table>

Discussion

Registers

The 8088 register structure contains three files of four 16-bit registers and a set of miscellaneous registers. The three files of registers are the general register file, the pointer and index file, and the segment register file. The miscellaneous set consists of the instruction pointer, flag register, CAUSE register,_OPCODE register, PIP register, TIMER register, HTIMER register, BUFFERSIZE register, UPPER register and LOWER register. The register structures are described in the following paragraphs.

Table 7-9. Classes of Hardware Elements

<table>
<thead>
<tr>
<th>Class Name</th>
<th>Hardware Elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>general-register</td>
<td>16-bit work register</td>
</tr>
<tr>
<td>pointer-register</td>
<td>16-bit address register</td>
</tr>
<tr>
<td>index-register</td>
<td>16-bit address register</td>
</tr>
<tr>
<td>segment-register</td>
<td>16-bit segment reference register</td>
</tr>
<tr>
<td>status-register</td>
<td>8 and 16-bit status registers</td>
</tr>
</tbody>
</table>
**General Register File.** The RAX, RBX, RCX, and RDX registers compose the General Register File. These registers participate interchangeably in 8088 arithmetic and logical operations. These registers are assigned the following mnemonics:

- RAX: Accumulator
- RBX: Base Register
- RCX: Count Register
- RDX: Data Register

The general registers are unique within the 8088 as their upper and lower bytes are individually addressable. Thus the general registers contain two 8-bit register files called the H file and L file as illustrated below:

```
  H File     L File
  15  8  7  0
RAX:  RAM  RAL
RBX:  RBH  RBL
RCX:  RCH  RCL
RDX:  RDH  RDL
```

**General Register File**

**Pointer and Index Register File.** The BP, SP, SI, and DI set of 16-bit registers is called the Pointer and Index Register File. The registers in this group are similar in that they generally contain offset addresses used for addressing within a segment. They can participate interchangeably in 16-bit arithmetic and logical operations. They are also used in address computation. The mnemonics associated with these registers are:

- SP: Stack Pointer
- BP: Base Pointer
- SI: Source Index
- DI: Destination Index

The pointer and index registers are illustrated below:

```
  15  0
SP: 
BP: 
SI: 
DI: 
```

**Pointer and Index Register File**

**Segment Register File.** The CS, DS, SS, and ES registers constitute the Segment Register File. These registers provide a significant function in the memory addressing mechanisms of the 8088. They are similar in that they are used in all memory address computations. The mnemonics associated with these registers are:

- CS: Code Segment Register
- DS: Data Segment Register
- SS: Stack Segment Register
- ES: Extra Segment Register

The contents of the CS register define the current code segment. All instruction fetches are taken to be relative to CS using the instruction pointer (IP) as an offset.
The contents of the DS register define the current data segment. All data references except those involving BP, SP, or DI in a string instruction, are taken by default to be relative to DS.

The contents of the SS register define the current stack segment. All data references which implicitly or explicitly involve SP or BP are taken by default to be relative to SS.

The contents of the ES register define the current extra segment. The extra segment has no specific use, although it is usually treated as an additional data segment.

The segment registers are illustrated below:

---

**Segment Register File**

**Status Register**

The instruction pointer, flag register, CAUSE register, OPCODE register, PIP register, TIMER register, HTIMER register, BUFFERSIZE register, UP, and LOWER register constitute the status register set. These registers provide a variety of functions to the 8088. These registers are assigned the following mnemonics:

- **IP**: Instruction Register
- **RF**: Flag Register
- **CAUSE**: CAUSE Register
- **OPCODE**: OPCODE Register
- **PIP**: Previous Instruction Register
- **TIMER**: TIMER Register
- **HTIMER**: HTIMER Register
- **BUFFERSIZE**: BUFFERSIZE Register
- **LOWER**: LOWER Register
- **UPPER**: UPPER Register

The contents of the IP register defines the offset to the CS register in instruction address computations. The Flag Register contains the status flag values. The CAUSE register is used to retain the cause of the last break in emulation. The OPCODE register stores the opcode fetched in the last instruction-fetch cycle in trace data. The Previous Instruction Register is used to store the displacement part of the address of the last instruction-fetch in trace data. TIMER contains the low-order 16 bits of the 2-MHz timer indicating how long emulation has run (read only). HTIMER contains the high-order 16 bits of the timer (read only). BUFFERSIZE contains the count (displayed in decimal only) of frames of valid trace data collected in the trace buffer (16 bit, read only). The LOWER register contains the lowest available address in Intellec memory above the ICE-88 software (16 bit read only). The UPPER register contains the highest available address in Intellec memory below the user's symbol table (16 bit, read only).
The status registers are illustrated below:

![Status Registers Diagram]

**Status Registers**

The Flag Register (RF) contains nine status values, each one a bit in length. The following mnemonics are assigned to each of the status values in the register:

- **AFL**: Auxiliary-carry
- **CFL**: Carry
- **DFL**: Direction
- **IFL**: Interrupt-enable
- **OFL**: Overflow
- **PFL**: Parity
- **SFL**: Sign
- **TFL**: Trap
- **ZFL**: Zero

AFL is set if an instruction caused a carry out of bit 3 and into bit 4 of a resulting value. CFL is set if an instruction caused a carry or a borrow out of the high order bit. DFL controls the direction of the string manipulation instructions. IFL enables or disables external interrupts. OFL is used to denote an overflow condition in a signed arithmetic operation. SFL is used to indicate the sign of the result of an operation. TFL is used to place the processor in a single-step mode for program debugging. ZFL is used to indicate a zero valued result of an instruction. The position of the status bits in the RF Register are shown below:

![Flag Registers Diagram]

**Flag Registers**

The CAUSE Register is used to store the cause for the last break in emulation. The contents of this register is shown below:

![Cause Register Diagram]

**Cause Register**
The byte returned by the “Read Break Cause” hardware command contains the following bit values (if bit = 1, then the specified condition is true, otherwise false):

<table>
<thead>
<tr>
<th>Bit Position</th>
<th>Condition</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Breakpoint 0 matched</td>
</tr>
<tr>
<td>1</td>
<td>Breakpoint 1 matched</td>
</tr>
<tr>
<td>2</td>
<td>Both breakpoints matched sequentially</td>
</tr>
<tr>
<td>3</td>
<td>Guarded memory access occurred</td>
</tr>
<tr>
<td>4</td>
<td>User aborted processing</td>
</tr>
<tr>
<td>5</td>
<td>Timeout on user READY</td>
</tr>
<tr>
<td>6</td>
<td>Timeout on user HOLD</td>
</tr>
</tbody>
</table>

Pin References.

In addition to the status registers, the ICE-88 emulator provides access to six 8088 pins. The following mnemonics are assigned to reference the 8088 processor pins shown below:

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>8088 Pin</th>
</tr>
</thead>
<tbody>
<tr>
<td>RDY</td>
<td>READY</td>
</tr>
<tr>
<td>NMI</td>
<td>NMI</td>
</tr>
<tr>
<td>TEST</td>
<td>TEST</td>
</tr>
<tr>
<td>HOLD</td>
<td>HOLD</td>
</tr>
<tr>
<td>RST</td>
<td>RESET</td>
</tr>
<tr>
<td>MN</td>
<td>MN/MX</td>
</tr>
<tr>
<td>IR</td>
<td>INTR</td>
</tr>
</tbody>
</table>

General Formats for Numeric Values

The ICE-88 emulator displays numeric values in a variety of formats depending upon the type of the numeric value. Table 7-10 defines the display formats for each of the numeric types.

<table>
<thead>
<tr>
<th>Type</th>
<th>Class Name</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>byte</td>
<td>An 8-bit value displayed in the current base.</td>
</tr>
<tr>
<td>SINTEGER</td>
<td>sinteger</td>
<td>sign byte-number (short integer number)</td>
</tr>
<tr>
<td>WORD</td>
<td>word</td>
<td>A 16-bit value displayed in the current base.</td>
</tr>
<tr>
<td>INTEGER</td>
<td>integer</td>
<td>sign word</td>
</tr>
<tr>
<td></td>
<td>sign</td>
<td>+ :: −</td>
</tr>
<tr>
<td></td>
<td>bit</td>
<td>0 :: 1</td>
</tr>
<tr>
<td>POINTER</td>
<td>pointer</td>
<td>base : displacement H</td>
</tr>
<tr>
<td></td>
<td>base</td>
<td>4 hexadecimal digits (the base value of pointer)</td>
</tr>
<tr>
<td></td>
<td>displacement</td>
<td>4 hexadecimal digits (the displacement value of pointer)</td>
</tr>
</tbody>
</table>
If the high order bit of an INTEGER or SINTEGER is 1, the numeric value is complemented and the sign is set negative ("-`). For example, FFH is displayed as -01H as SINTEGER and F000H is displayed as -1000H as INTEGER.

In base ASCII, a pair of apostrophes enclose a single character in the case of a byte value or encloses two characters if a word value is to be displayed. In the case of a word, the high order byte appears on the left of the character pair.

In the four numeric bases, the BYTE and WORD values have a suffix and sufficient leading zeroes to contain the following number of digits.

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Octal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>WORD</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

The Display Processor and Status Register Command allows you to display any of the 8-bit and 16-bit registers, the status flags and 8088 pin values. All referenced items are displayed on one line separated by spaces. However, if any displayed value would extend beyond column 80, a new line of display is initiated. Each reference is displayed according to the appropriate format shown below. The names are truncated to three characters.

8-bit-register-name = byte
16-bit-register-name = word
status-flag-name = bit
pin-name = bit

Example 1.

RAX, RBH, SP, CAUSE, AFL, HTIMER, BUFFERSIZE

Display:

RAX=0001H RBH=2FH SP=FFE7H CAU=00H AFL=1 HTI=008EH BUF=D1A2H

Example 2.

REGISTER

Display:

RAX=0000H RBX=00A2H RCX=0001H RDX=0010H SP=000AH BP=0000H SI=0123H DI=0000H CS=0000H DS=FF1EH SS=0000H ES=0000H RF=0000H IP=FABCH

Example 3.

FLAG

Display:

CLF=0 PFL=0 AFL=0 ZFL=0 SFL=0 TFL=1 IFL=0 DFL=0 OFL=0
Example 4.

PIN Display:

\[ \text{RDY=1 NMI=0 TES=1 HOL=0 RST=0 MN=1 IR=0} \]

The Display Memory command enables you to display a range of one or more memory locations. When displaying memory, the format of the display depends upon the memory type, given either explicitly by a \textit{mem-type} or implicitly by the type of a symbol in a \textit{typed-memory-reference}. For example, suppose that four consecutive bytes of memory contain A5H, 81H, 34H and 0C0H. The following are sample outputs in each of the four bases: hexadecimal, decimal, octal and binary:

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Hexadecimal</th>
<th>Decimal</th>
<th>Octal</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>A5H</td>
<td>165T</td>
<td>245Q</td>
<td>10100101Y</td>
</tr>
<tr>
<td>WORD</td>
<td>81A5H</td>
<td>33189T</td>
<td>100645 Q</td>
<td>1000000110100101Y</td>
</tr>
<tr>
<td>SINTEGER</td>
<td>-5BH</td>
<td>-91T</td>
<td>-133Q</td>
<td>-01011011Y</td>
</tr>
<tr>
<td>INTEGER</td>
<td>-7E5BH</td>
<td>-32347T</td>
<td>-77133 Q</td>
<td>-0111110010111011Y</td>
</tr>
<tr>
<td>POINTER</td>
<td>C034:81A5H</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Pointer type memory is always displayed in hexadecimal, regardless of the current output base. The Display memory command permits you to display more than one line of memory values. Each line of display contains the memory address of the first value displayed on that line followed by a maximum number of values as indicated by the following table.

<table>
<thead>
<tr>
<th>Hexadecimal</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>64</td>
</tr>
</tbody>
</table>

Assume that the following memory locations contain the values shown below (in hexadecimal):

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000H</td>
<td>00H 41H 42H 43H 45H 20H 30H 3DH 74H 68H 65H 72H 20H 43H 61H 6CH</td>
</tr>
<tr>
<td>00010H</td>
<td>69H 66H 6FH 72H 6EH 69H 61H 0DH 0AH 20H 20H 20H 20H 20H 20H 20H</td>
</tr>
<tr>
<td>00020H</td>
<td>20H 20H 20H 6CH 61H 77H 20H 70H 72H 6FH 76H 69H 64H 65H 64H 20H</td>
</tr>
<tr>
<td>00030H</td>
<td>66H 6FH 72H 20H 70H 72H 6FH 72H 61H 6DH 73H 20H 6FH 68H 20H</td>
</tr>
<tr>
<td>00040H</td>
<td>74H 68H 69H 73H 20H 73H 6FH 72H 74H 2EH 20H 20H 54H 68H 69H 73H</td>
</tr>
</tbody>
</table>
The following commands display the above memory values with BASE = H.

Example 1.

BYTE 0000:0030

Display:

BYTE 0000:0030H=66H

Example 2.

BYTE 0:30 TO 0:3F

Display:

BYTE 0000:0030H=66H 6FH 72H 20H 70H 72H 6FH 67H 72H 61H 6DH 73H 20H 6FH 66H 20H

Example 3.

WORD 0000:30

Display:

WORD 0000:0030H=6F66H

Example 4.

WOR 0:30 LEN 10H

Display:

WOR 0000:0030H=6F66H 2072H 7270H 676FH 6172H 736DH 6F20H 2066H

WOR 0000:0040H=6874H 7369H 7320H 726FH 2E74H 2020H 6854H 7369H

Example 5.

POINTER 0:30

Display:

POI 0000:0030H=2072:6F66H

Example 6.

POI 0:30 TO 0:3C

Display:


Example 7.

POI 0:30 TO 0:4F

Display:


POI 0000:0040H=7369:6874H 726F:7320H 2020:2E74H 7369:6854H
The following examples illustrate display in decimal (BASE=T).

Example 8.

BYTE 0000:0030

Display:

BYTE 0000:0030H=102T

Example 9.

BYTE 0:30 TO 0:3F

Display:

BYTE 0000:0030H=102T 111T 114T 032T 112T 114T 111T 103T
BYTE 0000:0038H=114T 097T 109T 115T 032T 111T 102T 032T

Example 10.

WORD 0:30

Display:

WOR 0000:0030H=28518T

Example 11.

WORD 00:30 TO 00:4E

Display

WOR 0000:0030H= 28518T 08306T 29296T 26479T 24946T 29549T 28448T
WOR 0000:0040H= 26740T 29545T 29472T 29295T 11892T 08224T 26708T 29545T

Example 12.

POINTER 0:30

Display:

POI 0000:0030H=2072:6F66H

Example 13.

POINTER 0:30 TO 0:3C

Display:

The following examples illustrate memory displays in octal (BASE=Q).

Example 14.

BYTE 0000:0030

Display:

BYTE 0000:0030H=146Q

Example 15.

BYTE 0:30 LEN 10H

Display:

BYTE 0000:0030H=146Q 157Q 162Q 040Q 160Q 162Q 157Q 147Q
BYTE 0000:0038H=162Q 141Q 155Q 163Q 040Q 157Q 146Q 040Q

Example 16.

WOR 0:30

Display

WOR 0000:0030H=067546Q

Example 17:

WORD 0:30 LEN 10H

Display:

WOR 0000:0030H=067546Q 020162Q 071160Q 063557Q
WOR 0000:0038H=060562Q 071555Q 067440Q 020146Q
WOR 0000:0040H=064164Q 071551Q 071440Q 071157Q
WOR 0000:0048H=027164Q 020040Q 064124Q 071551Q

Example 18.

POINTER 0:30 LEN 4

Display:


The following examples illustrate memory displays in binary (BASE=Y).

Example 19.

BYTE 0:30

Display:

BYTE 0000:0030H=01100110Y
Example 20.
BYTE 0:30 TO 0:3F

Display:

<table>
<thead>
<tr>
<th>Byte Address</th>
<th>Hex Value</th>
<th>Binary Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000:0030H</td>
<td>01101111Y</td>
<td>11011001Y</td>
</tr>
<tr>
<td>0000:0034H</td>
<td>01101111Y</td>
<td>11011001Y</td>
</tr>
<tr>
<td>0000:0038H</td>
<td>01101111Y</td>
<td>11011001Y</td>
</tr>
<tr>
<td>0000:003CH</td>
<td>01101111Y</td>
<td>11011001Y</td>
</tr>
</tbody>
</table>

Example 21.
WOR 0:30

Display:

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000:0030H</td>
<td>0110111101100110Y</td>
</tr>
</tbody>
</table>

Example 22.
WOR 0:30 LEN 10H

Display:

<table>
<thead>
<tr>
<th>Word Address</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000:0030H</td>
<td>0110111101100110Y</td>
</tr>
<tr>
<td>0000:0034H</td>
<td>0110111101100110Y</td>
</tr>
<tr>
<td>0000:0038H</td>
<td>0110111101100110Y</td>
</tr>
<tr>
<td>0000:003CH</td>
<td>0110111101100110Y</td>
</tr>
</tbody>
</table>

Example 23.
POINTER 0:30

Display:

<table>
<thead>
<tr>
<th>Pointer Value</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000:0030H</td>
<td>2072:6F66H</td>
</tr>
</tbody>
</table>

The Display I/O command enables you to display byte ports (PORT) and word ports (WPORT) in a manner similar to the Display memory command. Single port contents or the contents of a range of ports may be displayed. However, only a single integer is required to specify the port address. The following examples illustrate the Display I/O command. The implied suffix in these examples is H (hexadecimal) for address specification.

Example 1.

BASE = Y
PORT 120H

Display:

<table>
<thead>
<tr>
<th>Port Address</th>
<th>Hex Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0120H</td>
<td>10111111Y</td>
</tr>
</tbody>
</table>
Example 2.

PORT 120 LEN 10

Display:

POR 0120H=10111111Y 10111111Y 10111111Y 01111111Y
POR 0124H=10111111Y 10111111Y 10111111Y 01111111Y
POR 0128H=10111111Y 10111111Y 10111111Y 01111111Y
POR 012CH=10111111Y 10111111Y 10111111Y 01111111Y

Example 3.

BASE = Q
PORT 120

Display:

POR 0120H=277Q

Example 4.

PORT 120 TO 12F

Display:

POR 0120H=277Q 177Q 177Q 177Q 177Q 177Q 177Q
POR 0128H=277Q 177Q 177Q 177Q 177Q 177Q 177Q

Example 5.

BASE = T
PORT 120

Display

POR 0120H=191T

Example 6.

PORT 120 TO 12F

Display;

POR 0120H=191T 127T 191T 127T 191T 127T 191T 27T
POR 0127H=191T 127T 191T 127T 191T 127T 191T 127T

Example 7.

BASE = H
PORT 120

Display:

POR 0120H=BFH
Example 8.
PORT 120 TO 12F
Display:
\[ \text{POR0120H=BFH 7FH BFH 7FH BFH 7FH BFH 7FH BFH 7FH BFH 7FH BFH 7FH} \]

Example 9.
BASE = Y
WPORT 120
Display:
\[ \text{WPO 0120H=011110111011111Y} \]

Example 10.
WPORT 120 TO 12E
Display:
\[
\begin{align*}
\text{WPO 0120H} &= 0111101110111111Y \\
\text{WPO 0124H} &= 0111111110111111Y \\
\text{WPO 0128H} &= 0111101110111111Y \\
\text{WPO 012CH} &= 0111101110111111Y \\
\end{align*}
\]

Example 11.
BASE = Q
WPORT 120
Display
\[ \text{WPO 0120H=077677Q} \]

Example 12.
WPORT 120 TO 13E
Display:
\[
\begin{align*}
\text{WPO 0120H} &= 077677Q \ 077677Q \ 075677Q \ 075677Q \ 075677Q \ 075677Q \ 075677Q \\
\text{WPO 0130H} &= 077677Q \ 077677Q \ 075677Q \ 075677Q \ 075677Q \ 075677Q \ 075677Q \\
\end{align*}
\]

Example 13.
BASE = T
WPORT 120
Display:
\[ \text{WPO 0120H=32703T} \]
Example 14.

