MULTI-ICE™
OPERATING INSTRUCTIONS
FOR ISIS-II USERS

Manual Order Number: 9800672-02 Rev. B
This manual describes Multi-ICE, a set of enhancements to Intel's In-Circuit Emulator (ICE) products.

The manual assumes that you are familiar with at least one standard ICE such as ICE-85 or ICE-49, and with the Intellec system facilities including ISIS-II. Information on these topics can be found in the following Intel documents:

<table>
<thead>
<tr>
<th>Manual Title</th>
<th>Order Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICE-85 Operating Instructions for ISIS-II Users</td>
<td>9800463</td>
</tr>
<tr>
<td>ICE-49 Operating Instructions for ISIS-II Users</td>
<td>9800632</td>
</tr>
<tr>
<td>Intellec Series II Hardware Reference Manual</td>
<td>9800462</td>
</tr>
<tr>
<td>Intellec Series II Interface Manual</td>
<td>9800555</td>
</tr>
<tr>
<td>Intellec Series II Installation and Service Manual</td>
<td>9800559</td>
</tr>
<tr>
<td>ISIS-II User's Guide</td>
<td>9800306</td>
</tr>
</tbody>
</table>

Here is a brief summary of the chapter contents of the manual:

Chapter 1 introduces the Multi-ICE product and describes the hardware required to operate a Multi-ICE system. Additionally, the chapter suggests ways to use the manual and explains the notation system used to present command syntax.

Chapter 2 introduces the user to the Multi-ICE system with a brief example of a debugging task involving two ICE-85's.

Chapter 3 describes the construction of arithmetic and boolean expressions to be used in Multi-ICE commands.

Chapter 4 presents a set of commands that can be executed by any ICE in a Multi-ICE system; the commands include loops, conditional blocks, macros, and display commands.

Chapter 5 presents a set of commands that can be executed by any ICE-85 in a Multi-ICE system.

Chapter 6 describes the three basic Multi-ICE commands, and offers a simple model of Multi-ICE operation to help explain how these commands work.

Chapters 7 and 8 present additional Multi-ICE synchronization commands.

Chapter 9 contains a discussion of the overall operation of a Multi-ICE system, and systematically describes the interrelationships among the system components and commands.

Appendix A is a summary of command syntax and keywords particular to the Multi-ICE product.

Appendix B contains instructions for installing a Multi-ICE system in an Intellec Series II system, Model 220 or 230.

Appendix C is a summary of error messages particular to Multi-ICE.