WPORT 120 TO 13E

Display:

WPO 0120H = 32703T 32703T 32703T 31679T 31679T 32703T 31679T 32703T
WPO 0130H = 32703T 32703T 31679T 32703T 31679T 32703T 32703T 31679T

Example 15.

BASE = H
WPORT 120

Display:

WPO 0120H = 7FBFH

Example 16.

WPORT 120 TO 13E

Display:

WPO 0120H = 7FBFH 7FBFH 7FBFH 7BBFH 7BBFH 7FBFH 7BBFH 7FBFH 7FBFH
WPO 0130H = 7FBFH 7FBFH 7BBFH 7FBFH 7BBFH 7FBFH 7FBFH 7BBFH

The Display STACK causes the top \( n \) words of the user's stack (i.e., user memory pointed at by SS:SP) where \( n \) is an integer value in the command that specifies the number words in the stack to be displayed.

Example

STACK 10

Display:

WOR 0000:0000H = 4100H 4342H 2045H 3D30H 6874H 265H 4320H 6C61H
WOR 0000:0010H = 6669H 726FH 696EH 0D61H 200AH 2020H 2020H 2020H

The Display Boolean command is used to display the boolean value of an integer value contained in the command:

Example:

BOOL FFH

Display:

TRUE

Example:

BOOL !X = !Y

Display:

FALSE
The Display NESTING Command

The Display NESTING command enables you to display the starting and return addresses of all procedures that are currently active. The user stack is scanned for procedure starting and return addresses as follows (all references to stack manipulations are restricted to the scope of the nesting module only):

1. Set \( b = \text{CS} \)
2. Set \( d = \text{SP} \) (word at the top of the user stack). If the 5 bytes from \((b,d-5)\) through \((b,d-1)\) can be interpreted as a short call (direct or indirect) (2, 3, or 4 bytes), then \( b,d \) is assumed to be a return address.
3. Set \( \text{WORD} \ temp$b:d = \text{next word on the user stack.} \) If the 5 bytes from \((temp$b,d-5)\) through \((temp$b,d-1)\) can be interpreted as a long call (direct or indirect) (2, 3, 4 or 5 bytes), INT (1 or 2 bytes), or INTO, then \( b,d \) (and the next word in the stack for INT and INTO) can be assumed to be a return address.
4. The return address, type of call (i.e., short call direct), INT, or INTO, and the starting address (for direct only) are displayed.

The above procedure is repeated until 16 iterations fail to find a return address or the stack memory enters guarded memory. Care should be taken in using this command as the above method is susceptible to error.

The EVALUATE Command

The EVALUATE command handles the arithmetic computation involved in translating integers from one radix to another and computes the 20-bit address of a pointer. This command has the form \( \text{EVALUATE expr} \), where \( \text{expr} \) is any numeric constant or numeric expression. Upon receiving this command, the ICE-88 emulator evaluates any expression to a single number. If it is an integer, it displays the result in the four bases \( Y \), \( Q \), \( T \), and \( H \), and the corresponding ASCII characters, all on one line. For ASCII, the characters are enclosed in single quotes (\' \'); printing characters are displayed (ASCII codes \( 20\text{H} \) through \( 7\text{E}\text{H} \) after bit 7 is masked off), while non-printing characters are suppressed.

When the EVALUATE command is followed by the keyword ‘SYMBOLICALLY’ (preceding the carriage return) the numeric value output by the command displayed as a \( \text{symbol} \) or \( \text{statement number} \) plus a remainder. The ICE-88 symbols and \( \text{statement numbers} \) are searched to find the one with the same base whose value is closest to but not greater than the value being output. In the event that a \( \text{symbol} \) and a \( \text{statement number} \) have the same value, the \( \text{symbol} \) is used. The value is then displayed as either the selected \( \text{symbol} \) plus a \( \text{numeric-constant} \) or the selected \( \text{statement number} \) plus a \( \text{numeric-constant} \), where the \( \text{numeric-constant} \) is the remainder in the current output base. If no \( \text{symbol} \) or \( \text{statement number} \) has a value less than or equal to the number being output, the value is output as a numeric-constant. If the remainder is zero, the \( \text{numeric constant} \) is omitted.

If the \( \text{numeric expression} \) has a non-zero base, the value is displayed as a pointer (base:disp).
Here are several examples of the use of the EVALUATE command, with the display produced by each one.

Example 1:

EVALUATE 123T

Display:

1111011Y 173Q 123T 7BH '{'

Example 2:

EVA FFH + 256T

Display:

1111111Y 777Q 511T 1FFH ''

Note that the addition was performed first, then the result was displayed in the four bases. The result contained only non-printing ASCII characters, displayed as empty quotes.

Example 3:

EVA 111:0222H SYMBOLICALLY

Display:

0111:0222H

This example assumes that no symbol or statement numbers match.

Example 4:

EVA 111:222H SYM

Display:

..MOD1.SAM + 0021H

This example assumes that symbol ..MOD1.SAM is at address at 111:201 and no symbols or statement numbers exist in the interval 111:201 through 111:222.
Display Processor and Status Registers Command

\[ reference \[, \, reference\]... \]

- REGISTER
- FLAG
- PIN

Examples:

- RAX
- RBH, SI, AFL, HOL
- REGISTER
- R
- FLA
- PIN

**reference**

- Any of the reference keywords that reference processor registers, status registers and pins. Also can include memory and I/O in list.

**REGISTER**

- A command keyword requesting the display of the thirteen 16-bit 8088 registers and RF.

**FLAG**

- A command keyword requesting the display of the nine status flags.

**PIN**

- A command keyword requesting the display of the contents of the seven input pins.
Display Memory Command

memory-designator address
[TO address
LENGTH length]

ASM address
[TO address
LENGTH length]

Examples:
BYTE 1000:100H
WOR 0:123 TO 0:200
SIN 100:0 LENGTH 200
INT 200:200
POINTER 200:200
POI 200:200 TO 200:2FE
ASM 1000:123
ASM 1000:123 TO 1000:400
ASM 2000:000 LEN 100

memory-designator
One of the following keywords that specify the size and type of
memory referenced:

BYTE A 1-byte (8-bit) integer value.
WORD A 2-byte (16-bit) integer value.
SINTEGER A 1-byte (8-bit) short integer number (see
Table 7-2).
INTEGER A 2-byte (16-bit) integer value (see Table 7-2).
POINTER A 4-byte (two 16-bit integer) value.

address
A pointer value containing a base and a displacement that
together specify an address of a memory location.

TO
A partition keyword that denotes that an address is to follow.
This address defines the upper bound of the required range of
addresses in the partition.

LENGTH
A partition keyword that denotes that a length value is to
follow.

length
An integer value specifying the number of addresses to be
contained in the partition (bytes, words or pairs of words,
depending on memory-designator).

ASM
A command keyword specifying the display of instructions in
disassembled format. (See Display of Trace Data in Instruc-
tions Mode, page 6-26.)
Display I/O Command

\[
\begin{align*}
\text{PORT} & \quad \text{WPORT} \\
\{ & \quad \{ & \quad \text{address} \quad \text{TO} \quad \text{address} \\
\} & \quad \} & \quad \text{LENGTH} \quad \text{length}
\end{align*}
\]

Examples:

PORT FF12H
POR FF00 TO FFFF
POR 1000 LEN 200
WPORT 123H
WPO 100 TO 200
WPO 100 LENGTH 101

PORT
Keyword reference to 8088 8-bit I/O port(s).

WPORT
Keyword reference to 8088 16-bit I/O port(s).

address
An integer value to that specifies the address of a 8088 port.

TO
A \textit{partition} keyword that denotes that an address is to follow. This address defines the upper bound of the required range of port addresses in the partition.

LENGTH
A \textit{partition} keyword that denotes that a \textit{length} value is to follow.

length
An integer value specifying the number of port addresses (byte or word ports) to be contained in the partition.
Display STACK Command

STACK expression

STACK 10H
STA .SAM
STACK .SAM + 20

STACK
expression

A command keyword that requests the display of the contents of the user's stack. The stack is located in user memory referenced by the pointer value SS:SP.

expression

An integer expression. The value of this expression defines the number of words at the top of the STACK that are to be displayed.
Display Boolean Command

BOOL expression

Examples:

    BOOL FFH
    BOO CS=DS AND IP > 50
    BOO BYTE .X = F2H
    BOOL !SAM
    BOOL CFL

BOOL A command keyword requesting the display of the boolean value (TRUE, FALSE) of the input value, expression.

d-expression A boolean expression. The value of this expression is evaluated to a boolean value. If the least significant bit of the expression = 1, the boolean value is TRUE, otherwise the boolean value is FALSE.
Display NESTING Command

NESTING

Example:

NESTING

NESTING  A command keyword that causes the display of the starting and return address of all procedures that are currently active.
EVALUATE Command

EVALUATE numeric-expression [SYMBOLICALLY]

Examples:

EVALUATE 123T
EVALUATE 4142H
EVALUATE FFH + 256T

EVALUATE numeric-expression

A command keyword that directs the ICE-88 emulator to evaluate the expression and display the result in all four number bases and ASCII.

ing numeric expression

A numeric expression or numeric constant.

SYMBOLICALLY

This keyword causes each numeric value output by the command to be displayed as a symbol or source statement number plus a remainder.
The command features described in this chapter enhance the operation of the ICE-88 emulator by extending the power of the simple ICE-88 commands.

ICE-88 enhancements are of two kinds: compound commands and macro commands. These features are as follows.

Compound commands:

- REPEAT command
- COUNT command
- IF command

Macro commands:

- DEFINE MACRO command
- Invoke macro command
- Display macro command
- Macro directory command
- REMOVE MACRO command
- PUT MACRO command

Please note that the examples in this chapter are independent of each other. The introduction to each example gives the initial conditions for that example, and does not assume any results or conditions from any previous examples.

**Compound Commands**

A compound command is a control structure that contains zero or more commands. The compound commands discussed in this chapter are the REPEAT, COUNT, and IF commands; the DEFINE MACRO command is also a type of compound command. As the command titles indicate, REPEAT and COUNT are looping commands, and IF establishes conditional execution, and DEFINE MACRO establishes a named command block. All compound commands can have a “local” setting for the default SUFFIX (console input radix), as described under Local and Global Defaults in this chapter.

**REPEAT Command**

The REPEAT command executes zero or more ICE commands in a loop; the loop can also contain zero or more logical conditions for termination.

The REPEAT command consists of the REPEAT keyword, zero or more commands of any type, zero or more exit conditions using WHILE or UNTIL, and the keyword END. Enter each of these elements on its own line of the console display; terminate each input line with an intermediate carriage return (shown as cr in the command syntax). The syntax or REPEAT can be shown as follows:

```
REPEAT
  command cr
  WHILE boolean-expression cr
  UNTIL boolean-expression cr
  ...
END
```
After each intermediate carriage return, the ICE emulator begins the next line with a period (giving an indented appearance), then the asterisk prompt to signal readiness to accept the next element. After the END keyword, enter a final carriage return to begin the sequence of execution. The final carriage return after END is not shown in the syntax, since all commands terminate with a final carriage return. The END keyword can be entered as ENDR or ENDREPEAT; the characters after END serve as a form of "comment" to indicate which loop is being terminated.

The elements to be repeated are shown in brackets in the syntax. Each element can be a command, a WHILE clause, or an UNTIL clause. You can mix these elements in any order, using any number of each type of element.

Each command is executed when it is encountered on each iteration. After the command has been completely executed, the loop proceeds to the next element.

The WHILE and UNTIL keywords introduce exit clauses. The WHILE clause terminates execution of the loop when its boolean-expression evaluates FALSE. The UNTIL clause terminates the loop when its boolean-expression evaluates TRUE.

In both the WHILE and UNTIL clauses, the boolean-expression is evaluated each time the clause is encountered; that is, once per iteration. Evaluation at each iteration involves looking up the values of any references in the expression. Thus, the result can change with each evaluation. Refer to Chapter 5 for an explanation of how expressions are evaluated.

The choice of WHILE or UNTIL is usually a matter of convenience — there is always a way to convert one into the other. For example, "WHILE bool-expr" is equivalent to "UNTIL NOT (bool-expr)."

NOTE
To terminate execution of a REPEAT (or COUNT) loop, press the ESC key at the console. The ICE command currently executing halts wherever it happens to be; if you are emulating, the current instruction is completed before the break. ICE responds to the ESC with the asterisk prompt.

Here are some brief examples of the REPEAT command.

Example 1: Generate an ASCII table similar to Table 5-2.

```
DEFINE .TEMP = 40H
REPEAT
  WHILE .TEMP <= 7EH
  EVALUATE .TEMP
  .TEMP = .TEMP + 1
ENDR
```

Example 2: Single-step through the user program and display the trace data collected for each instruction until a repetitious routine (.DELAY) is reached.

```
TRACE = INSTRUCTIONS
CS = SEG .START
IP = OFF .START
REPEAT
  UNTIL CS:IP = .DELAY
  STEP
  PRINT -1
ENDR
```
Example 3: Using a complex combination of conditions in the boolean expression.

```
REPEAT
  UNTIL (CS:IP > .END XOR !VAR1 = 0) AND (.TEMP > 0 XOR !VAR2 = 1)
  STEP
  REGISTER
ENDR
```

Example 4: Emulate from the start of the program (.START) until a breakpoint (.END EXECUTED) is reached, display status registers, then continue emulating, halting, and displaying status until a terminating condition (BYTE .VAR = 2) is reached.

```
CS = SEG .START
IP = OFF .START
REPEAT
  GO TILL .END EXECUTED
  REGISTER
  UNTIL !VAR = 2
ENDR
```

**COUNT Command**

Like REPEAT, the COUNT command sets up a loop. In addition to the WHILE and UNTIL clauses discussed under REPEAT, COUNT includes a loop counter that terminates the loop if no exit condition is met before the counter runs out.

The COUNT command has the form:

```
COUNT arithmetic expression cr
  command cr
    WHILE boolean-expression cr
    UNTIL boolean-expression cr
  END
```

The *arithmetic-expression* after COUNT controls the (maximum) number of iterations to be performed. If a numeric constant is used (for example, COUNT 10), the ICE emulator interprets it in implicit decimal radix; in other words, any number entered after COUNT without an explicit radix is interpreted as a decimal number.

If the entry after COUNT is an *arithmetic-expression*, it is evaluated to give the number of iterations. The COUNT expression is evaluated once, before any loop elements are encountered. It is not evaluated again on any iteration. The COUNT expression uses the values of any references it contains as they stand at the time of evaluation. For example, consider the following command sequence:

```
DEFINE .XX = 2
COUNT .XX
  .XX = .XX + 1
END
```

This loop goes through *two* iterations, although .XX has value 4 when the loop terminates.
The loop terminates when the number of iterations given by the **COUNT** expression has been performed or when an exit condition is tested and causes exit, *whichever comes first*. The following example illustrates this concept.

```
DEFINE .XX = 1
COUNT 5
  .XX = .XX + 1
  WHILE .XX < 5
END
```

To show that the loop terminates on the **WHILE** condition before the **COUNT** expression is exhausted, we can "track" the loop in operation. Table 8-1 shows the track.

<table>
<thead>
<tr>
<th>Iteration</th>
<th>.XX</th>
<th>.XX &lt; 5</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>TRUE</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>TRUE</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>TRUE</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

The loop terminates during the fourth iteration, when .XX < 5 becomes FALSE.

Conversely, the **COUNT** expression specifies the maximum number of iterations to be performed in case no exit clause produces an exit on any iteration. For example:

```
TRACE = INSTRUCTION
CS = SEG .START
IP = OFF .START
COUNT 10
  UNTIL CS:IP = .DELAY
  STEP
  PRINT -1
END
```

In this command, the **COUNT** expression specifies a maximum of ten **STEPs**, in case the first instruction at .DELAY is not reached during any iteration.

With a **REPEAT** command or with a **COUNT** command that include one or more clauses, there may be no direct way to tell how many iterations occurred before the loop terminated. For these cases, you can insert a loop counter as a loop element. For example, to obtain table 8-1 as a display (or LIST file output) you could use the following sequence.

```
BASE = T
DEFINE .ITER = 0
DEFINE .XX = 1
COUNT 10
  .XX = .XX + 1
  .ITER = .ITER + 1
  .ITER
  .XX
  BOOL .XX < 5
  WHILE .XX < 5
END
```

The command `BOOL .XX < 5` produces a display of **TRUE** or **FALSE**.
The following example emulates to a breakpoint, displays status registers, then continues emulating, breaking, and displaying status for a definite number of iterations:

```
CS = SEG .START
IP = OFF .START
COUNT 10
GO TILL .PAUSE EXECUTED
REGISTER
END
```

**IF Command**

The IF command permits conditional execution in a command sequence. The IF command has the form:

```
IF boolean-expression [THEN] cr
   [command cr] ...
[ORIF boolean-expression [THEN] cr] ...
   [command cr] ...
ELSE cr
   [command cr] ...
END
```

The command must have the IF clause; the ORIF and ELSE clauses are optional. The command can include as many ORIF clauses as desired. The IF and ORIF clauses each contain a single condition (boolean expression). Any clause can contain none, one, or more commands. A clause with no commands simply produces an exit when its condition is TRUE.

ICE examines each boolean expression in turn, clause by clause, looking for the first TRUE condition. If a TRUE condition is found, the commands in that clause are executed and the IF command terminates. If none of the conditions is TRUE, the commands in the ELSE clause are executed and the IF command terminates. If the ELSE clause is omitted and no condition is TRUE, the IF command terminates with no commands executed.

The END keyword is required to close off the IF command; it can be written as ENDIF to clarify nesting.

Here is an example of the IF command.

```
BASE = T
IP = 1
IF IP < 1
  EVALUATE 1
ORIF IP < 2
  EVALUATE 2
ORIF IP < 3
  EVALUATE 3
ELSE
  EVALUATE 4
END
```

This example displays the result of EVALUATE 2 and then terminates. The first condition (IF IP < 1) is FALSE, so EVALUATE 1 is skipped. The second condition (ORIF IP < 2) is TRUE, so EVALUATE 2 is executed and the IF command terminates. The third condition (ORIF IP < 3) is not tested, even though it happens to be TRUE.
In practice, however, the IF command is useful when it is nested in a REPEAT or COUNT loop rather than appearing at the “top” level. The reason for this is that you want to test conditions that can change (due to other commands in the loop), whereas at the top level the TRUE or FALSE state of any condition is known, or can be determined with the BOOL command. Thus, the result from the previous example can be obtained with fewer steps:

- `BOOL IP < 1  (Displays FALSE)`
- `BOOL IP < 2  (Displays TRUE)`
- `EVALUATE 2`

### Nesting Compound Commands

The REPEAT, COUNT, and IF commands can be nested to provide a variety of control structures.

Each nested compound command must have its own END keyword. When entering a nested command sequence, you may wish to use the keywords ENDR, ENDC, and ENDF, to help you keep straight which command you intend to close off. The ICE-88 emulator does not check nesting levels at entry, and if an END is omitted, the resulting error makes it necessary to enter the entire command again.

Each nested REPEAT or COUNT command can contain its own exit clauses (WHILE or UNTIL). Each such exit clause can terminate the loop that contains it, but has no effect on any outer loops or commands.

As an example of nesting, suppose you want to STEP through a program with trace display, but skip a repetitive timeout routine, .DELAY, that is called with an 8088 short-call instruction several times during program execution. One way to achieve this effect is with the following command sequence:

```
TRACE = INSTRUCTION
CS = SEG .START
IP = OFF .START
REPEAT
  IF CS:IP = .DELAY
    IP = WORD SS:SP
    SP = SP + 2
  ENDF
STEP
PRINT -1
ENDR
```

At each call to .DELAY in the program, the displacement of the return address for the call is pushed on the stack. The keyword SP refers to the stack pointer, and SS is the stack segment register; SS:SP is the address of the top of the stack where the return address is stored. The effects of the commands `IP = WORD SS:SP` and `SP = SP + 2` are to load the return address back into IP and reset the stack pointer just as if the return instruction at the end of .DELAY had been executed.

As another example of nesting, suppose the user code at statements #21 and #22 is incorrect or not written yet. The following sequence emulates to the point where substitute code is to be inserted, inserts the code (equivalent to IF MARK > 0 THEN PTR = PTR + 2 in PL/M), then continues emulating beginning with statement #23 (the insertion is made any time emulation reaches statement #21):
An exit can be made only when a condition is tested, not when it occurs. To cause an exit, the test must be placed at the point in the loop where the condition occurs. For example, consider the following command sequence:

```plaintext
CS = SEG .START
IP = OFF .START
REPEAT
    UNTIL IP = 1000H
    STEP
ENDR
```

In this command the condition IP = 100H is tested after every STEP. If the sequence of STEPs reaches IP = 1000H as the next instruction, the loop will terminate. By contrast, consider this example:

```plaintext
CS = SEG .START
IP = OFF .START
REPEAT
    UNTIL IP = 1000H
    COUNT 10
    STEP
    ENDC
ENDR
```

In the second example, the condition IP = 1000H is tested after every ten STEPs. The loop exits only if IP = 1000H occurs at the end of some group of ten instructions. If IP = 1000H occurs during one of the groups of ten STEPs, the loop does not terminate because that condition is changed by subsequent STEPs before the test can be made.

If the command has more than one exit clause, each exit clause is tested when it is encountered. If the result at the moment of the test causes an exit, the loop terminates; otherwise, the loop proceeds to the next element.

The loop exits only when the current test causes it, even though some other clause in the loop would cause an exit if it could be tested at that moment. Consider this (artificial) example:

```plaintext
DEFINE .ZZ = 0
CS = 0
IP = 0
REPEAT
    UNTIL IP > 10H
    COUNT 5
    STEP
    ENDC
    PRINT -10
    WHILE .ZZ = 0
        .ZZ = .ZZ + 1
ENDR
```

```plaintext
An exit can be made only when a condition is tested, not when it occurs. To cause an exit, the test must be placed at the point in the loop where the condition occurs. For example, consider the following command sequence:

```
CS = SEG .START
IP = OFF .START
REPEAT
    UNTIL IP = 1000H
    STEP
ENDR
```

In this command the condition IP = 1000H is tested after every STEP. If the sequence of STEPs reaches IP = 1000H as the next instruction, the loop will terminate. By contrast, consider this example:

```
CS = SEG .START
IP = OFF .START
REPEAT
    UNTIL IP = 1000H
    COUNT 10
    STEP
    ENDC
ENDR
```

In the second example, the condition IP = 1000H is tested after every ten STEPs. The loop exits only if IP = 1000H occurs at the end of some group of ten instructions. If IP = 1000H occurs during one of the groups of ten STEPs, the loop does not terminate because that condition is changed by subsequent STEPs before the test can be made.

If the command has more than one exit clause, each exit clause is tested when it is encountered. If the result at the moment of the test causes an exit, the loop terminates; otherwise, the loop proceeds to the next element.

The loop exits only when the current test causes it, even though some other clause in the loop would cause an exit if it could be tested at that moment. Consider this (artificial) example:

```
DEFINE .ZZ = 0
CS = 0
IP = 0
REPEAT
    UNTIL IP > 10H
    COUNT 5
    STEP
    ENDC
    PRINT -10
    WHILE .ZZ = 0
        .ZZ = .ZZ + 1
ENDR
```
Assume for this example that the code being emulated (with STEP) contains only two-byte instructions. Then, after the first time through the loop, IP = 0AH (10T) and .ZZ = 1. On the second iteration, the test IP > 10H is FALSE when it is encountered, so the STEP and PRINT commands are executed again. At this point, IP > 10H is TRUE but since it is not tested, no exit occurs. Instead, the condition .ZZ = 0 is tested, found to be false, and the loop exits.

Macro Commands

A macro is a block of commands. When a block of commands is defined as a macro, it is stored on diskette so that it can be executed more than once without having to enter the commands each time. The macro commands described in this chapter allow you to perform the following functions:

- Define a macro, specifying the macro name, the command block, and any formal parameters (points where text can be filled in at the time of the macro call).
- Invoke (call) a macro by name, giving actual parameters to fill in the blank fields in the macro definition, to begin the execution of the command block.
- Display the text of any macro as it was defined.
- Display the names of all macros currently defined.
- Remove one or more macros.
- Save one or more macro definitions on an ISIS-II file.

In addition, the off-line facility (INCLUDE command) allows you to enter macro definitions from diskette files for use in the current test sequence.

Defining and Invoking Macros

Each macro is defined once in the test session. The syntax of the DEFINE MACRO (DEF MAC) command is as follows:

```
DEF MAC macro-name cr
    [command cr]...
EM
```

Once it is defined, you can invoke (call) a macro as often as desired. The syntax of a macro call is:

```
:macro-name[actual-parameter-list]
```

The macro definition command causes the macro name and block of commands to be stored in a table of macro definitions in a temporary ISIS-II file named MAC.TMP. (This file is removed by the ICE EXIT command).

**WARNING**

If you have a file on the ICE diskette named MAC.TMP it will be lost when you enter the ICE-88 emulator.

A **macro-name** must begin with an alphabet letter, or with one of the character "@" or "?". The characters after the first character can be alphabet letters, "@", "?", or numeric digits. The macro name must not duplicate a previously-defined macro name.
A macro definition may not appear within any other command (REPEAT, COUNT, IF, or another macro definition). The command block in the macro definition can include any command except another DEF MAC command or a REMOVE MACRO command.

The macro name in the macro invocation must be the name of a previously-defined macro. The form of actual-parameter-list is discussed later in this chapter.

Here is a simple macro definition:

```
DEF MAC GOER
  REPEAT
    GO FROM .START TILL BR0
  END
EM
```

To invoke this macro and cause its command block to begin executing, enter the macro name preceded by a colon (:). For example:

```
:GOER
```

A macro definition can include calls to other macros, but a macro cannot call itself recursively. Any macros called from within a macro must have been defined when the calling macro is invoked. Macro calls can be nested; i.e., one macro calls another, which calls another, and so on. The level of nesting is limited only by the memory space required to contain the macro expansions and "stack" the macro calls.

When a macro is called as an outer level command the following operations occur:

- System default (SUFFIX) is saved in case a new default is set inside the macro.
- The text of each actual parameter in the call is substituted for the corresponding formal parameter in the definition.
- The expanded command block is executed if all commands are valid as expanded.
- When the last command has finished, the former system default is restored.
- The macro exits. Control returns to the console (asterisk prompt).

The next several sections provide details on these operations, including the treatment of nested macro calls.

**Local and Global Defaults**

The system default can have a "local" setting within a macro; this default is as follows:

<table>
<thead>
<tr>
<th>Default</th>
<th>Refers to:</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUFFIX</td>
<td>Default radix for console input.</td>
</tr>
</tbody>
</table>

When a macro is called (or any compound command is executed), the current "global" setting of SUFFIX, is saved so that it can be restored after the macro finishes executing its commands. The global default continues in effect within the macro unless and until a new (local) default is set with a SUFFIX command in the macro. Defaults other than SUFFIX are changed globally when they are set within a macro.
When the macro finishes executing its command block, the previous SUFFIX default is restored. Thus, any SUFFIX default that is set within a macro has no effect after that macro has exited.

Here is an (artificial) example of a macro with a local default:

```plaintext
DEF MAC SET0
   SUFFIX = H
   BYTE 0 TO 10 = 0
EM
```

Without the local SUFFIX command, the range of addresses to be set would depend on the global SUFFIX in effect when the macro SET0 is called. The global SUFFIX is restored after SET0 exits.

**Formal and Actual Parameters**

A formal parameter marks a place in an ICE command where variable text can be "filled in" when the macro is called. A formal parameter can represent part of a token or a field of one or more tokens. A macro definition can contain up to ten formal parameters. A formal parameter has the form:

```
%\n
```

where \( \text{n} \) is a decimal digit, 0 to 9.

Formal parameters can appear in the macro definition in any order, and each one can appear any number of times. In most cases, the formal parameters form a complete numeric sequence with \( %0 \) as the lowest numbered parameter (even if \( %0 \) is not the first parameter to appear). However, one or more parameters can be omitted from the sequence; the effect of omitting a formal parameter from the sequence is to ignore the actual parameter in the call that corresponds to the omitted formal parameter.

The macro call can contain as many actual parameters as desired. Enter multiple parameters as a list, with entries separated by commas. The first actual parameter in the list is substituted at all points that \( %0 \) appears in the macro definition; the second parameter substitutes for \( %1 \), and so on.

An actual parameter can be "null," causing ICE to substitute a null for the formal parameter to which it corresponds. You can pass a null parameter to a macro in two ways:

- Enter no actual parameter between consecutive commas.
- Omit one or more parameters from the end of the list.

If too few actual parameters are entered, the ICE-88 emulator supplies nulls for the extra formal parameters. If too many actual parameters are entered, the extra actual parameters are ignored.

If any actual parameter contains a carriage return, a comma, or a single quote mark, the entire parameter must be enclosed in single quotes to identify it as a single actual parameter. In other words, parameters with these characters must be entered as *strings*. A single quote within a string is entered as (\'\').
Here are some examples to demonstrate the use of formal and actual parameters:

**Example 1:**

```plaintext
DEF MAC MEM
  %0INTEGER %1
EM
```

In the call to this macro, parameter %0 can become "S" or null. Parameter %1 can be any valid address or partition. Examples of calls to this macro:

<table>
<thead>
<tr>
<th>Macro call</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>:MEM S, 1000H</td>
<td>SINTEGER 1000H</td>
</tr>
<tr>
<td>:MEM, 1000H TO 100FH</td>
<td>INTEGER 1000H TO 100FH</td>
</tr>
</tbody>
</table>

**Example 2:**

```plaintext
DEF MAC RPT
  REPEAT
    %0
    %1
    %2
    %3
    %4
    %5
    %6
    %7
    %8
    %9
  END
EM
```

Macro RPT can accept up to ten commands to be repeated. For example:

```plaintext
:RPT GO TILL BR0, PRINT ~1, REGISTERS, GO TILL BR1, PRINT ~10
```

If fewer than ten commands are given, as in the example above, the extra formal parameters are ignored (treated as nulls). This shows how to do REPEAT on one line with no end required.

**Example 3:**

```plaintext
DEF MAC BRS
  BR%0 = %1
EM
```

Use of macro BRS may require parameters entered as strings, since some ways to set breakpoints involve embedded commas. For example:

```plaintext
:BRS 0, 1000H EXECUTED
```

This parameter is valid, but this one:

```plaintext
BRS 0, FFH, .START LEN 100H
```

results in the expansion:

```plaintext
BR0 = FFH
```
To obtain the correct expansion, make the parameter a string:

:BRS 0, ‘FFH, .START LEN 100H’

This results in the expansion:

BR0 = FFH, .START LEN 100H

Details on Macro Expansion

The syntax and semantics of commands in a macro block are ignored at the point of definition; they are not determined until invocation, any may be different on each invocation through the use of formal parameters.

When a macro is called, its definition is expanded by adding the text of any actual parameters in the call at the points indicated by formal parameters in the definition. If the expanded macro contains any calls to other macros, the text of any such macros is also expanded, forming in effect one overall block of commands. The results of expansion are displayed at the console. Expansion continues until the last EM is reached. If the expansion results in a set of complete, valid commands, the commands are executed. An error results if any command is incomplete or invalid after expansion.

A macro invoked in a REPEAT, COUNT, or IF command is expanded immediately after the macro call command is entered. Thus, a macro called in a REPEAT or COUNT command is expanded only once, and a macro called in an IF command is expanded whether the condition in the IF or ORIF clause that contains the macro call is TRUE or FALSE.

Macro Table Commands

The macro table contains the name and text of all macros currently defined. The text is stored as it is defined, and does not contain any expansions.

The DEFINE MACRO (DEF MAC) command adds the macro defined to the end of the table. The syntax of this command appears earlier in this chapter. The DEF MAC command may not appear with any other command.

The REMOVE MACRO (REM MAC) command removes one or more macro definitions from the table. The syntax of this command is:

REM MAC [macro-name[,macro-name]...]

If the list of macro-names is omitted, all macros are removed. The REM MAC command may not appear within any other command.

The display macro command displays the name and definition of one or more macros from the macro table. The syntax is:

MAC [macro-name,macro-name]...

If the list of macro-names is omitted the definitions of all macros in the table are displayed.

The macro directory command displays the names of all macros in the table. The syntax is:

DIR MAC
Here are some examples of these commands (assume that the table contains all the macro examples defined thus far in this chapter):

Example 1:

```
*DIR MAC
GOER
SET0
MEM
RPT
BRS
```

Example 2:

```
*MAC GOER
DEF MAC GOER
REPEAT
GO FROM .START TILL BR0
END
EM
```

Example 3:

```
REM MAC BRS
```

Example 4:

```
*DIR MAC
GOER
SET0
MEM
RPT
```

Example 5:

```
*DEF MAC NULL
*EM
```

Example 6:

```
*DIR MAC
GOER
SET0
MEM
RPT
NULL
```

### Saving Macros

The PUT MACRO (PUT MAC) command causes one or more macro definitions to be copied from table to an ISIS-II diskette file. The syntax is:

```
PUT [:drive:] filename MACRO [macro-name[,macro-name][,...]
```

If any macro names are entered, those macro definitions are saved. If no list of macro names is given, all macros in the table are saved. The definitions in the macro table are not affected by the operation.
The file containing the saved macro can later be edited or brought into another session with the INCLUDE command, discussed later in this chapter.

If the named file does not exist, it is created by the PUT command. If the file does exist on the diskette, the file is opened for input and the macros in the list are written on the file, destroying the previous contents of that file.

Further Examples

Here are a few more examples of macros. These macros simulate stack operations, calls, and returns in the ICE-88 emulator.

A stack is an area of memory, indexed (addressed) by a register called the stack pointer (SP) and stack segment register (SS). The stack is used to save status information required for an orderly return from a procedure call.

In the ICE-88 emulator, the stack is in mapped memory. The bottom (first available location) is the highest address in the stack area; the stack expands as needed into successively lower addresses. The stack pointer points to the address (word) at the top of the stack; this address contains the last item pushed on the stack. As each new word is pushed on the stack, SP is decremented to point to the new top address. Most of the values that need to be saved on the stack are 16-bit values. The high byte is stored in the address pointed to by (SS:SP - 1), and the low byte is stored in the next lower address (equivalent to SS:SP - 2).

The MCS-86 assembly language PUSH rp instruction sets SP to the next available word, then stores the content of the given register pair rp in adjacent addresses at that position. We can simulate this action with a macro, as follows:

```
DEF MAC PUSH88
  SP = SP - 2T ; decrement SP
  WORD SS:SP = %0 ; low byte in low address, high byte in high address.
EM
```

The formal parameter %0 lets us use PUSH88 to save any register pair or other 16-bit value; for examples:

```
: PUSH88 IP ; save instruction register
: PUSH88 RAX ; save RAX
: PUSH88 RBX ; save RBX
: PUSH88 RCX ; save RCX
```

The complementary MCS-86 POP rp instruction copies the contents of the two top bytes pack into the given register pair, then increments SP to the new top of the stack. A macro for this function is:

```
DEF MAC POP88
  %0 = WORD SS:SP
  SP = SP + 2T
EM
```

Here are some calls to POP88, corresponding to the PUSH88 calls given earlier:

```
: POP88 RCX
: POP88 RBX
: POP88 RAX
: POP88 IP
```
Here are some macros that use PUSH88 and POP88.

1. Macro to “call” (short-call) a procedure:

   DEF MAC CALL88
   :PUSH88 IP
   GO FROM CS:%0
   EM

   This macro can be invoked with or without a halt condition:

   :CALL88 .PROC or
   :CALL88 .PROC TILL BR0

2. Macro to “call” (long-call) a procedure:

   DEF MAC LCALL88
   :PUSH88 CS
   :PUSH88 IP
   GO FROM %0
   EM

   To invoke this macro:

   :LCALL88 .PROC or
   :LCALL88 .PROC TILL BR1

3. Macro to “return” (short-return) from a procedure:

   DEF MAC RET88
   :POP88 IP
   GO %0
   EM

   To invoke this macro:

   :RET88 or
   :RET88 TILL BR0

4. Macro to “return” (long-return) from a procedure:

   DEF MAC LRET88
   :POP88 IP
   :POP88 CS
   GO %0
   EM

   To invoke this macro:

   :LRET88 or
   :LRET88 TILL BR0
5. Macro to single-step through user code, skipping over a specified procedure whenever that procedure is called from the user program, and printing the instruction just executed each time.