Appendix D contains suggestions for operating Multi-ICE so as to avoid inadvertent errors. The appendix also discusses both the limitations of Multi-ICE as compared to standard-ICE products and ways to use Multi-ICE capabilities to compensate for these limitations.
<table>
<thead>
<tr>
<th>CHAPTER 1</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTRODUCTION</td>
<td></td>
</tr>
<tr>
<td>The Multi-ICE Product</td>
<td>1-1</td>
</tr>
<tr>
<td>Multi-ICE Program Names</td>
<td>1-1</td>
</tr>
<tr>
<td>About This Manual</td>
<td>1-2</td>
</tr>
<tr>
<td>Notation Used In This Manual</td>
<td>1-4</td>
</tr>
<tr>
<td>Basic Definitions</td>
<td>1-4</td>
</tr>
<tr>
<td>Notational Forms and Examples</td>
<td>1-5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 2</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>GETTING STARTED WITH MULTI-ICE</td>
<td></td>
</tr>
<tr>
<td>Sample Programs</td>
<td>2-1</td>
</tr>
<tr>
<td>Installation and Configuration Summary</td>
<td>2-1</td>
</tr>
<tr>
<td>Sample Multi-ICE Session</td>
<td>2-5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 3</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXPRESSIONS</td>
<td></td>
</tr>
<tr>
<td>Operands</td>
<td>3-1</td>
</tr>
<tr>
<td>Numbers</td>
<td>3-1</td>
</tr>
<tr>
<td>Primaries</td>
<td>3-1</td>
</tr>
<tr>
<td>Numeric Constants</td>
<td>3-2</td>
</tr>
<tr>
<td>String Constants</td>
<td>3-2</td>
</tr>
<tr>
<td>Direct References</td>
<td>3-3</td>
</tr>
<tr>
<td>Operators</td>
<td>3-4</td>
</tr>
<tr>
<td>Classes of Operators</td>
<td>3-4</td>
</tr>
<tr>
<td>Arithmetic Operators</td>
<td>3-4</td>
</tr>
<tr>
<td>Content Operators</td>
<td>3-6</td>
</tr>
<tr>
<td>Relational Operators</td>
<td>3-7</td>
</tr>
<tr>
<td>Logical Operators</td>
<td>3-7</td>
</tr>
<tr>
<td>Environment Controls</td>
<td>3-8</td>
</tr>
<tr>
<td>How Expressions Are Evaluated</td>
<td>3-9</td>
</tr>
<tr>
<td>&quot;Case Studies&quot; in Evaluating Expressions</td>
<td>3-9</td>
</tr>
<tr>
<td>Command Contexts</td>
<td>3-10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 4</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICE-INDEPENDENT COMMANDS</td>
<td></td>
</tr>
<tr>
<td>Compound Commands</td>
<td>4-1</td>
</tr>
<tr>
<td>Local and Global Defaults</td>
<td>4-1</td>
</tr>
<tr>
<td>REPEAT Command</td>
<td>4-2</td>
</tr>
<tr>
<td>Discussion</td>
<td>4-3</td>
</tr>
<tr>
<td>COUNT Command</td>
<td>4-5</td>
</tr>
<tr>
<td>Discussion</td>
<td>4-6</td>
</tr>
<tr>
<td>IF Command</td>
<td>4-8</td>
</tr>
<tr>
<td>Discussion</td>
<td>4-9</td>
</tr>
<tr>
<td>Nesting Compound Commands</td>
<td>4-10</td>
</tr>
<tr>
<td>Macro Commands</td>
<td>4-11</td>
</tr>
<tr>
<td>DEFINE MACRO Command</td>
<td>4-12</td>
</tr>
<tr>
<td>Macro Call Command</td>
<td>4-12</td>
</tr>
<tr>
<td>Display Macro Definition Command</td>
<td>4-13</td>
</tr>
<tr>
<td>Display Macro Directly Command</td>
<td>4-13</td>
</tr>
<tr>
<td>REMOVE MACRO Command</td>
<td>4-13</td>
</tr>
<tr>
<td>Put Macro On File Command</td>
<td>4-14</td>
</tr>
<tr>
<td>INCLUDE From File Command</td>
<td>4-14</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 5</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICE-85 DEPENDENT COMMANDS</td>
<td></td>
</tr>
<tr>
<td>SEARCH Command</td>
<td>5-1</td>
</tr>
<tr>
<td>DOMAIN Commands</td>
<td>5-3</td>
</tr>
<tr>
<td>NESTING Command</td>
<td>5-3</td>
</tr>
<tr>
<td>LINES Command</td>
<td>5-3</td>
</tr>
<tr>
<td>MODULES Command</td>
<td>5-4</td>
</tr>
<tr>
<td>FLAG Keyword</td>
<td>5-4</td>
</tr>
<tr>
<td>REMOVE MODULE Command</td>
<td>5-4</td>
</tr>
<tr>
<td>LIMIT and LOWER</td>
<td>5-4</td>
</tr>
<tr>
<td>Sample Mapping Sequences</td>
<td>5-6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 6</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>BASIC MULTI-ICE COMMANDS</td>
<td></td>
</tr>
<tr>
<td>SWITCH Commands</td>
<td>6-1</td>
</tr>
<tr>
<td>ACTIVATE Command</td>
<td>6-1</td>
</tr>
<tr>
<td>KILL Commands</td>
<td>6-2</td>
</tr>
<tr>
<td>Discussion</td>
<td>6-3</td>
</tr>
<tr>
<td>Details on SWITCH Commands</td>
<td>6-4</td>
</tr>
<tr>
<td>Details on ACTIVATE</td>
<td>6-5</td>
</tr>
<tr>
<td>Details on KILL</td>
<td>6-5</td>
</tr>
<tr>
<td>Using the ESC Key to Kill a Process</td>
<td>6-5</td>
</tr>
<tr>
<td>Summary of Multi-ICE Messages</td>
<td>6-6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER 7</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTI-ICE SYNCHRONIZATION COMMANDS</td>
<td></td>
</tr>
<tr>
<td>SUSPEND Commands</td>
<td>7-1</td>
</tr>
<tr>
<td>CONTINUE Commands</td>
<td>7-2</td>
</tr>
<tr>
<td>WAIT Command</td>
<td>7-2</td>
</tr>
<tr>
<td>SUSPEND Commands and CONTINUE Command</td>
<td>7-3</td>
</tr>
<tr>
<td>WAIT Command</td>
<td>7-3</td>
</tr>
</tbody>
</table>
## CHAPTER 8
**BREAK AND LOCK COMMANDS**

<table>
<thead>
<tr>
<th>Command</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>BREAK Command</td>
<td>8-1</td>
</tr>
<tr>
<td>LOCK Command</td>
<td>8-1</td>
</tr>
<tr>
<td>Discussion</td>
<td>8-2</td>
</tr>
</tbody>
</table>

## CHAPTER 9
**MULTI-ICE THEORY OF OPERATION**

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components of a Dual-ICE System</td>
<td>9-1</td>
</tr>
<tr>
<td>Processes and Process Status</td>
<td>9-2</td>
</tr>
<tr>
<td>HOST and ICE Processes</td>
<td>9-2</td>
</tr>
<tr>
<td>Process Status</td>
<td>9-3</td>
</tr>
<tr>
<td>Querying Process Status</td>
<td>9-3</td>
</tr>
<tr>
<td>HOST parsing and Execution Environment</td>
<td>9-3</td>
</tr>
<tr>
<td>Tasks and Task Status</td>
<td>9-4</td>
</tr>
<tr>
<td>The HOST parser</td>
<td>9-5</td>
</tr>
<tr>
<td>Obtaining a Prompt</td>
<td>9-5</td>
</tr>
<tr>
<td>Entering Commands</td>
<td>9-5</td>
</tr>
<tr>
<td>Intermediate and Final Carriage Return</td>
<td>9-6</td>
</tr>
<tr>
<td>Parser Task Status</td>
<td>9-6</td>
</tr>
<tr>
<td>HOST Execution Process</td>
<td>9-8</td>
</tr>
<tr>
<td>Commands to the HOST</td>
<td>9-8</td>
</tr>
<tr>
<td>HOST Process Status and Task Status</td>
<td>9-9</td>
</tr>
<tr>
<td>The ICE Process</td>
<td>9-11</td>
</tr>
<tr>
<td>Commands to the ICE Processes</td>
<td>9-11</td>
</tr>
</tbody>
</table>