```
DEF MACRO SKIP
  REPEAT
    IF CS:IP = %0
      :POP88 IP
    ENDF
    STEP
    PRINT -1
  ENDR
  EM

DEF MACRO LSKIP
  REPEAT
    IF CS:IP = %0
      :POP88 IP
      :POP88 CS
    ENDF
    STEP
    PRINT -1
  ENDR
  EM
```

(short-called procedure)

(long-called procedure)

Suppose the user program contains a respective timer routine named DELAY that is called from several places in the program. The following macro invocation causes the ICE-88 emulator to step through the program without emulating the timer routine:

`:SKIP .DELAY`

**Off-line Facilities**

In addition to the compound and macro commands described above, the INCLUDE command allows you to access macro definitions stored in diskette files and to cause them to be executed from these files rather than the console.

**INCLUDE Command**

The INCLUDE command causes input to be taken from the file specified until the end-of-file, at which point input continues to be taken from the previous source, normally the console.

The syntax of the INCLUDE command is:

```
INCLUDE [:drive:] filename
```

Nesting of INCLUDE command is permitted. For example:

```
IP = .START
REPEAT
  UNTIL IP = .HALT
  INCLUDE PROGA.INC
        INCLUDE PROGB.INC
  ENDR
```
The console (:CI:) may be given as the *filename*, in which cases control-Z must be used as end-of-file. The files are echoed on the console.

As macros can be complex and editing may be required on the macro definition, the INCLUDE command allows you to access offline macro definitions and to create online macros, which combine to form the macro suite for a particular debugging session. However, command lines may appear in the INCLUDE file, not just macro definitions.

**Write Command**

The WRITE command writes one or more list elements to the console and to the list file at run time.

The syntax of the WRITE command is:

```
WRITE list-element [,list-element]
```

where

```
list-element = string :: expr :: BOOL expr
```

For example:

```
WRITE $CTIME, 'SECONDS SINCE LIGHT CHANGE'
```

Would output a message showing the time in seconds (in CARS2) (see Chapter 3) since the traffic light changed.

This command writes the *list-element* (s) to the console all on the same line except when the *nest-element* will not fit into the remaining character space on the current line. In this event, a carriage return and a line feed will be inserted. If the *list-element* is a string, it will be displayed. If the *list-element* is an *expr*, the value of the expression is displayed in the current base. A single character string is displayed as a string. When a single character string is used in an expression, its corresponding hexadecimal value is used in evaluating the expression, and the value of the expression is displayed.

If the *list-element* is BOOL *expr*, ICE displays the boolean value TRUE when the least significant bit (LSB) of the result is 1, FALSE when the LSB is 0; no spaces are provided either before or after the boolean value.
<table>
<thead>
<tr>
<th>ICE-88 KEYWORDS</th>
<th>ABBREVIATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSOLUTE</td>
<td>ABS</td>
</tr>
<tr>
<td>ACKNOWLEDGE</td>
<td>ACK</td>
</tr>
<tr>
<td>ADDR</td>
<td>ADD</td>
</tr>
<tr>
<td>AFL</td>
<td>AFL</td>
</tr>
<tr>
<td>ALL</td>
<td>ALL</td>
</tr>
<tr>
<td>AND</td>
<td>AND</td>
</tr>
<tr>
<td>ASCII</td>
<td>ASC</td>
</tr>
<tr>
<td>ASM</td>
<td>ASM</td>
</tr>
<tr>
<td>BASE</td>
<td>BAS</td>
</tr>
<tr>
<td>BOOL</td>
<td>BOO</td>
</tr>
<tr>
<td>BP</td>
<td>BP</td>
</tr>
<tr>
<td>BR</td>
<td>BR</td>
</tr>
<tr>
<td>BRI</td>
<td>BRI</td>
</tr>
<tr>
<td>BUFFERSIZE</td>
<td>BUF</td>
</tr>
<tr>
<td>BYTE</td>
<td>BYT</td>
</tr>
<tr>
<td>CAUSE</td>
<td>CAU</td>
</tr>
<tr>
<td>CFL</td>
<td>CFL</td>
</tr>
<tr>
<td>CLOCK</td>
<td>CLO</td>
</tr>
<tr>
<td>CONDITIONALLY</td>
<td>CON</td>
</tr>
<tr>
<td>COUNT</td>
<td>COU,C</td>
</tr>
<tr>
<td>CS</td>
<td>CS</td>
</tr>
<tr>
<td>DEFINE</td>
<td>DEF</td>
</tr>
<tr>
<td>DFL</td>
<td>DFL</td>
</tr>
<tr>
<td>DI</td>
<td>DI</td>
</tr>
<tr>
<td>DIR</td>
<td>DIR</td>
</tr>
<tr>
<td>DISABLE</td>
<td>DIS</td>
</tr>
<tr>
<td>DISK</td>
<td>DIS</td>
</tr>
<tr>
<td>DMUX</td>
<td>DMU</td>
</tr>
<tr>
<td>DOMAIN</td>
<td>DOM</td>
</tr>
<tr>
<td>DOWN</td>
<td>DOW</td>
</tr>
<tr>
<td>DS</td>
<td>DS</td>
</tr>
<tr>
<td>ELSE</td>
<td>ELS</td>
</tr>
<tr>
<td>EM</td>
<td>EM</td>
</tr>
<tr>
<td>ENABLE</td>
<td>ENA</td>
</tr>
<tr>
<td>END</td>
<td>END</td>
</tr>
<tr>
<td>ERROR</td>
<td>ERR</td>
</tr>
<tr>
<td>ES</td>
<td>ES</td>
</tr>
<tr>
<td>EVALUATE</td>
<td>EVA</td>
</tr>
<tr>
<td>EXECUTED</td>
<td>EXE,E</td>
</tr>
<tr>
<td>EXIT</td>
<td>EXI</td>
</tr>
<tr>
<td>EXTERNAL</td>
<td>EXT</td>
</tr>
<tr>
<td>FETCHED</td>
<td>FET</td>
</tr>
<tr>
<td>FLAG</td>
<td>FLA</td>
</tr>
<tr>
<td>FOREVER</td>
<td>FOR</td>
</tr>
<tr>
<td>FRAME</td>
<td>FRA</td>
</tr>
<tr>
<td>FROM</td>
<td>FRO,F</td>
</tr>
<tr>
<td>GO</td>
<td>G</td>
</tr>
<tr>
<td>GR</td>
<td>G</td>
</tr>
<tr>
<td>GUARDED</td>
<td>GUA</td>
</tr>
<tr>
<td>H</td>
<td>H</td>
</tr>
<tr>
<td>HALT</td>
<td>HAL,H</td>
</tr>
<tr>
<td>HARDWARE</td>
<td>HAR</td>
</tr>
<tr>
<td>HOLD</td>
<td>HOL</td>
</tr>
<tr>
<td>HTIMER</td>
<td>HTI</td>
</tr>
<tr>
<td>ICE</td>
<td>ICE</td>
</tr>
<tr>
<td>IF</td>
<td>IF</td>
</tr>
<tr>
<td>IFL</td>
<td>IFL</td>
</tr>
<tr>
<td>INC</td>
<td>INC</td>
</tr>
<tr>
<td>INFINITE</td>
<td>INF</td>
</tr>
<tr>
<td>INPUT</td>
<td>INP,I</td>
</tr>
<tr>
<td>INSTRUCTION</td>
<td>INS</td>
</tr>
<tr>
<td>INTEGER</td>
<td>INT</td>
</tr>
<tr>
<td>INTELLEC</td>
<td>INT</td>
</tr>
<tr>
<td>INTERNAL</td>
<td>INT</td>
</tr>
<tr>
<td>IP</td>
<td>IP</td>
</tr>
<tr>
<td>IR</td>
<td>IR</td>
</tr>
<tr>
<td>LOAD</td>
<td>LOA</td>
</tr>
<tr>
<td>MACRO</td>
<td>MAC</td>
</tr>
<tr>
<td>MAP</td>
<td>MAP</td>
</tr>
<tr>
<td>MARK</td>
<td>MAR</td>
</tr>
<tr>
<td>MASM</td>
<td>MAS</td>
</tr>
<tr>
<td>MATCH</td>
<td>MAT</td>
</tr>
<tr>
<td>MN</td>
<td>MN</td>
</tr>
<tr>
<td>MOD</td>
<td>MOD</td>
</tr>
<tr>
<td>MODULE</td>
<td>MOD</td>
</tr>
<tr>
<td>MOVE</td>
<td>MOV,M</td>
</tr>
<tr>
<td>NESTING</td>
<td>NES</td>
</tr>
<tr>
<td>NEWEST</td>
<td>NEW,N</td>
</tr>
<tr>
<td>NMI</td>
<td>NMI</td>
</tr>
<tr>
<td>NOCODE</td>
<td>NOC</td>
</tr>
<tr>
<td>NOERROR</td>
<td>NOE</td>
</tr>
<tr>
<td>NOSYM</td>
<td>NOS</td>
</tr>
<tr>
<td>NOT</td>
<td>NOT</td>
</tr>
<tr>
<td>NOW</td>
<td>NOW</td>
</tr>
<tr>
<td>OBJECT</td>
<td>OBJ</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>OFFSET</td>
<td>OFFSET</td>
</tr>
<tr>
<td>OFFRACE</td>
<td>OFFRACE</td>
</tr>
<tr>
<td>OFL</td>
<td>OFL</td>
</tr>
<tr>
<td>OLDEST</td>
<td>OLD,O</td>
</tr>
<tr>
<td>ON</td>
<td>ON</td>
</tr>
<tr>
<td>ONTRACE</td>
<td>ONTRACE</td>
</tr>
<tr>
<td>OPCODE</td>
<td>OPC</td>
</tr>
<tr>
<td>OR</td>
<td>OR</td>
</tr>
<tr>
<td>ORIF</td>
<td>ORIF</td>
</tr>
<tr>
<td>OUTPUT</td>
<td>OUTPUT</td>
</tr>
<tr>
<td>PFL</td>
<td>PFL</td>
</tr>
<tr>
<td>PIN</td>
<td>PIN</td>
</tr>
<tr>
<td>PIPE</td>
<td>PIPE</td>
</tr>
<tr>
<td>POINTER</td>
<td>POI</td>
</tr>
<tr>
<td>PORT</td>
<td>PORT</td>
</tr>
<tr>
<td>PRINT</td>
<td>PRI,P</td>
</tr>
<tr>
<td>PUT</td>
<td>PUT</td>
</tr>
<tr>
<td>Q</td>
<td>Q</td>
</tr>
<tr>
<td>QDEPTH</td>
<td>QDE</td>
</tr>
<tr>
<td>QSTS</td>
<td>QST</td>
</tr>
<tr>
<td>RAH</td>
<td>RAH</td>
</tr>
<tr>
<td>RAL</td>
<td>RAL</td>
</tr>
<tr>
<td>RAX</td>
<td>RAX</td>
</tr>
<tr>
<td>RBH</td>
<td>RBH</td>
</tr>
<tr>
<td>RBL</td>
<td>RBL</td>
</tr>
<tr>
<td>RBX</td>
<td>RBX</td>
</tr>
<tr>
<td>RCH</td>
<td>RCH</td>
</tr>
<tr>
<td>RCX</td>
<td>RCX</td>
</tr>
<tr>
<td>RDH</td>
<td>RDH</td>
</tr>
<tr>
<td>RDL</td>
<td>RDL</td>
</tr>
<tr>
<td>RDX</td>
<td>RDX</td>
</tr>
<tr>
<td>RDI</td>
<td>RDI</td>
</tr>
<tr>
<td>READ</td>
<td>READ</td>
</tr>
<tr>
<td>REGISTER</td>
<td>REG,R</td>
</tr>
<tr>
<td>REMOVE</td>
<td>REM</td>
</tr>
<tr>
<td>REPEAT</td>
<td>REP</td>
</tr>
<tr>
<td>RESET</td>
<td>RES</td>
</tr>
<tr>
<td>RF</td>
<td>RF</td>
</tr>
<tr>
<td>RST</td>
<td>RST</td>
</tr>
<tr>
<td>SAVE</td>
<td>SAV</td>
</tr>
<tr>
<td>SEGMENT</td>
<td>SEG</td>
</tr>
<tr>
<td>SFL</td>
<td>SFL</td>
</tr>
<tr>
<td>SI</td>
<td>SI</td>
</tr>
<tr>
<td>SINTIGER</td>
<td>SIN</td>
</tr>
<tr>
<td>SP</td>
<td>SP</td>
</tr>
<tr>
<td>SS</td>
<td>SS</td>
</tr>
<tr>
<td>STACK</td>
<td>STA</td>
</tr>
<tr>
<td>STEP</td>
<td>STE,S</td>
</tr>
<tr>
<td>STS</td>
<td>STS</td>
</tr>
<tr>
<td>SYM</td>
<td>SYM</td>
</tr>
<tr>
<td>SYMBOLICALLY</td>
<td>SYM</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>TEST</td>
<td>TEST</td>
</tr>
<tr>
<td>TFL</td>
<td>TFL</td>
</tr>
<tr>
<td>THEN</td>
<td>THE</td>
</tr>
<tr>
<td>TILL</td>
<td>TILL,T</td>
</tr>
<tr>
<td>TIMER</td>
<td>TIM</td>
</tr>
<tr>
<td>TRA</td>
<td>TRA</td>
</tr>
<tr>
<td>TRACE</td>
<td>TRACE</td>
</tr>
<tr>
<td>TYPE</td>
<td>TYP</td>
</tr>
<tr>
<td>UNT</td>
<td>UNT</td>
</tr>
<tr>
<td>UP</td>
<td>UP</td>
</tr>
<tr>
<td>UPPER</td>
<td>UPPER</td>
</tr>
<tr>
<td>USE</td>
<td>USE</td>
</tr>
<tr>
<td>USER</td>
<td>USE</td>
</tr>
<tr>
<td>VALUE</td>
<td>VAL</td>
</tr>
<tr>
<td>WHILE</td>
<td>WHI</td>
</tr>
<tr>
<td>WORD</td>
<td>WORD</td>
</tr>
<tr>
<td>WPORT</td>
<td>WPORT</td>
</tr>
<tr>
<td>WRITE</td>
<td>WRTE</td>
</tr>
<tr>
<td>WRITTEN</td>
<td>WRITEN</td>
</tr>
<tr>
<td>XOR</td>
<td>XOR</td>
</tr>
<tr>
<td>XOR</td>
<td>XOR</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>Z</td>
<td>ZF</td>
</tr>
</tbody>
</table>
The following is a list of error messages.

ERR 10: RSLTS BLK INACCESSIBLE
A bus timeout was detected on an attempt to write the results block.

ERR 11: XMIT BLK INACCESSIBLE
A bus timeout was detected on an attempt to read the transmit block.

ERR 16: DVC CD FORMAT ERROR
The format byte of device code table was determined to be non-zero.

ERR 17: DVC NOT IN DVC CD TABLE
A device code corresponding to this ice was not found in the device code table.

ERR 21: COMMAND NOT ALLOWED NOW
The command code in the parameter block cannot be processed at this time.

ERR 30: PGM MEMORY FAILURE
Data read back from program memory did not agree with data written.

ERR 31: DATA MEMORY FAILURE
Data read back from data memory did not agree with data written.

ERR 32: BREAKPOINT MEM FAILURE
Data read back from breakpoint memory did not agree with data written.

ERR 33: MEMORY MAP FAILURE
Data read back from memory map did not agree with data written.

ERR 34: CABLE FAILURE
Cable diagnostic program detected a failure in the cable.

ERR 35: CONTROL CIRCUIT FAILURE
Control diagnostic program detected a failure in the control circuitry (see note 1).

ERR 36: PAGE FAULT
Not an error. Access was made to disk mapped memory and firmware doesn't have page containing that location.

ERR 37: INTELLEC MEMORY FAILURE
Intellec memory does not verify when written to: it may be missing, non-writable, or bad memory.
ERR 40: NO USER CLOCK
IN EXTERNAL CLOCK MODE, THE CPU CLOCK IS NOT PRESENT.

ERR 41: NO USER VCC
IN EXTERNAL CLOCK MODE, THE USER VCC IS NOT PRESENT.

ERR 42: GUARDED ACCESS
ACCESS WAS MADE TO A GUARDED MEMORY OR I/O LOCATION.

ERR 43: PROCESSOR NOT RUNNING
IN EXTERNAL CLOCK MODE, THE USER READY SIGNAL IS NOT PRESENT.

ERR 48: READY TIMEOUT
IN EXTERNAL CLOCK MODE WITH TIMEOUT ON READY SELECTED, A COMMAND TIMEOUT OCCURRED.

ERR 49: HOLD SEQUENCE ERROR
A HOLD REQUEST WAS INITIATED AND REMOVED BEFORE HOLD ACK BECAME ACTIVE (SEE NOTE 2).

ERR 4A: HOLD TIMEOUT
CANNOT EXIT EMULATION OR EXAMINE USER MEMORY BECAUSE HOLD IS INACTIVE TOO LONG IN THE USER SYSTEM (SEE NOTE 1).

ERR 4B: RESET TIMEOUT
CANNOT EXIT EMULATION BECAUSE RESET IS INACTIVE (SEE NOTE 1).

ERR 80: SYNTAX ERROR
THE TOKEN FLAGGED IS NOT ONE THAT IS ALLOWED IN THE CURRENT CONTEXT.

ERR 81: INVALID TOKEN
THE TOKEN FLAGGED DOES NOT FOLLOW THE RULES FOR A WELL-FORMED TOKEN.

ERR 82: NO SUCH LINE NUMBER
THE SPECIFIED LINE NUMBER DOES NOT EXIST IN THE CURRENT MODULE.

ERR 83: INAPPROPRIATE NUMBER
THE VALUE IS NOT APPROPRIATE IN THE CURRENT CONTEXT.

ERR 84: PARTITION BOUNDS ERROR
THE PARTITION VALUES ENTERED IN A COMMAND ARE NOT CORRECT. EITHER THE LEFT PART OF THE PARTITION IS GREATER THAN THE RIGHT PART OR THE VALUES OF THE PARTITION EXTREMES ARE OUT OF RANGE IN THE CURRENT CONTEXT.

ERR 85: ITEM ALREADY EXISTS
THE ITEM ENTERED IN A DEFINE COMMAND IS CURRENTLY DEFINED IN THE SYMBOL TABLE.

ERR 86: ITEM DOES NOT EXIST
THE ITEM PRINTED ON THE PRECEDING LINE DOES NOT RESIDE IN THE SYMBOL TABLE.
ERR 87: DUPLICATE CHANNEL
THE CHANNEL SPECIFIED APPEARS MORE THAN ONCE IN A CHANNEL LIST.

ERR 88: MACRO PARAMETER ERROR
TOO MANY MACRO PARAMETERS OR MACRO PARAMETER TOO LONG.

ERR 89: MISSING CR-LF IN FILE
INCLUDE FILE DOESN’T END IN CARRIAGE-RETURN LINE-FEED.

ERR 8A: FORMAT ALREADY EXISTS
THE FORMAT SPECIFIED IN A DEFINE COMMAND IS ALREADY DEFINED.

ERR 8B: FORMAT DOES NOT EXIST
THE FORMAT SPECIFIED HAS NOT BEEN DEFINED.

ERR 8C: COMPARE MODE NOT ACTIVE
FIND COMMAND WAS ISSUED WHILE COMPARE TRACE MODE WAS NOT ACTIVE.

ERR 8D: EMPTY TRACE BUFFER
TRACE BUFFER IS UNINITIALIZED.

ERR 8E: INVALID TRACE REFERENCE
TRACE REFERENCE MADE WHILE TRACE BUFFER UNINITIALIZED.

ERR 8F: NON-NULL STRING NEEDED
A NULL STRING WAS USED WHERE A NON-NULL STRING IS REQUIRED.

ERR 90: MEMORY OVERFLOW
MEMORY REQUIREMENTS OF ALL DYNAMIC TABLES EXCEED THE AMOUNT OF MEMORY AVAILABLE.

ERR 91: STACK OVERFLOW
THE CAPACITY OF A STATICALLY ALLOCATED STACK INTERNAL TO THE DIAGNOSTIC PROGRAM HAS BEEN EXCEEDED.

ERR 92: COMMAND TOO LONG
THE CAPACITY OF THE STATICALLY ALLOCATED INTERMEDIATE CODE BUFFER HAS BEEN EXCEEDED.

ERR 93: MODULE DOES NOT EXIST
MODULE SPECIFIED DOES NOT EXIST IN SYMBOL TABLE.

ERR 94: NON-CHANGEABLE ITEM
AN ATTEMPT WAS MADE TO CHANGE AN ITEM THAT MAY NOT BE CHANGED.

ERR 95: INVALID OBJECT FILE
FILE SPECIFIED IN A LOAD COMMAND IS NOT A VALID OBJECT FILE.

ERR 96: INVALID WITHIN ACTIVATE
THE COMMAND IS NOT VALID WITHIN AN ACTIVATE BLOCK.

ERR 97: EXCESSIVE DATA
THE AMOUNT OF DATA ATTEMPTED TO BE INSERTED INTO A PARTITION EXCEEDED THE SIZE OF THE PARTITION.
ERR 98: MORE THAN 16 CHANNELS
MORE THAN 16 CHANNELS SPECIFIED IN A CHANNEL LIST.

ERR 99: EXCESSIVE ITERATED DATA
THE AMOUNT OF DATA TO BE REPEATED THROUHOUT A RANGE
OF MEMORY EXCEEDS THE SIZE OF THE BUFFER ALLOCATED TO
HOLD SUCH DATA.

ERR 9A: TOO MANY GROUPS
NUMBER OF GROUPS DEFINED BY USER MAY NOT EXCEED 43.

ERR 9B: TOO MANY CHANNELS
NUMBER OF CHANNELS DEFINED BY USER MAY NOT EXCEED 128.

ERR 9C: UNSUITABLE EXECUTE FILE
THE FILE REFERENCED IN AN EXECUTE COMMAND EITHER CON­
TAINS CODE THAT IS OUT-OF-BOUNDS FOR THE EXECUTE COM­
MAND OR IT IS A MAIN MODULE.

ERR 9D: LINE TOO LONG
COMMAND LINE WAS LONGER THAN 122 CHARACTERS.

ERR 9E: HOST-ONLY COMMAND
THE COMMAND ISSUED IS NOT ALLOWED IN AN ACTIVATION LIST.

ERR 9F: PROCESS ALREADY ACTIVE
ATTEMPT MADE TO ACTIVATE A PROCESS THAT WAS ALREADY
ACTIVE.

ERR A0: TOO MANY PARTITIONS
NUMBER OF PARTITIONS OR SINGLE BREAKPOINTS IN A BREAK­
POINT REGISTER EXCEED MAXIMUM PERMISSIBLE VALUE.

ERR A1: PARTITION CROSSES PAGE
BREAKPOINT PARTITION WAS NOT CONTAINED ON A SINGLE PAGE.

ERR A2: ILEGAL CLOCK VALUE
VALUE SPECIFIED FOR CLOCK IS NOT A PERMISSIBLE VALUE.

ERR A3: PROCESS ALREADY DORMANT
ATTEMPT MADE TO SUSPEND OR TERMINATE A DORMANT
PROCESS.

ERR A4: MACRO FILE FULL
MACRO FILE CONTAINS MORE THAN 64K CHARACTERS.

ERR A7: POINTER VALUE REQUIRED
A NON-POINTER VALUE WAS USED IN A CONTEXT THAT MUST USE A
POINTER.

ERR A8: INTEGER VALUE REQUIRED
A NON-INTEGER (I.E., POINTER WITH NON-ZERO BASE) VALUE WAS
USED IN A CONTEXT THAT MUST USE AN INTEGER.

ERR A9: CANNOT REDECLARE MAP
AN ATTEMPT WAS MADE TO DECLARE THE DISK MAP AFTER IT WAS
ALREADY DECLARED—THE MAP MUST BE RESET IN ORDER TO
REDECLARE.
ERR AA: MEMORY UNAVAILABLE
THE INTELLEC OR DISK MEMORY EXPLICITLY GIVEN IN A SET-MAP COMMAND WAS NEVER DECLARED; OR NO EXPLICIT MEMORY WAS GIVEN AND THERE IS NO MORE INTELLEC, DISK OR ICE MEMORY AVAILABLE FOR THE ICE-88 EMULATOR TO ASSIGN.

ERR AC: TAKES TOO MANY BRs
A MATCH CONDITION WAS GIVEN THAT REQUIRES MORE BREAK-POINT REGISTERS THAN IS ALLOWED IN THE CURRENT CONTEXT; EITHER IT REQUIRED MORE THAN ONE REGISTER IN A SET BREAK-POINT COMMAND, OR REQUIRED MORE THAN TWO REGISTERS IN A TILL CLAUSE.

ERR AD: DIFFERING BASES
TWO POINTERS WITH DIFFERENT BASES WERE USED IN A CONTEXT WHERE THEY MUST HAVE THE SAME BASE; E.G., THE LOWER AND UPPER BOUNDS OF A PARTITION.

ERR AE: INVALID "AND" IN GO-REG
THE GO-REGISTER IS "TILL BR0 AND BR1" DURING A GO COMMAND BUT EITHER (1) BR0 OR BR1 CONTAINS AN EXECUTION-TYPE MATCH CONDITION OR (2) BR0 CONTAINS A DATA-TIME CONDITION AND BR1 CONTAINS AN ADDRESS-TIME CONDITION.

ERR B2: INVALID BASE
THE BASE USED IN THE DISPLAY BREAKPOINT/TRACEPOINT COMMAND IS OUT OF RANGE FOR PART OR ALL OF THE ADDRESSES IN THE REGISTER (E.G., "BR0 BASE 0" WHEN BR0 CONTAINS ADDRESS 10000H).

ERR B3: SYMBOl HAS NO TYPE
A SYMBOL BEING USED IN A TYPED MEMORY REFERENCE (E.G., "!'X'") HAS NO TYPE.

ERR B5: BLOCK IS EMPTY

WARN C0: UNSATISFIED EXTERNALS
THE PROGRAM JUST LOADED CONTAINS EXTERNALS WHICH WERE NOT SATISFIED AT LINK TIME. THE PROGRAM WAS LOADED CORRECTLY EXCEPT FOR REFERENCES TO THE UNSATISFIED EXTERNALS.

WARN C1: MAPPING OVER SYSTEM
THE USER HAS MODIFIED THE MAP SO THAT PART OF HIS ADDRESS SPACE INCLUDES EITHER THE ISIS SYSTEM OR THE GID SOFTWARE PACKAGE.

WARN C2: HARDWARE MISSING
AT ATTEMPT WAS MADE TO INITIALIZE THE DEVICE WHOSE GENERIC DEVICE CODE NUMBER IS PRINTED ON THE PREVIOUS LINE BUT NO DEVICE RESPONDED. A GENERIC DEVICE CODE IS THE FIRST OF FOUR CONSECUTIVE DEVICE CODES RESERVED FOR A SPECIFIC TYPE OF DEVICE.

WARN C3: MULTIPLE HARDWARE
AN ATTEMPT WAS MADE TO INITIALIZE THE DEVICE WHOSE DEVICE CODE NUMBER IS PRINTED ON THE PREVIOUS LINE BUT MORE THAN ONE DEVICE RESPONDED.
WARN C4: INVALID "AND" IN GR
THE GO-REGISTER IS AS DESCRIBED FOR ERR AE AFTER GR, BR0 OR
BR1 WAS CHANGED.

WARN C5: INTELLEC MEM FAILURE
THE INTELLEC MEMORY WHOSE PHYSICAL SEGMENT NUMBER IS
ON PREVIOUS LINE DOES NOT VERIFY WHEN WRITTEN TO: IT MAY
BE MISSING, NON-WRITABLE, OR BAD MEMORY.

WARN C6: HARDWARE REINITIALIZED
THE HARDWARE HAS BEEN REINITIALIZED, CLEARING THE MAP
AND MAKING TRACE UNCONDITIONALLY ON.

WARN C7: CLEARING TFL TO 0

WARN C8: REINITIALIZING—FAULT
THE HARDWARE IS BEING REINITIALIZED

I ERR E7: ILLEGAL FILENAME [4]
THE FILENAME SPECIFIED DOES NOT CONFORM TO A WELL-
FORMED ISIS FILENAME.

I ERR E8: ILLEGAL DEVICE [5]
ILLEGAL OR UNRECOGNIZED DEVICE IN FILENAME.

ATTEMPT TO WRITE TO A FILE OPEN FOR INPUT.

I ERR EB: FILE OPEN FOR OUTPUT [8]

I ERR EC: DIRECTORY FULL [9]

I ERR EE: FILE ALREADY IN USE [11]

I ERR EF: FILE ALREADY OPEN [12]
ATTEMPT TO OPEN A FILE THAT WAS ALREADY OPEN.

I ERR F0: NO SUCH FILE [13]
THE FILE SPECIFIED DOES NOT EXIST.

I ERR F1: WRITE-PROTECT FILE [14]
ATTEMPT TO OPEN A WRITE-PROTECTED FILE FOR THE PURPOSES
OF WRITING DATA INTO IT.

I ERR F3: CHECKSUM ERROR [16]
A CHECKSUM ERROR IN A HEX OBJECT FILE WAS ENCOUNTERED
DURING LOADING.

I ERR F6: DISK FILE REQUIRED [19]
ATTEMPT TO USE A NON-DISKETTE FILE WHERE A DISKETTE FILE
WAS REQUIRED.

I ERR F9: ILLEGAL ACCESS [22]
ATTEMPT TO OPEN A READ-ONLY FILE FOR THE PURPOSES OF
STORING DATA (I.E., SPECIFYING :CI: AS THE LIST DEVICE) OR A
WRITE-ONLY FILE AS A SOURCE OF DATA (I.E., :LP: IN A LOAD
COMMAND).
ERR FA: NO FILE NAME [23]
   NO FILENAME SPECIFIED FOR A DISKETTE FILE (I.E., NO FILENAME FOLLOWING :F1:).

ERR FD: "DONE" TIMED OUT
   THE DEVICE WHOSE DEVICE CODE NUMBER IS PRINTED ON THE PRECEDING LINE WAS INVOKED BUT FAILED TO RETURN DONE WITHIN FIVE SECONDS.

ERR FE: "ACKNOWLEDGE" TIMED OUT
   THE DEVICE WHOSE DEVICE CODE NUMBER IS PRINTED ON THE PRECEDING LINE WAS INVOKED BUT FAILED TO ACKNOWLEDGE WITHIN 5 MILLISECONDS.

ERR FF: NULL FILE EXTENSION [28]
   A FILE WAS SPECIFIED SO AS TO CONTAIN AN EXTENSION BUT NO EXTENSION WAS SPECIFIED.

Note 1. If error 35, 4A, or 4B occurs during emulation, hardware will be reset as if a RESET HARDWARE was executed. The emulation will not be recoverable as all registers will be set to the values they contained at the beginning of emulation. Warning message C6 may be issued.

Note 2. Error 49 will cause emulation to exit properly and warning message C6 will be issued.

Note 3. Bracketed number following error message refers to the ISIS error identified by this number.

Note 4. Error messages other than those documented in this list should not occur. If you encounter such an error, please report it to Intel Corporation, MCSD Customer Marketing, 3065 Bowers Avenue, Santa Clara, CA 95051, or to your local Field Application Engineer.
APPENDIX C
ICE-88 COMMAND SYNTAX SUMMARY

Command Summary

debg session = [top-level command cr] ...
top-level command = define macro command :: remove macro command :: command
command = compound command :: simple command
compound command = if command :: repeat command :: count command :: write command
simple command = display break/trace command :: set break/trace command ::
go command :: step command :: go-register command ::
enable/disable trace command :: trace command :: oldest command ::
newest command :: print command :: move command :: clock command ::
command signal timeout command :: enable/disable ready command ::
display command :: change command :: define command ::
display symbols command :: display lines command ::
display modules command :: change symbol command ::
remove symbols command :: set domain command ::
reset domain command :: display map command ::
declare map command :: set map command :: reset map command ::
load command :: save command :: suffix command :: base command ::
evaluate command :: list command :: exit command ::
reset hardware command :: display macro command ::
put macro command :: dir command :: include command

Expressions

expr = boolean term [or-op boolean term] ...
or-op = OR :: XOR
boolean term = boolean factor [AND boolean factor] ...
boolean factor = [NOT] boolean primary
boolean primary = arith expr [rel-op arith expr]
rel-op = < :: > :: <= :: >= :: <> :: =
arith expr = memory reference :: port name :: address
address = arith term [MASK arith term] ...
arith term = term [plus-op term] ...
plus-op = + :: -
term = factor [mult-op factor] ...
mult-op = * :: / :: MOD
factor = [plus-op] [segment-op] primary
segment-op = primary :: OFFSET :: SEGMENT
primary = (expr) :: numeric constant :: source statement number :: string ::
symbolic reference :: keyword reference
symbolic reference = [module name] symbol [symbol] ...
module name = .identifier
symbol = .identifier
source statement number \( = [\text{module name}] \# \text{primary-10} \)

\text{primary-10} \equiv \text{primary}

keyword reference \( = \text{register name} :: \text{flag name} :: \text{pin name} :: \text{typed memory reference} \)

partition \( = \text{address} \to \text{address} :: \text{address LENGTH address} \)

**Keyword Operators**

\text{register name} \( = \text{RAL} :: \text{RAH} :: \text{RBL} :: \text{RBH} :: \text{RCL} :: \text{RCH} :: \text{RDL} :: \text{RDH} :: \text{RA} :: \text{RB} :: \text{RC} :: \text{RD} :: \text{RDX} :: \text{RSP} :: \text{RB} :: \text{SI} :: \text{DI} :: \text{SS} :: \text{DS} :: \text{ES} :: \text{IP} :: \text{CAUSE} :: \text{OPCODE} :: \text{RF} :: \text{PI} :: \text{TIMER} :: \text{HTIMER} :: \text{BUFFERSIZE} :: \text{UPPER} :: \text{LOWER} \)

\text{flag name} \( = \text{AFL} :: \text{CFL} :: \text{DFL} :: \text{IFL} :: \text{OF} :: \text{PFL} :: \text{SFL} :: \text{TFL} :: \text{ZFL} \)

\text{pin name} \( = \text{RDY} :: \text{NMI} :: \text{TEST} :: \text{HOLD} :: \text{RST} :: \text{MN} :: \text{IR} \)

\text{port name} \( = \text{PORT address} :: \text{WPORT address} \)

memory reference \( = \text{memory-designation address} \)

memory-designation \( = \text{BYTE} :: \text{WORD} :: \text{SINTEGER} :: \text{INTEGER} :: \text{POINTER} \)

typed memory reference \( = [!] \text{identifier} [!] \text{identifier} [!] \text{identifier} ... \)

**Emulation Controls and Commands**

display break/trace command \( = \text{break/trace reg} \) [display break/trace mode]

set break/trace command \( = \text{break/trace reg} = \text{match-cond} \)

break/trace reg \( = \text{break reg} :: \text{trace reg} \)

break reg \( = \text{BR} :: \text{BR0} :: \text{BR1} \)

trace reg \( = \text{ONTRACE} :: \text{OFFTRACE} \)

display break/trace mode \( = \text{ABSOLUTE} :: \text{BASE [expr]} \)

match-cond \( = \text{execution match code} :: \text{non-execution match cond} \)

execution match cond \( = \text{match value EXECUTED} \)

non-execution match cond \( = \text{address match range [match status list] [data match range]} \)

\[ \text{segment register usage} :: \text{match status list} [\text{data match range}] \]

\[ \text{segment register usage} :: \text{data match range} \]

\[ \text{segment register usage} :: \text{segment register usage} \]

match value \( = \text{address :: masked constant} \)

address match range \( = \text{match range} \)

data match range \( = \text{VALUE match range} \)

match range \( = \text{match value :: match partition [, match partition] ... :: address up/down} \)

match partition \( = \text{partition :: OBJECT memory reference :: OBJECT typed memory reference} \)

up/down \( = \text{UP :: DOWN} \)

match status list \( = \text{match status [, match status] ...} \)

match status \( = \text{READ :: WRITTEN :: INPUT :: OUTPUT :: FETCHED :: HALT :: ACKNOWLEDGE} \)

segment register usage \( = \text{USING segment register name} \)

segment register name \( = \text{SS} :: \text{CS} :: \text{DS} :: \text{ES} \)

go command \( = \text{GO [FROM address]} [\text{go-register}] \)

step command \( = \text{STEP [FROM address]} \)

go-register command \( = \text{GR [= go-register]} \)
Command Syntax Summary

**go-register** = FOREVER :: TILL break

**break** = break reg [and/or break reg] :: match-cond [and/or match-cond]

and/or = AND :: OR

**enable/disable trace command** = ENABLE TRACE [CONDITIONALLY [NOW initial trace ] ] ::
DISABLE TRACE

**initial trace** = ON :: OFF

**trace command** = TRACE [ = trace mode ]

**trace mode** = FRAME :: INSTRUCTION

**oldest command** = OLDEST

**newest command** = NEWEST

**print command** = PRINT [ [plus-op] primary-10 ] :: PRINT ALL

**move command** = MOVE [ [plus-op] primary-10 ]

**clock command** = CLOCK [ = clock setting ]

**clock setting** = INTERNAL :: EXTERNAL

**command signal timeout command** = RWTIMEOUT [ = new signal ]

**new signal** = INFINITE :: expr-10 [ERROR] :: expr-10 NOERROR

expr-10 = expr

**enable/disable ready command** = ENABLE RDY :: DISABLE RDY

---

**Interrogation and Utility Commands**

display command = reference [, reference] ... :: mem or i/o partition :: ASM partition ::
REGISTER :: FLAG :: PIN :: STACK expr :: BOOL expr

mem or i/o = memory-designation :: PORT :: WPORT

change command = reference = expr :: mem or i/o partition = change exp [, change exp] ...

change exp = mem or i/o partition :: expr :: string

define command = DEFINE [module name] symbol = expr [OF type]

display symbols command = SYMBOL :: symbolic reference

display lines command = LINE :: source statement number

display modules command = MODULE

change symbol command = symbolic reference = expr [OF type]

remove symbols command = REMOVE symbolic reference [, symbolic reference] ... ::
 REMOVE SYMBOL ::
 REMOVE MODULE module name [, module name] ...

**type** = memory design

set domain command = DOMAIN module name

reset domain command = RESET DOMAIN

display map command = MAP [partition]

declare map command = MAP DISK = file name :: MAP INTELLEC = partition [, partition] ...

set map command = MAP partition = new memory map

reset map command = RESET MAP

new memory map = GUARDED :: USER [NOVERIFY] :: ICE [address] [NOVERIFY] ::
INTELLEC [address] [NOVERIFY] :: DISK [address] [NOVERIFY]

---

C-3
load command = LOAD path name {NOCODE
NOSYMBOL
NOLINE}
save command = SAVE path name {save code
NOSYMBOL
NOLINE}
save code = NOCODE :: partition [, partition] ...
suffix command = SUFFIX [= suffix]
base command = BASE [= base]
suffix = Y :: O :: Q :: T :: H
base = suffix :: ASCII
display nesting command = NESTING
evaluate command = EVALUATE expr [SYMBOLICALLY]
list command = LIST path name
exit command = EXIT
reset hardware command = RESET HARDWARE
cr = carriage-return line-feed

Macro Definition Command
define macro command = DEFINE MACRO macro name cr macro body EM
macro name = identifier
macro body = [command cr] ...

IF Command
if command = IF expr [THEN] cr true-list
[ORIF expr [THEN] cr true-list] ...
[ELSE cr false-list]
true-list = [command cr] ...
false-list = [command cr] ...
end if = END

Looping Commands
repeat command = REPEAT cr loop-list end-repeat
end-repeat = END
count command = COUNT expr-10 cr loop-list end-count
eexpr-10 = expr
end-count = END
loop-list = [loop element cr] ...
loop element = command :: loop exit
loop exit = WHILE EXPRESSION :: UNTIL expr
Macro Invocation Command

\[ \text{macro invocation command} = \text{macro name} :: \text{[actual parameter list]} \]
\[ \text{actual parameter list} = \text{actual parameter [, actual parameter]} \ldots \]
\[ \text{actual parameter} = \text{[limited token]} \ldots :: \text{string} \]
\[ \text{limited token} = \text{any token except cr, string or ","} \]

Remove Macro Command

\[ \text{remove macro command} = \text{REMOVE MACRO [macro list]} \]
\[ \text{macro list} = \text{macro name [, macro name]} \ldots \]

Display Macro Command

\[ \text{display macro command} = \text{MACRO [macro list]} \]

Put Macro Command

\[ \text{put macro command} = \text{PUT file name MACRO [macro list]} \]

Director Command

\[ \text{dir command} = \text{DIR directory} \]
\[ \text{directory} = \text{MACRO} \]

Include Command

\[ \text{include command} = \text{INCLUDE file name} \]

Write Command

\[ \text{write command} = \text{WRITE list element [, list element]} \ldots \]
\[ \text{list element} = \text{string :: expr :: BOOL expr} \]
DC Characteristics of ICE-88 User Cable

1. Output Low Voltages

<table>
<thead>
<tr>
<th>Pin</th>
<th>$V_{OL}(\text{Max})$</th>
<th>$I_{OL}(\text{Min})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0-A7, A8-A15</td>
<td>0.4V</td>
<td>8 mA</td>
</tr>
<tr>
<td>A16/S3-A19/S6, SSO, RD, LOCK, QS0, QS1, S0, S1, S2, WR, IO/M, DT/R, DEN, ALE, INTA</td>
<td>0.4V</td>
<td>8 mA</td>
</tr>
<tr>
<td>HLDA</td>
<td>0.4V</td>
<td>7 mA</td>
</tr>
<tr>
<td>MATCH0</td>
<td>0.4V</td>
<td>16 mA</td>
</tr>
</tbody>
</table>

2. Output High Voltages

<table>
<thead>
<tr>
<th>Pin</th>
<th>$V_{OH}(\text{Min})$</th>
<th>$I_{OH}(\text{Min})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0-A7, A8-A15</td>
<td>2.4V</td>
<td>~2 mA</td>
</tr>
<tr>
<td>A16/S3-A19/S6, SSO, RD, LOCK, QS0, QS1, S0, S1, S2, WR, IO/M, DT/R, DEN, ALE, INTA</td>
<td>2.4V</td>
<td>~2 mA</td>
</tr>
<tr>
<td>HLDA</td>
<td>2.4V</td>
<td>~3.0 mA</td>
</tr>
<tr>
<td>MATCH0 OR MATCH1</td>
<td>2.4V</td>
<td>~0.8 mA</td>
</tr>
</tbody>
</table>

3. Input Low Voltages

<table>
<thead>
<tr>
<th>Pin</th>
<th>$V_{IL}(\text{Max})$</th>
<th>$I_{IL}(\text{Max})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0-A7, A8-A15</td>
<td>0.8V</td>
<td>~0.2 mA</td>
</tr>
<tr>
<td>NMI, CLK</td>
<td>0.8V</td>
<td>~0.4 mA</td>
</tr>
<tr>
<td>READY</td>
<td>0.8V</td>
<td>~0.8 mA</td>
</tr>
<tr>
<td>INTR, HOLD, TEST, RESET</td>
<td>0.8V</td>
<td>~1.4 mA</td>
</tr>
<tr>
<td>MN/MX (0.1 $\mu$F to GND)</td>
<td>0.8V</td>
<td>~3.3 mA</td>
</tr>
</tbody>
</table>

4. Input High Voltages

<table>
<thead>
<tr>
<th>Pin</th>
<th>$V_{IH}(\text{Min})$</th>
<th>$I_{IH}(\text{Max})$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0-A7, A8-A15</td>
<td>2.0V</td>
<td>80 $\mu$A</td>
</tr>
<tr>
<td>NMI, CLK</td>
<td>2.0V</td>
<td>20 $\mu$A</td>
</tr>
<tr>
<td>READY</td>
<td>2.0V</td>
<td>60 $\mu$A</td>
</tr>
<tr>
<td>INTR, HOLD, TEST, RESET</td>
<td>2.0V</td>
<td>~0.4 mA</td>
</tr>
<tr>
<td>MN/MX</td>
<td>2.0V</td>
<td>~1.1 mA</td>
</tr>
</tbody>
</table>

5. RQ/GT0, RQ/GT1 are pulled up to +5V through a 5.6K ohm resistor. No current is taken from user circuit at $V_{CC}$ pin.

Specifications

ICE-88 Operating Environment

Required Hardware:
- Intellic Microcomputer Development System with:
  1. Three adjacent slots for ICE-88
  2. 64K of Intellic Memory. If expansion memory is desired, no more than 32K may be 16K RAM boards.
- System Console
- Intellic Diskette Operating System
- ICE-88 Module
Electrical and Physical Characteristics

Required Software:
- System Monitor
- ISIS-II, Version 3.4 or subsequent
- ICE-88 Software

Equipment Supplied
- Printed Circuit Boards (3)
- Interface Cable and Emulation Buffer Module
- Operator’s Manual
- ICE-88 Software

Emulation Clock
User system clock up to 5 MHz or 2 MHz internal clock in stand-alone mode.

Physical Characteristics
Printed Circuit Boards:

| Width | 12.00 in | (30.48 cm) |
| Height | 6.75 in | (17.15 cm) |
| Depth | 0.50 in | (1.27 cm) |
| Packaged Weight | — | — |

Electrical Characteristics

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$V_{CC}$</td>
<td>$+5V \pm 1%$</td>
</tr>
<tr>
<td>$I_{CC}$</td>
<td>16A maximum; 11A typical</td>
</tr>
<tr>
<td>$V_{DD}$</td>
<td>$+12V \pm 5%$</td>
</tr>
<tr>
<td>$I_{DD}$</td>
<td>120 mA maximum; 80 mA typical</td>
</tr>
<tr>
<td>$V_{BB}$</td>
<td>$-10V \pm 5%$ or $-12V \pm 5%$ (optional)</td>
</tr>
<tr>
<td>$I_{BB}$</td>
<td>15 mA maximum; 12 mA typical</td>
</tr>
</tbody>
</table>

Environmental Characteristics

Operating Temperature: 0° to 40°C
Operating Humidity: Up to 95% relative humidity without condensation.
### Instructions in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hexadecimal</th>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00 00000000</td>
<td>MOD REG/R/M ADD EA,REG</td>
<td>BYTE ADD (REG) TO EA</td>
</tr>
<tr>
<td>01 00000001</td>
<td>MOD REG/R/M ADD EA,REG</td>
<td>WORD ADD (REG) TO EA</td>
</tr>
<tr>
<td>02 00000010</td>
<td>MOD REG/R/M ADD REG,EA</td>
<td>BYTE ADD (EA) TO REG</td>
</tr>
<tr>
<td>03 00000011</td>
<td>MOD REG/R/M ADD REG,EA</td>
<td>WORD ADD (EA) TO REG</td>
</tr>
<tr>
<td>04 00000100</td>
<td>ADD AL,DATA8</td>
<td>BYTE ADD DATA TO REG AL</td>
</tr>
<tr>
<td>05 00000101</td>
<td>ADD AX,DATA16</td>
<td>WORD ADD DATA TO REG AX</td>
</tr>
<tr>
<td>06 00000110</td>
<td>PUSH ES</td>
<td>PUSH (ES) ON STACK</td>
</tr>
<tr>
<td>07 00000111</td>
<td>POP ES</td>
<td>POP STACK TO REG ES</td>
</tr>
<tr>
<td>08 00010000</td>
<td>MOD REG/R/M OR EA,REG</td>
<td>BYTE OR (REG) TO EA</td>
</tr>
<tr>
<td>09 00010010</td>
<td>MOD REG/R/M OR REG,EA</td>
<td>WORD OR (REG) TO EA</td>
</tr>
<tr>
<td>0A 00010110</td>
<td>MOD REG/R/M OR REG,EA</td>
<td>WORD OR (EA) TO REG</td>
</tr>
<tr>
<td>0C 00011000</td>
<td>OR AL,DATA8</td>
<td>BYTE OR DATA TO REG AL</td>
</tr>
<tr>
<td>0D 00011010</td>
<td>OR AX,DATA16</td>
<td>WORD OR DATA TO REG AX</td>
</tr>
<tr>
<td>0E 00011100</td>
<td>PUSH CS</td>
<td>PUSH (CS) ON STACK</td>
</tr>
<tr>
<td>0F 00011111</td>
<td>(not used)</td>
<td></td>
</tr>
<tr>
<td>10 00100000</td>
<td>MOD REG/R/M ADC EA,REG</td>
<td>BYTE ADD (REG) W/ CARRY TO EA</td>
</tr>
<tr>
<td>11 00100001</td>
<td>MOD REG/R/M ADC EA,REG</td>
<td>WORD ADD (REG) W/ CARRY TO EA</td>
</tr>
<tr>
<td>12 00100100</td>
<td>MOD REG/R/M ADC REA,EA</td>
<td>BYTE ADD (EA) W/ CARRY TO REG</td>
</tr>
<tr>
<td>13 00100110</td>
<td>MOD REG/R/M ADC REG,EA</td>
<td>WORD ADD (EA) W/ CARRY TO REG</td>
</tr>
<tr>
<td>14 00101000</td>
<td>ADC AL,DATA8</td>
<td>WORD ADD DATA W/ CARRY TO REG AL</td>
</tr>
<tr>
<td>15 00101010</td>
<td>ADC AX,DATA16</td>
<td>WORD ADD DATA W/ CARRY TO REG AX</td>
</tr>
<tr>
<td>16 00101100</td>
<td>PUSH SS</td>
<td>PUSH (SS) ON STACK</td>
</tr>
<tr>
<td>17 00101111</td>
<td>POP SS</td>
<td>POP STACK TO REG SS</td>
</tr>
<tr>
<td>18 00110000</td>
<td>MOD REG/R/M SBB EA,REG</td>
<td>BYTE SUB (REG) W/ BORROW FROM EA</td>
</tr>
<tr>
<td>19 00110010</td>
<td>MOD REG/R/M SBB EA,REG</td>
<td>WORD SUB (REG) W/ BORROW FROM EA</td>
</tr>
<tr>
<td>1A 00110100</td>
<td>MOD REG/R/M SBB REG,EA</td>
<td>WORD SUB (EA) W/ BORROW FROM REG</td>
</tr>
<tr>
<td>1B 00110110</td>
<td>MOD REG/R/M SBB REG,EA</td>
<td>WORD SUB (EA) W/ BORROW FROM REG</td>
</tr>
<tr>
<td>1C 00111000</td>
<td>SBB AL,DATA8</td>
<td>BYTE SUB DATA W/ BORROW FROM REG AL</td>
</tr>
<tr>
<td>1D 00111010</td>
<td>SBB AX,DATA16</td>
<td>WORD SUB DATA W/ BORROW FROM REG AX</td>
</tr>
<tr>
<td>1E 00111100</td>
<td>PUSH DS</td>
<td>PUSH (DS) ON STACK</td>
</tr>
<tr>
<td>1F 00111111</td>
<td>POP DS</td>
<td>POP STACK TO REG DS</td>
</tr>
<tr>
<td>20 00100000</td>
<td>MOD REG/R/M AND EA,REG</td>
<td>BYTE AND (REG) TO EA</td>
</tr>
<tr>
<td>21 00100001</td>
<td>MOD REG/R/M AND EA,REG</td>
<td>WORD AND (REG) TO EA</td>
</tr>
<tr>
<td>22 00100010</td>
<td>MOD REG/R/M AND REG,EA</td>
<td>BYTE AND (EA) TO REG</td>
</tr>
<tr>
<td>23 00100100</td>
<td>MOD REG/R/M AND REG,EA</td>
<td>WORD AND (EA) TO REG</td>
</tr>
<tr>
<td>24 00101000</td>
<td>AND AL,DATA8</td>
<td>BYTE AND DATA TO REG AL</td>
</tr>
<tr>
<td>25 00101010</td>
<td>AND AX,DATA16</td>
<td>WORD AND DATA TO REG AX</td>
</tr>
<tr>
<td>26 00101100</td>
<td>ES:</td>
<td>SEGMENT OVERIDE W/ SEGMENT REG ES</td>
</tr>
<tr>
<td>27 00101111</td>
<td>DAA</td>
<td>DECIMAL ADJUST FOR ADD</td>
</tr>
<tr>
<td>28 00101000</td>
<td>MOD REG/R/M SUB EA,REG</td>
<td>BYTE SUBTRACT (REG) FROM EA</td>
</tr>
<tr>
<td>29 00101010</td>
<td>MOD REG/R/M SUB EA,REG</td>
<td>WORD SUBTRACT (REG) FROM EA</td>
</tr>
<tr>
<td>2A 00101100</td>
<td>MOD REG/R/M SUB REG,EA</td>
<td>BYTE SUBTRACT (EA) FROM REG</td>
</tr>
<tr>
<td>2B 00101110</td>
<td>MOD REG/R/M SUB AL,DATA8</td>
<td>BYTE SUBTRACT DATA FROM REG AL</td>
</tr>
<tr>
<td>2C 00110010</td>
<td>SUB AX,DATA16</td>
<td>WORD SUBTRACT DATA FROM REG AX</td>
</tr>
<tr>
<td>2D 00110100</td>
<td>CS:</td>
<td>SEGMENT OVERIDE W/ SEGMENT REG CS</td>
</tr>
<tr>
<td>2E 00110110</td>
<td>DAS</td>
<td>DECIMAL ADJUST FOR SUBTRACT</td>
</tr>
<tr>
<td>2F 00111111</td>
<td>(not used)</td>
<td></td>
</tr>
<tr>
<td>30 00100000</td>
<td>MOD REG/R/M XOR EA,REG</td>
<td>BYTE XOR (REG) TO EA</td>
</tr>
<tr>
<td>31 00100001</td>
<td>MOD REG/R/M XOR EA,REG</td>
<td>WORD XOR (REG) TO EA</td>
</tr>
<tr>
<td>32 00100100</td>
<td>MOD REG/R/M XOR REG,EA</td>
<td>BYTE XOR (EA) TO REG</td>
</tr>
<tr>
<td>33 00100110</td>
<td>MOD REG/R/M XOR REG,EA</td>
<td>WORD XOR (EA) TO REG</td>
</tr>
<tr>
<td>34 00101000</td>
<td>XOR AL,DATA8</td>
<td>BYTE XOR DATA TO REG AL</td>
</tr>
<tr>
<td>35 00101010</td>
<td>XOR AX,DATA16</td>
<td>WORD XOR DATA TO REG AX</td>
</tr>
<tr>
<td>36 00101100</td>
<td>SS:</td>
<td>SEGMENT OVERIDE W/ SEGMENT REG SS</td>
</tr>
<tr>
<td>37 00101111</td>
<td>AAA</td>
<td>ASCII ADJUST FOR ADD</td>
</tr>
<tr>
<td>38 00110000</td>
<td>MOD REG/R/M CMP EA,REG</td>
<td>BYTE COMPARE (EA) WITH (REG)</td>
</tr>
<tr>
<td>39 00110010</td>
<td>MOD REG/R/M CMP EA,REG</td>
<td>WORD COMPARE (EA) WITH (REG)</td>
</tr>
<tr>
<td>3A 00110100</td>
<td>MOD REG/R/M CMP REG,EA</td>
<td>BYTE COMPARE (REG) WITH (EA)</td>
</tr>
<tr>
<td>3B 00110110</td>
<td>MOD REG/R/M CMP REG,EA</td>
<td>WORD COMPARE (REG) WITH (EA)</td>
</tr>
<tr>
<td>3C 00111000</td>
<td>CMP AL,DATA8</td>
<td>BYTE COMPARE DATA WITH (AL)</td>
</tr>
<tr>
<td>3D 00111010</td>
<td>CMP AX,DATA16</td>
<td>WORD COMPARE DATA WITH (AX)</td>
</tr>
<tr>
<td>3E 00111100</td>
<td>DS:</td>
<td>SEGMENT OVERIDE W/ SEGMENT REG DS</td>
</tr>
<tr>
<td>3F 00111111</td>
<td>AAS</td>
<td>ASCII ADJUST FOR SUBTRACT</td>
</tr>
<tr>
<td>40 01000000</td>
<td>INC AX</td>
<td>INCREMENT (AX)</td>
</tr>
<tr>
<td>41 01000001</td>
<td>INC CX</td>
<td>INCREMENT (CX)</td>
</tr>
</tbody>
</table>
Instructions in Hexadecimal Order

8088 Assembly Language

42 01000010  INC  DX  INCREMENT (DX)
43 01000011  INC  BX  INCREMENT (BX)
44 01000100  INC  SP  INCREMENT (SP)
45 01000101  INC  BP  INCREMENT (BP)
46 01000110  INC  SI  INCREMENT (SI)
47 01000111  INC  DI  INCREMENT (DI)
48 01001000  DEC  AX  DECREMENT (AX)
49 01001001  DEC  CX  DECREMENT (CX)
4A 01001010  DEC  DX  DECREMENT (DX)
4B 01001011  DEC  BX  DECREMENT (BX)
4C 01001100  DEC  SP  DECREMENT (SP)
4D 01001101  DEC  BP  DECREMENT (BP)
4E 01001110  DEC  SI  DECREMENT (SI)
4F 01001111  DEC  DI  DECREMENT (DI)
50 01010000  PUSH AX  PUSH (AX) ON STACK
51 01010001  PUSH CX  PUSH (CX) ON STACK
52 01010010  PUSH DX  PUSH (DX) ON STACK
53 01010011  PUSH BX  PUSH (BX) ON STACK
54 01010100  PUSH SP  PUSH (SP) ON STACK
55 01010101  PUSH BP  PUSH (BP) ON STACK
56 01010110  PUSH SI  PUSH (SI) ON STACK
57 01010111  PUSH DI  PUSH (DI) ON STACK
58 01011000  POP AX  POP STACK TO REG AX
59 01011001  POP CX  POP STACK TO REG CX
5A 01011010  POP DX  POP STACK TO REG DX
5B 01011011  POP BX  POP STACK TO REG BX
5C 01011100  POP SP  POP STACK TO REG SP
5D 01011101  POP BP  POP STACK TO REG BP
5E 01011110  POP SI  POP STACK TO REG SI
5F 01011111  POP DI  POP STACK TO REG DI
60 01100000  (not used)
61 01100001  (not used)
62 01100010  (not used)
63 01100011  (not used)
64 01100100  (not used)
65 01100101  (not used)
66 01100110  (not used)
67 01100111  (not used)
68 01101000  (not used)
69 01101001  (not used)
6A 01101010  (not used)
6B 01101011  (not used)
6C 01101100  (not used)
6D 01101101  (not used)
6E 01101110  (not used)
6F 01101111  (not used)
70 01110000  JO  DISP8  JUMP ON OVERFLOW
71 01110001  JNO  DISP8  JUMP ON NOT OVERFLOW
72 01110010  JB /JNAE  DISP8  JUMP ON BELOW / NOT ABOVE OR EQUAL
73 01110011  JNB/JAE  DISP8  JUMP ON NOT BELOW / ABOVE OR EQUAL
74 01110100  JE/ JZ  DISP8  JUMP ON EQUAL /ZERO
75 01110101  JNE/ JNZ  DISP8  JUMP ON NOT EQUAL / NOT ZERO
76 01110110  JBE/JNA  DISP8  JUMP ON BELOW OR EQUAL / NOT ABOVE
77 01110111  JNB/JA  DISP8  JUMP ON NOT BELOW OR EQUAL / ABOVE
78 01111000  JS  DISP8  JUMP ON SIGN
79 01111001  JNS  DISP8  JUMP ON NOT SIGN
7A 01111010  JP /JPE  DISP8  JUMP ON PARITY / PARITY EVEN
7B 01111011  JNP/JPO  DISP8  JUMP ON NOT PARITY / PARITY ODD
7C 01111100  JL/JNGE  DISP8  JUMP ON LESS / NOT GREATER OR EQUAL
7D 01111101  JNL/JGE  DISP8  JUMP ON NOT LESS / GREATER OR EQUAL
7E 01111110  JLE/JNG  DISP8  JUMP ON LESS OR NOT EQUAL / NOT GREATER
7F 01111111  JNL/EJG  DISP8  JUMP ON NOT LESS OR EQUAL / GREATER
80 01000000  MOD 000 R/M  ADD EA, DATA8  BYTE ADD DATA TO EA
81 01000000  MOD 001 R/M  OR  EA, DATA8  BYTE OR DATA TO EA
82 01000000  MOD 010 R/M  ADC EA, DATA8  BYTE ADD DATA W/ CARRY TO EA
83 01000000  MOD 011 R/M  SBB EA, DATA8  BYTE SUB DATA W/ BORROW FROM EA
84 01000000  MOD 100 R/M  AND EA, DATA8  BYTE AND DATA TO EA
85 01000000  MOD 101 R/M  SUB EA, DATA8  BYTE SUBTRACT DATA FROM EA
86 01000000  MOD 110 R/M  XOR EA, DATA8  BYTE XOR DATA TO EA
87 01000000  MOD 111 R/M  CMP EA, DATA8  BYTE COMPARE DATA WITH (EA)
88 01000000  MOD 000 R/M  ADD EA, DATA16  WORD ADD DATA TO EA
89 01000000  MOD 001 R/M  OR EA, DATA16  WORD OR DATA TO EA
8A 01000000  MOD 010 R/M  ADC EA, DATA16  WORD ADD DATA W/ CARRY TO EA
81 10000001 MOD 011 R/M SBB EA,DATA16 WORD SUB DATA W/BORROW FROM EA
85 10000001 MOD 100 R/M AND EA,DATA16 WORD AND DATA TO EA
81 10000001 MOD 110 R/M SUB EA,DATA16 WORD SUBTRACT DATA FROM EA
81 10000001 MOD 111 R/M XOR EA,DATA16 WORD XOR DATA TO EA
81 10000001 MOD 111 R/M CMP EA,DATA16 WORD COMPARE DATA WITH (EA)
82 10000010 MOD 000 R/M ADD EA,DATA8 BYTE ADD DATA TO EA
82 10000010 MOD 001 R/M (not used)
82 10000010 MOD 010 R/M ADC EA,DATA8 BYTE ADD DATA W/CARRY TO EA
82 10000010 MOD 011 R/M SBB EA,DATA8 BYTE SUB DATA W/BORROW FROM EA
82 10000100 MOD 100 R/M (not used)
82 10000100 MOD 101 R/M SUB EA,DATA8 BYTE SUBTRACT DATA FROM EA
82 10000100 MOD 110 R/M (not used)
82 10000100 MOD 111 R/M CMP EA,DATA8 BYTE COMPARE DATA WITH (EA)
83 10000011 MOD 000 R/M ADD EA,DATA8 WORD ADD DATA TO EA
83 10000011 MOD 001 R/M (not used)
83 10000011 MOD 010 R/M ADC EA,DATA8 WORD ADD DATA W/CARRY TO EA
83 10000011 MOD 011 R/M SBB EA,DATA8 WORD SUB DATA W/BORROW FROM EA
83 10000011 MOD 100 R/M (not used)
83 10000011 MOD 101 R/M SUB EA,DATA8 WORD SUBTRACT DATA FROM EA
83 10000011 MOD 110 R/M (not used)
83 10000011 MOD 111 R/M CMP EA,DATA8 WORD COMPARE DATA WITH (EA)
84 10000001 MOD REG/R/M TEST EA,REG BYTE TEST (EA) W/ (REG)
85 10000101 MOD REG/R/M TEST EA,REG WORD TEST (EA) W/ (REG)
86 10000110 MOD REG/R/M XCHG REG,EA BYTE EXCHANGE (REG) W/ (EA)
87 10000111 MOD REG/R/M XCHG REG,EA WORD EXCHANGE (REG) W/ (EA)
88 10001000 MOD REG/R/M MOV EA,REG BYTE MOVE (REG) TO EA
89 10001001 MOD REG/R/M MOV EA,REG WORD MOVE (REG) TO EA
9A 10001010 MOD REG/R/M MOV REG,EA BYTE MOVE (EA) TO REG
9B 10001011 MOD REG/R/M MOV REG,EA WORD MOVE (EA) TO REG
9C 10001100 MOD 0SR R/M MOV EA,SR WORD MOVE (SEGMENT REG SR) TO EA
9D 10001101 MOD 1 — R/M (not used)
9E 10001110 MOD 0SR R/M MOV SR,EA WORD MOVE (EA) TO SEGMENT REG SR
9E 10001111 MOD — R/M (not used)
8F 10001111 MOD 000 R/M POP EA POP STACK TO EA
8F 10001111 MOD 001 R/M (not used)
8F 10001111 MOD 010 R/M (not used)
8F 10001111 MOD 011 R/M (not used)
8F 10001111 MOD 100 R/M (not used)
8F 10001111 MOD 101 R/M (not used)
8F 10001111 MOD 110 R/M (not used)
8F 10001111 MOD 111 R/M (not used)
90 10010000 XCHG AX,AX EXCHANGE (AX) W/ (AX), (NOP)
91 10010001 XCHG AX,CX EXCHANGE (AX) W/ (CX)
92 10010010 XCHG AX,DX EXCHANGE (AX) W/ (DX)
93 10010100 XCHG AX,BX EXCHANGE (AX) W/ (BX)
94 10010100 XCHG AX,SP EXCHANGE (AX) W/ (SP)
95 10010101 XCHG AX,BP EXCHANGE (AX) W/ (BP)
96 10010110 XCHG AX,SI EXCHANGE (AX) W/ (SI)
97 10010111 XCHG AX,DI EXCHANGE (AX) W/ (DI)
98 10011000 CBW BYTE CONVERT (AL) TO WORD (AX)
99 10011001 CWD WORD CONVERT (AX) TO DOUBLE WORD
9A 10011010 CALL DISP16,SEG16 DIRECT INTER SEGMENT CALL
9B 10011011 WAIT WAIT FOR TEST SIGNAL
9C 10011100 PUSHF PUSH FLAGS ON STACK
9D 10011101 POPF POP STACK TO FLAGS
9E 10011110 SAHF STORE (AH) INTO FLAGS
9F 10011111 LAHF LOAD REG AH WITH FLAGS
A0 10100000 MOV AL,ADDR16 BYTE MOVE (ADDR) TO ADDR
A1 10100001 MOV AX,ADDR16 WORD MOVE (ADDR) TO ADDR AX
A2 10100010 MOV ADDR16,AL BYTE MOVE (AL) TO ADDR
A3 10100111 MOV ADDR16,AX WORD MOVE (AX) TO ADDR
A4 10101000 MOVS DST8SRC8 BYTE MOVE, STRING OP
A5 10101001 MOVS DST16,SRC16 WORD MOVE, STRING OP
A6 10101010 CMPS SIPTR,DIPTR COMPARE BYTE, STRING OP
A7 10101101 CMPS SIPT,R,DIPTR COMPARE WORD, STRING OP
A8 10101100 TEST AL,DATA8 BYTE TEST (AL) W/ DATA
A9 10101101 TEST AX,DATA16 WORD TEST (AX) W/ DATA
AA 10101110 STOS DST8 BYTE STORE, STRING OP
AB 10101111 STOS DST16 WORD STORE, STRING OP
AC 10101110 LODS SRC8 BYTE LOAD, STRING OP
AD 10101111 LODS SRC16 WORD LOAD, STRING OP
AE 10101110 SCAS DIPT8 BYTE SCAN, STRING OP
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AF 10101111</td>
<td>SCAS DIPT16 WORD SCAN STRING OP</td>
</tr>
<tr>
<td>B0 10110000</td>
<td>MOV AL,DATA8 BYTE MOVE DATA TO REG AL</td>
</tr>
<tr>
<td>B1 10110001</td>
<td>MOV CL,DATA8 BYTE MOVE DATA TO REG CL</td>
</tr>
<tr>
<td>B2 10110010</td>
<td>MOV DL,DATA8 BYTE MOVE DATA TO REG DL</td>
</tr>
<tr>
<td>B3 10110011</td>
<td>MOV BL,DATA8 BYTE MOVE DATA TO REG BL</td>
</tr>
<tr>
<td>B4 10110100</td>
<td>MOV AH,DATA8 BYTE MOVE DATA TO REG AH</td>
</tr>
<tr>
<td>B5 10110101</td>
<td>MOV CH,DATA8 BYTE MOVE DATA TO REG CH</td>
</tr>
<tr>
<td>B6 10111010</td>
<td>MOV DH,DATA8 BYTE MOVE DATA TO REG DH</td>
</tr>
<tr>
<td>B7 10111111</td>
<td>MOV BH,DATA8 BYTE MOVE DATA TO REG BH</td>
</tr>
<tr>
<td>B8 10111000</td>
<td>MOV AX,DATA16 WORD MOVE DATA TO REG AX</td>
</tr>
<tr>
<td>B9 10111001</td>
<td>MOV CX,DATA16 WORD MOVE DATA TO REG CX</td>
</tr>
<tr>
<td>BA 10111010</td>
<td>MOV DX,DATA16 WORD MOVE DATA TO REG DX</td>
</tr>
<tr>
<td>BB 10111100</td>
<td>MOV SP,DATA16 WORD MOVE DATA TO REG SP</td>
</tr>
<tr>
<td>BD 10111101</td>
<td>MOV BP,DATA16 WORD MOVE DATA TO REG BP</td>
</tr>
<tr>
<td>BE 10111110</td>
<td>MOV SI,DATA16 WORD MOVE DATA TO REG SI</td>
</tr>
<tr>
<td>BF 10111111</td>
<td>MOV DI,DATA16 WORD MOVE DATA TO REG DI</td>
</tr>
<tr>
<td>C0 11000000</td>
<td>(not used)</td>
</tr>
<tr>
<td>C1 11000001</td>
<td>(not used)</td>
</tr>
<tr>
<td>C2 11000010</td>
<td>RET DATA16 INTRA SEGMENT RETURN, ADD DATA TO REG SP</td>
</tr>
<tr>
<td>C3 11000111</td>
<td>RET DATA16 INTRA SEGMENT RETURN</td>
</tr>
<tr>
<td>C4 11000100</td>
<td>MOD REG/R MOD REG AND SEGMENT REG ES</td>
</tr>
<tr>
<td>C5 11000010</td>
<td>MOD REG/R MOD REG AND SEGMENT REG DS</td>
</tr>
<tr>
<td>C6 11000110</td>
<td>MOV 00 R/M MOV EA,DATA8 BYTE MOVE DATA TO EA</td>
</tr>
<tr>
<td>C8 11000110</td>
<td>(not used)</td>
</tr>
<tr>
<td>C9 11000101</td>
<td>(not used)</td>
</tr>
<tr>
<td>CA 11000110</td>
<td>RET DATA16 INTER SEGMENT RETURN, ADD DATA TO REG SP</td>
</tr>
<tr>
<td>CB 11001011</td>
<td>RET INTER SEGMENT RETURN</td>
</tr>
<tr>
<td>CD 11001010</td>
<td>INT 3 TYPE 3 INTERRUPT</td>
</tr>
<tr>
<td>CD 11001101</td>
<td>INT TYPE TYPED INTERRUPT</td>
</tr>
<tr>
<td>CE 11001110</td>
<td>INTO INTERRUPT ON OVERFLOW</td>
</tr>
<tr>
<td>CF 11001111</td>
<td>IRET RETURN FROM INTERRUPT</td>
</tr>
<tr>
<td>D0 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>D1 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>D2 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>D3 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>D4 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>D5 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>D6 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>D7 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>D8 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>D9 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>DA 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>DB 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>DC 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>DD 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>DE 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>DF 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>E0 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>E1 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>E2 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>E3 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>E4 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>E5 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>E6 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>E7 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>E8 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>E9 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>EA 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>EB 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>EC 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>ED 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>EE 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>EF 11010000</td>
<td>MOD 00 R/M ROL EA,1 BYTE ROTATE EA LEFT 1 BIT</td>
</tr>
<tr>
<td>Instruction</td>
<td>Description</td>
</tr>
<tr>
<td>-------------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>RCL EA,CL</code></td>
<td>WORD ROTATE EA LEFT THRU CARRY (CL) BITS</td>
</tr>
<tr>
<td><code>RCR EA,CL</code></td>
<td>WORD ROTATE EA RIGHT THRU CARRY (CL) BITS</td>
</tr>
<tr>
<td><code>SHL EA,CL</code></td>
<td>WORD SHIFT EA LEFT (CL) BITS</td>
</tr>
<tr>
<td><code>SHR EA,CL</code></td>
<td>WORD SHIFT EA RIGHT (CL) BITS</td>
</tr>
<tr>
<td><code>SAR EA,CL</code></td>
<td>WORD SHIFT SIGNED EA RIGHT (CL) BITS</td>
</tr>
<tr>
<td><code>AAM</code></td>
<td>ASCII ADJUST FOR MULTIPLY</td>
</tr>
<tr>
<td><code>ADD</code></td>
<td>ASCII ADJUST FOR DIVIDE</td>
</tr>
<tr>
<td><code>LOOPNZI LOOPNE DISP8 LOOP</code></td>
<td>08</td>
</tr>
<tr>
<td><code>E71110011</code></td>
<td>OUT AL,DX BYTE OUTPUT (AL) TO PORT</td>
</tr>
<tr>
<td><code>E91110101</code></td>
<td>JMP DISP16 DIRECT INTRA SEGMENT JUMP</td>
</tr>
<tr>
<td><code>EA111010</code></td>
<td>JMP DISP16,SEG16 DIRECT INTER SEGMENT JUMP</td>
</tr>
<tr>
<td><code>EB111010</code></td>
<td>JMP DISP8 DIRECT INTRA SEGMENT JUMP</td>
</tr>
<tr>
<td><code>EC111010</code></td>
<td>IN AL,DX BYTE INPUT FROM PORT (DX) TO REG AX</td>
</tr>
<tr>
<td><code>ED111010</code></td>
<td>IN AX,DX WORD INPUT FROM PORT (DX) TO REG AX</td>
</tr>
<tr>
<td><code>EE111010</code></td>
<td>OUT DX BYTE OUTPUT (AL) TO PORT (DX)</td>
</tr>
<tr>
<td><code>EF111010</code></td>
<td>OUT AX WORD OUTPUT (AX) TO PORT</td>
</tr>
<tr>
<td><code>F01110000</code></td>
<td>LOCK BUS LOCK PREFIX</td>
</tr>
<tr>
<td><code>F1111001</code></td>
<td>NOT (EA)</td>
</tr>
<tr>
<td><code>F21110010</code></td>
<td>REPN REPEAT WHILE (CX)≠0 AND (ZF)=0</td>
</tr>
<tr>
<td><code>F31110011</code></td>
<td>REPN REPEAT WHILE (CX)≠0 AND (ZF)=1</td>
</tr>
<tr>
<td><code>F41110100</code></td>
<td>HALT</td>
</tr>
<tr>
<td><code>F51111010</code></td>
<td>CMC COMPLEMENT CARRY FLAG</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>TEST EA,DATA8 BYTE TEST (EA) WITH DATA</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>NOT EA BYTE INVERT EA</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>MOD 011 R/M</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>NOT EA BYTE INVERT EA</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>NEG EA BYTE NEGATE EA</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>MOD 100 R/M</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>MUL EA BYTE MULTIPLY BY (EA), UNSIGNED</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>MOD 101 R/M</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>IMUL EA BYTE MULTIPLY BY (EA), SIGNED</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>MOD 110 R/M</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>DIV EA BYTE DIVIDE BY (EA), UNSIGNED</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>MOD 111 R/M</td>
</tr>
<tr>
<td><code>F61111010</code></td>
<td>IDIV EA BYTE DIVIDE BY (EA), SIGNED</td>
</tr>
<tr>
<td><code>F71111011</code></td>
<td>TEST EA,DATA16 WORD TEST (EA) WITH DATA</td>
</tr>
<tr>
<td><code>F71111011</code></td>
<td>NOT EA WORD INVERT EA</td>
</tr>
<tr>
<td><code>F71111011</code></td>
<td>MOD 011 R/M</td>
</tr>
<tr>
<td><code>F71111011</code></td>
<td>NEG EA WORD NEGATE EA</td>
</tr>
<tr>
<td><code>F71111011</code></td>
<td>MOD 100 R/M</td>
</tr>
<tr>
<td><code>F71111011</code></td>
<td>MUL EA WORD MULTIPLY BY (EA), UNSIGNED</td>
</tr>
<tr>
<td><code>F71111011</code></td>
<td>MOD 101 R/M</td>
</tr>
<tr>
<td><code>F71111011</code></td>
<td>IMUL EA WORD MULTIPLY BY (EA), SIGNED</td>
</tr>
<tr>
<td><code>F71111011</code></td>
<td>MOD 110 R/M</td>
</tr>
<tr>
<td><code>F71111011</code></td>
<td>DIV EA WORD DIVIDE BY (EA), UNSIGNED</td>
</tr>
<tr>
<td><code>F71111011</code></td>
<td>MOD 111 R/M</td>
</tr>
<tr>
<td><code>F71111011</code></td>
<td>IDIV EA WORD DIVIDE BY (EA), SIGNED</td>
</tr>
<tr>
<td><code>F81111000</code></td>
<td>CLC CLEAR CARRY FLAG</td>
</tr>
<tr>
<td><code>F91111001</code></td>
<td>STC SET CARRY FLAG</td>
</tr>
<tr>
<td><code>FA1111010</code></td>
<td>CLI CLEAR INTERRUPT FLAG</td>
</tr>
<tr>
<td><code>FB1111011</code></td>
<td>STI SET INTERRUPT FLAG</td>
</tr>
<tr>
<td><code>FC1111100</code></td>
<td>CLD CLEAR DIRECTION FLAG</td>
</tr>
<tr>
<td><code>FD1111101</code></td>
<td>STD SET DIRECTION FLAG</td>
</tr>
<tr>
<td><code>FE1111110</code></td>
<td>INC EA BYTE INCREMENT EA</td>
</tr>
<tr>
<td><code>FE1111110</code></td>
<td>MOD 000 R/M</td>
</tr>
<tr>
<td><code>FE1111110</code></td>
<td>DEC EA BYTE DECREMENT EA</td>
</tr>
<tr>
<td><code>FE1111110</code></td>
<td>MOD 001 R/M</td>
</tr>
<tr>
<td><code>FE1111110</code></td>
<td>INC EA WORD INCREMENT EA</td>
</tr>
<tr>
<td><code>FE1111110</code></td>
<td>MOD 001 R/M</td>
</tr>
<tr>
<td><code>FE1111110</code></td>
<td>DEC EA WORD DECREMENT EA</td>
</tr>
<tr>
<td><code>FE1111110</code></td>
<td>MOD 100 R/M</td>
</tr>
<tr>
<td><code>FE1111110</code></td>
<td>CALL EA INDIRECT INTRA SEGMENT CALL</td>
</tr>
<tr>
<td><code>FF111111</code></td>
<td>MOD 011 R/M</td>
</tr>
<tr>
<td><code>FF111111</code></td>
<td>CALL EA INDIRECT INTER SEGMENT CALL</td>
</tr>
<tr>
<td><code>FF111111</code></td>
<td>MOD 100 R/M</td>
</tr>
<tr>
<td><code>FF111111</code></td>
<td>JMP EA INDIRECT INTRA SEGMENT JUMP</td>
</tr>
<tr>
<td><code>FF111111</code></td>
<td>MOD 101 R/M</td>
</tr>
<tr>
<td><code>FF111111</code></td>
<td>JMP EA INDIRECT INTER SEGMENT JUMP</td>
</tr>
<tr>
<td><code>FF111111</code></td>
<td>MOD 110 R/M</td>
</tr>
<tr>
<td><code>FF111111</code></td>
<td>PUSH EA PUSH (EA) ON STACK</td>
</tr>
<tr>
<td><code>FF111111</code></td>
<td>MOD 111 R/M</td>
</tr>
</tbody>
</table>
REG IS ASSIGNED ACCORDING TO THE FOLLOWING TABLE:

<table>
<thead>
<tr>
<th>16-BIT (W=1)</th>
<th>8-BIT (W=0)</th>
<th>SEGMENT REG</th>
</tr>
</thead>
<tbody>
<tr>
<td>000 AX</td>
<td>000 AL</td>
<td>00 ES</td>
</tr>
<tr>
<td>001 CX</td>
<td>001 CL</td>
<td>01 CS</td>
</tr>
<tr>
<td>010 DX</td>
<td>010 DL</td>
<td>10 SS</td>
</tr>
<tr>
<td>011 BX</td>
<td>011 BL</td>
<td>11 DS</td>
</tr>
<tr>
<td>100 SP</td>
<td>100 AH</td>
<td></td>
</tr>
<tr>
<td>101 BP</td>
<td>101 CH</td>
<td></td>
</tr>
<tr>
<td>110 SI</td>
<td>110 DH</td>
<td></td>
</tr>
<tr>
<td>111 DI</td>
<td>111 BH</td>
<td></td>
</tr>
</tbody>
</table>

EA IS COMPUTED AS FOLLOWS: (DISP8 SIGN EXTENDED TO 16 BITS)

<p>| | | | | | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>000</td>
<td>(BX) + (SI)</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>001</td>
<td>(BX) + (DI)</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>010</td>
<td>(BP) + (SI)</td>
<td>SS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>011</td>
<td>(BP) + (DI)</td>
<td>SS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>100</td>
<td>(SI)</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>101</td>
<td>(DI)</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>110</td>
<td>DISP16 (DIRECT ADDRESS)</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00</td>
<td>111</td>
<td>(BX)</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>000</td>
<td>(BX) + (SI) + DISP8</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>001</td>
<td>(BX) + (DI) + DISP8</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>010</td>
<td>(BP) + (SI) + DISP8</td>
<td>SS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>011</td>
<td>(BP) + (DI) + DISP8</td>
<td>SS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>100</td>
<td>(SI) + DISP8</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>101</td>
<td>(DI) + DISP8</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>110</td>
<td>(BP) + DISP8</td>
<td>SS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>01</td>
<td>111</td>
<td>(BX) + DISP8</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>000</td>
<td>(BX) + (SI) + DISP16</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>001</td>
<td>(BX) + (DI) + DISP16</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>010</td>
<td>(BP) + (SI) + DISP16</td>
<td>SS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>011</td>
<td>(BP) + (DI) + DISP16</td>
<td>SS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>100</td>
<td>(SI) + DISP16</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>101</td>
<td>(DI) + DISP16</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>110</td>
<td>(BP) + DISP16</td>
<td>SS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>111</td>
<td>(BX) + DISP16</td>
<td>DS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>000</td>
<td>REG AX / AL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>001</td>
<td>REG CX / CL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>010</td>
<td>REG DX / DL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>011</td>
<td>REG BX / BL</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>100</td>
<td>REG SP / AH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>101</td>
<td>REG BP / CH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>110</td>
<td>REG SI / DH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>111</td>
<td>REG DI / BH</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

FLAGS REGISTER CONTAINS:

### 8086 Assembly Language

#### SET MATRIX

<table>
<thead>
<tr>
<th>Hi</th>
<th>Lo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>OR b.t/r/m OR w.t/r/m OR b.t/m OR w.t/m OR b.i OR w.i</td>
</tr>
<tr>
<td>1</td>
<td>SBB b.t/r/m SBB b.t/m SBB w.t/r/m SBB w.t/m SBB b.i SBB w.i</td>
</tr>
<tr>
<td>2</td>
<td>SUB b.t/r/m SUB b.t/m SUB w.t/r/m SUB w.t/m SUB b.i SUB w.i</td>
</tr>
<tr>
<td>3</td>
<td>CMP b.t/r/m CMP b.t/m CMP w.t/r/m CMP w.t/m CMP b.i CMP w.i</td>
</tr>
<tr>
<td>4</td>
<td>DEC AX DEC CX DEC DX DEC SP DEC BP DEC DI DEC DI</td>
</tr>
<tr>
<td>5</td>
<td>POP AX POP CX POP DX POP BX POP SP POP BP POP SI POP DI</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>JS JNS JP/ JPD JLP/ JLPD</td>
</tr>
<tr>
<td>8</td>
<td>MOV b.t/r/m MOV b.t/m MOV b.f/r/m MOV b.f/m MOV b.t/r/m</td>
</tr>
<tr>
<td>9</td>
<td>CBW CWX CALL d WAIT</td>
</tr>
<tr>
<td>A</td>
<td>TEST b.LA TEST w.LA STOS STOS LODS LODS SCAS SCAS</td>
</tr>
<tr>
<td>B</td>
<td>MOV i - AX MOV i - CX MOV i - DX MOV i - BX MOV i - SP MOV i - BP MOV i - SI MOV i - DI</td>
</tr>
<tr>
<td>C</td>
<td>RET i(SP) RET i INT Type 3 INT (Any) INTO IRET</td>
</tr>
<tr>
<td>D</td>
<td>ESC 0 ESC 3 ESC 4 ESC 3 ESC 4 ESC 5 ESC 7</td>
</tr>
<tr>
<td>E</td>
<td>CALL d JMP d</td>
</tr>
<tr>
<td>F</td>
<td>CLC STC CLI STI CLD STD Grp 2 b.t/r/m Grp 2 w.t/r/m</td>
</tr>
</tbody>
</table>

#### Instructions in Hexadecimal Order

<table>
<thead>
<tr>
<th>Hi</th>
<th>Lo</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>ADD b.t/r/m ADD b.t/r/m ADD w.t/r/m ADD b.i ADD w.i</td>
</tr>
<tr>
<td>1</td>
<td>ADC b.t/r/m ADC b.t/r/m ADC w.t/r/m ADC b.i ADC w.i</td>
</tr>
<tr>
<td>2</td>
<td>AND b.t/r/m AND b.t/r/m AND w.t/r/m AND b.i AND w.i</td>
</tr>
<tr>
<td>3</td>
<td>XOR b.t/r/m XOR b.t/r/m XOR b.t/r/m XOR b.i XOR w.i</td>
</tr>
<tr>
<td>4</td>
<td>INC AX INC CX INC DX INC BP INC SP INC SI INC DI</td>
</tr>
<tr>
<td>5</td>
<td>PUSH AX PUSH CX PUSH DX PUSH BX PUSH SP PUSH SI PUSH DI</td>
</tr>
<tr>
<td>6</td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>JO JNO JB/ JNAE JB/ JAE JE/ JZ JE/ JNZ JE/ JNA</td>
</tr>
<tr>
<td>9</td>
<td>XCHG AX XCHG CX XCHG DX XCHG SP XCHG BX XCHG SI XCHG DI</td>
</tr>
<tr>
<td>A</td>
<td>MOV m - AL - AX AX m MOV m - AL - AX AX m MOV m - AL - AX AX m</td>
</tr>
<tr>
<td>B</td>
<td>MOV i - AL MOV i - CL MOV i - DL MOV i - BL MOV i - AH MOV i - CH MOV i - DH MOV i - BH</td>
</tr>
<tr>
<td>C</td>
<td>RET i(SP) RET i LES LOS MOV b.t/r/m MOV b.t/m</td>
</tr>
<tr>
<td>D</td>
<td>Shift b Shift w Shift b.v Shift b.v</td>
</tr>
<tr>
<td>E</td>
<td>LOOPNZ LOOPZ LOOP LOOP LOOP LOOPJ LOOPJ LOOPJ LOOPJ</td>
</tr>
<tr>
<td>F</td>
<td>LOCK REP REP R REP R REP R REP R REP R</td>
</tr>
</tbody>
</table>

---

Where:
- **mod** = r/m
- **imm** = ADD
- **Shift** = ADD
- **Grp** = ADD

**a** = byte operation
**d** = direct
**f** = from CPU reg
**i** = immediate
**s** = segment register
**l** = to CPU reg
**v** = variable
**w** = word operation

---

**E-7**
INDEX

ABSOLUTE, 6-18
ACKNOWLEDGE, 6-7
Actual Parameters, 8-10
ADDR, 6-25, 6-27
Address match range, 6-6
AFL, 4-10, 7-16
ALL, 6-24
AND, 5-11, 5-13, 5-14
Arithmetic operators, 5-9
ASCII, 4-14
ASCII codes, 5-8, 7-59
ASM, 4-15, 7-71
Base, 7-11
BASE, 4, 14, 7-11, 7-13
Base pointer, 4-9
Binary operator, 5-9, 5-10, 5-11
BOOL, 7-67, 7-74
BP, 4-9, 7-15
Breakpoint registers, 4-12, 6-4
BR, 4-12, 6-4
BR0, 4-12, 6-4
BR1, 4-12, 6-4
Buffer pointer, 6-23
BUFFERSIZE, 7-15
BUFFERSIZE register, 4-10
BYTE, 5-5, 5-6, 5-12, 7-30, 7-58
CARS module listing, 3-3
CAUSE, 4-10, 7-15
CAUSE register, 4-10
CFL, 4-10, 7-16
Change Symbol command, 7-43, 7-49
Character set, 4-6
CLOCK, 4-15, 6-19
Code segment register, 4-9
Command contexts, 5-22
Command keywords, 4-12
Command signal timeout, 6-12
COMMENTS, 6-26
Compound commands, 8-1
Console input radices (SUFFIX), 7-10
Console output radices (BASE), 7-11
Content operators, 5-12
CS, 4-9, 7-15
COUNT, 8-3
COUNT command, 8-3, 8-6
Data match range, 6-8
Data segment register, 4-9
DEFINE MACRO command, 8-8, 8-12
DEFINE Symbol command, 7-40, 7-45
DELAY module listing, 3-4
Destination index, 4-9
Device, 7-2, 7-9
DFL, 4-10, 7-16
DI, 4-9, 7-15
Digit, 5-2
DIR, 8-12
DISK, 4-14, 7-22, 7-23
DISABLE, 4-15
Display, 1-4
Display Boolean command, 7-67, 7-74
Display CLOCK command, 6-19
Display commands, 4-13, 7-54
Display Emulation Register command, 6-18
Display I/O command, 7-64, 7-72
Display Macro command, 8-12
Display MAP Status command, 7-22, 7-28
Display Memory command, 7-60, 7-71
Display Modules command, 7-43, 7-48
Display Processor and Status Register command, 7-59, 7-70
Display RWTIMEOUT command, 6-20
Display STACK command, 7-67, 7-73
Display Statement Numbers command, 7-42, 7-47
Display Symbols command, 7-41, 7-46
Display TRACE command, 6-33
DMUX, 6-25
DOWN, 6-8, 6-9
Drive, 7-2, 7-7, 7-9
DS, 4-9, 7-15
ELSE, 8-5
EM, 8-8
Emulation control keywords, 4-16
Emulation control commands, 6-1
Emulation timer, 6-12
ENABLE, 4-15
ENABLE/DISABLE RDY command, 6-21
ENABLE/DISABLE TRACE command, 6-23, 6-32
END, 8-1, 8-3
ES, 4-9, 7-15
EVALUATE, 4-14
EVALUATE command, 7-68, 7-76
EXECUTED, 6-4
Execution commands, 4-13
Execution match condition, 6-4
EXIT, 7-6
EXIT command, 7-3, 7-6
Explicit radix, 5-3, 7-10
Expressions, 5-1
Extra segment register, 4-9
EXTERNAL, 4-15
FETCHED, 6-7
Filename, 7-2, 7-7, 7-9
FLAG, 4-15
Flag references, 7-16
Flag register, 4-10
FOREVER, 6-2, 6-3, 6-10
Formal parameters, 8-10
FRAME, 6-25, 6-26, 6-27
Frames mode, 6-25
FROM, 6-3, 6-4, 6-10
General register file, 7-55
General registers, 7-14
Index-1
Generalized Development Cycle with ICE-88, 1-10
Generalized Emulation Session, 1-12
GO, 6-3, 6-4
GO command, 6-3, 6-15
GR, 4-12, 6-2, 6-10
GR register, 4-12, 6-2, 6-10
GUARDED, 7-22
HALT, 6-7
HALT GO condition, 6-10
HARDWARE, 4-15
Hardware installation procedures, 2-2
Hardware register command keywords, 4-15
HTIMER, 4-10, 6-12, 7-15
HTIMER register, 4-10, 6-12, 6-22
HOLD, 4-12
ICE, 7-21
ICE-88 architecture, 1-8
ICE-88 components, 2-1
ICE-88 firmware, 1-9
ICE-88 hardware, 1-9
ICE-88 In-circuit emulation, 1-2
ICE-88 In-circuit Emulator, 1-1
ICE-88 software, 1-9
ICE88 command, 7-3, 7-5
IF, 8-5
IF command, 8-5, 8-6
IFL, 4-10, 7-16
Implicit radix, 7-10
INCLUDE, 8-16
INCLUDE command, 8-16
Index registers, 7-15
INPUT, 6-7
Installation Procedure for Intellec Series II Model 220 and 230, 2-3
Installation Procedure for Intellec Model 800 and 888, 2-3
INSTRUCTION, 6-23
Instruction mode, 6-26
Instruction pointer, 4-10
Interrogation and Utility commands, 1-6
Integer, 5-1
INTEGER, 5-5, 5-6, 5-12, 7-30, 7-58
INTELLEC, 7-20, 7-21, 7-22
INTERNAL, 4-15
INTR, 4-12
Invoke Macro command, 8-8
IP, 4-10, 7-15
IR, 4-12, 7-16
Keyword references, 5-3
Keywords, 4-7
LENGTH, 6-6, 6-7
LINE, 7-47
LIST, 7-4
LIST command, 7-4, 7-9
LOAD, 7-3
LOAD command, 7-3, 7-7
Local and global defaults, 8-9
Logical operators, 5-12
LOWER, 4-10, 7-15
MAC TMP, 8-8
MACRO, 8-8
Macro commands, 8-8
Macro Directory command, 8-12
Macro expansion, 8-12
Macro table commands, 8-12
MAP DISK command, 7-21, 7-24
MAP INTELLEC command, 7-21, 7-25
MARK, 6-25
MASK, 5-10, 5-14
Masked constant, 5-3
Match condition, 6-4
Match condition restrictions, 6-8
Match status list, 6-7
Memory and port contents, 4-15
Memory mapping, 4-14
Memory mapping commands, 7-20
Memory references, 5-5
MN, 7-16
MN/MX, 4-12
MNEMONIC, 6-26
MOD, 5-10, 5-14
Module name, 5-4, 7-39
MOVE, 6-23, 6-33
MOVE command, 6-33
Nesting compound command, 8-6
NEWEST, 6-23
NEWEST command, 6-33
NMI, 7-16
NOCODE, 7-3
NOLINE, 7-3
Non-execution match condition, 6-6, 6-9
NOSYMBOL, 7-3
NOT, 5-11, 5-12, 5-14
Notation and conventions used in this manual, 4-3
Notational symbols, 4-3
NOVERIFY, 7-22
Number base, 5-3
Number base and radix commands, 4-14, 7-10
Numeric constant, 5-2
Numeric value display formats, 7-58
O, 4-14
OBJECT, 6-7
OF, 7-40
OFF, 6-23, 6-32
OFFSET, 5-10, 5-14
OFFTRACE, 6-11, 6-12
OFL, 4-11, 7-16
OLDEST, 6-23
OLDEST command, 6-33
ON, 6-23, 6-32
ONTRACE, 6-11, 6-12
OPCODE, 4-10, 7-15
OPCODE register, 4-10
OPERANDS, 6-26
Operands, 5-2
Operating modes, 1-5
Operators, 4-18, 5-9
OR, 5-11, 5-13, 5-14
ORIF, 8-5
OUTPUT, 6-7
Partition, 6-6, 7-21, 7-31
Parenthesized expressions, 5-9
Pathname, 7-2
PFL, 4-11, 7-16
PIN, 4-15
Pin references 4-11, 7-16
PIP, 4-10, 7-15
POINTER, 5-5, 5-12, 7-30, 7-58
Pointer, 5-1
Prefix command, 4-7
PORT, 5-7, 5-12
Port content references, 7-35
Port reference, 5-12
Prefix command, 4-7
Previous instruction register, 4-10, 7-15
PRINT, 6-24
PRINT command, 6-34
Punctuation, 4-19
PUT MACRO command, 8-13
Q, 4-14
QDEPTH, 6-25
QSTS, 6-25
RAH, 7-14
RAL, 7-14
RAX, 7-14
RBH, 7-14
RBL, 7-14
RBX, 7-14
RCH, 7-14
RCL, 7-14
RCX, 7-14
RDH, 7-14
RDL, 7-14
RDX, 7-14
RDX, 7-14
RDY, 7-16
READ, 6-7
READY, 4-11
Reference keywords, 4-7
REGISTER, 4-15
Registers, 4-8
Relational Operators, 5-12
REMOVE MACRO command, 8-12
REMOVE MODULE command, 7-43, 7-50
REMOVE Symbols command, 7-43, 7-50
REPEAT command, 8-1, 8-6
Required and optional hardware, 2-1
RESET, 4-12
RESET Domain command, 7-44, 7-57
RESET HARDWARE command, 7-19
RESET MAP command, 7-23, 7-29
RF, 4-10, 7-15
RST, 7-16
RWTIMEOUT, 4-15, 6-12
SAVE command, 7-3, 7-8
Saving macros, 8-13
SEGMENT, 5-10, 5-14
Segment register file, 4-9
Segment register usage, 6-8
Segment registers, 7-15
Semantic rules, 5-14
Set Breakpoint Register command, 6-13
Set/change commands, 4-12
Set CLOCK command, 6-19
Set Domain command, 7-44, 7-52
Set GO-Register (GR) command, 6-16
Set Input/Output Port Contents command, 7-35, 7-38
Set Map Status command, 7-21, 7-26
Set Memory command, 7-37
Set memory and Port Contents commands, 7-30
Set or Display Console Input Radix commands, 7-12
Set or Display Console Output Radix commands, 7-13
Set Register command, 7-18
Set RWTIMEOUT command, 6-20
Set TRACE Display Mode command, 6-31
Set Tracepoint Register command, 6-14
Setting the GO Register, 6-10
Setting memory contents, 7-32
Setting tracepoint registers, 6-11
SFL, 4-11, 7-16
SS, 4-9, 7-15
Simple commands, 4-12
SINTEGER, 5-5, 5-6, 5-12, 7-30, 7-58
Source index, 4-9
SP, 4-9, 7-15
Special tokens, 4-18
SS, 4-9, 7-15
STACK, 7-73
Stack pointer, 4-9
Stack segment register, 4-9
Statement number reference, 5-5
Statement number table, 4-16
Statement numbers, 4-18
Statement references, 7-41
Status bits, 4-10
Status registers, 4-10, 7-15
STEP command, 6-17
String constants, 5-7
STTS, 6-25
Suffix, 7-10
SUFFIX, 4-14, 7-10, 7-12, 8-9
SUFFIX command, 7-10
Symbolic debugging, 1-4
Symbolic references, 5-4, 7-40, 7-41
Symbols, 4-17, 7-39
Symbol table, 4-16
Symbol Table and Statement Number Table commands, 7-39
T, 4-14
TEST, 7-16
TFL, 4-11, 7-16
THEN, 8-5
TILL, 6-4, 6-10
TIMER, 4-10, 6-12, 6-22, 7-15
TIMER register, 4-10
TO, 6-6, 7-31
Tokens, 4-7
Trace buffer, 6-22
Trace control, 4-17
Trace control commands, 6-22
Trace display formats, 6-25
Trace display mode, 6-23
Tracepoint registers, 6-11
TYPE command, 7-44, 7-51
Typed memory reference, 5-6

Unary operators, 5-9, 5-11, 5-14
UNTIL, 8-1, 8-2, 8-3
UP, 6-7, 6-8
UPPER, 4-10, 7-15
USER, 7-21, 7-22
User names, 4-17
USING CS, 6-8
USING DS, 6-8
USING ES, 6-8
USING SS, 6-8
Utility commands, 4-14, 7-2

WHILE, 8-1, 8-2, 8-3
WORD, 5-5, 5-6, 5-12, 7-30, 7-58
WPORT, 5-7, 5-12, 7-35
WRITE, 8-17
WRITE command, 8-17
WRITTEN, 6-7

XOR, 5-11, 5-13, 5-14
Y, 4-14

ZFL, 4-11, 7-16

8086 Pin references, 4-11
REQUEST FOR READER’S COMMENTS

The Microcomputer Division Technical Publications Department attempts to provide documents that meet the needs of all Intel product users. This form lets you participate directly in the documentation process. Please restrict your comments to the usability, accuracy, readability, organization, and completeness of this document.

1. Please specify by page any errors you found in this manual.

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

2. Does the document cover the information you expected or required? Please make suggestions for improvement.

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

3. Is this the right type of document for your needs? Is it at the right level? What other types of documents are needed?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

4. Did you have any difficulty understanding descriptions or wording? Where?

________________________________________________________________________
________________________________________________________________________
________________________________________________________________________
________________________________________________________________________

5. Please rate this document on a scale of 1 to 10 with 10 being the best rating. _____________

NAME _______________________________ DATE __________________________
TITLE ________________________________________________________________
COMPANY NAME/DEPARTMENT ____________________________________________
ADDRESS ______________________________________________________________
CITY ___________________ STATE ___________ ZIP CODE _________________

Please check here if you require a written reply. ☐
WE'D LIKE YOUR COMMENTS ...

This document is one of a series describing Intel products. Your comments on the back of this form will help us produce better manuals. Each reply will be carefully reviewed by the responsible person. All comments and suggestions become the property of Intel Corporation.