## APPENDIX A
**SUMMARY OF MULTI-ICE COMMANDS AND KEYWORDS**

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Expressions</td>
<td>A-1</td>
</tr>
<tr>
<td>Commands</td>
<td>A-1</td>
</tr>
<tr>
<td>Keywords</td>
<td>A-4</td>
</tr>
</tbody>
</table>

## APPENDIX B
**INSTALLATION PROCEDURES FOR INTELLEC SERIES II SYSTEMS**

## APPENDIX C
**MULTI-ICE ERROR MESSAGES**

## APPENDIX D
**OPERATING HINTS AND LIMITATIONS**
TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-1</td>
<td>Multi-ICE Commands</td>
<td>1-3</td>
</tr>
<tr>
<td>2-1</td>
<td>Basic Multi-ICE Keywords</td>
<td>2-5</td>
</tr>
<tr>
<td>3-1</td>
<td>Elements of Numeric Constants</td>
<td>3-2</td>
</tr>
<tr>
<td>3-2</td>
<td>ASCII Printing Characters and</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Codes (20H-7EH)</td>
<td>3-3</td>
</tr>
<tr>
<td>3-3</td>
<td>Enhanced-ICE Operators</td>
<td>3-5</td>
</tr>
<tr>
<td>3-4</td>
<td>Classes of Operators</td>
<td>3-6</td>
</tr>
<tr>
<td>3-5</td>
<td>Content Operators</td>
<td>3-7</td>
</tr>
<tr>
<td>3-6</td>
<td>Representative Cases of Expressions</td>
<td>3-11</td>
</tr>
<tr>
<td>3-7</td>
<td>Conditions and Notations for Examples</td>
<td>3-11</td>
</tr>
<tr>
<td>3-8</td>
<td>Command Contexts</td>
<td>3-17</td>
</tr>
<tr>
<td>4-1</td>
<td>Tracking a COUNT Command</td>
<td>4-7</td>
</tr>
<tr>
<td>4-2</td>
<td>ICE-4X Stack Pointer Locations</td>
<td>4-23</td>
</tr>
<tr>
<td>5-1</td>
<td>ICE-85 Memory Control Keywords</td>
<td>5-4</td>
</tr>
</tbody>
</table>

TABLES I

<table>
<thead>
<tr>
<th>TABLE</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5-2</td>
<td>ICE-85 Memory Blocks for Mapping</td>
<td>5-5</td>
</tr>
<tr>
<td>9-1</td>
<td>Process and Environment Keywords</td>
<td>9-2</td>
</tr>
<tr>
<td>9-2</td>
<td>Process Status Keywords</td>
<td>9-3</td>
</tr>
<tr>
<td>9-3</td>
<td>HOST Parser Task Status</td>
<td>9-6</td>
</tr>
<tr>
<td>9-4</td>
<td>HOST Execution Process Status and Task Status</td>
<td>9-9</td>
</tr>
<tr>
<td>9-5</td>
<td>ICE Process Status and Task Status</td>
<td>9-11</td>
</tr>
<tr>
<td>9-6</td>
<td>How an ICE Process Becomes DORMANT</td>
<td>9-13</td>
</tr>
<tr>
<td>9-7</td>
<td>Dispatch Table</td>
<td>9-16</td>
</tr>
<tr>
<td>9-8</td>
<td>Current Process and Next Process</td>
<td>9-19</td>
</tr>
<tr>
<td>9-9</td>
<td>HOST Execution Process Status and Task Status</td>
<td>9-20</td>
</tr>
<tr>
<td>B-1</td>
<td>Multi-ICE Device Codes</td>
<td>B-1</td>
</tr>
<tr>
<td>B-2</td>
<td>ICE-85 Replacement PROM Locations</td>
<td>B-1</td>
</tr>
<tr>
<td>D-1</td>
<td>PROM Transfer Commands</td>
<td>D-1</td>
</tr>
<tr>
<td>D-2</td>
<td>PROM Compare Commands</td>
<td>D-1</td>
</tr>
</tbody>
</table>

ILLUSTRATIONS

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>TITLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2-1</td>
<td>PROG1 Program Listing</td>
<td>2-2</td>
</tr>
<tr>
<td>2-2</td>
<td>PROG2 Program Listing</td>
<td>2-4</td>
</tr>
<tr>
<td>3-1</td>
<td>A Simple Mode of Evaluation</td>
<td>3-10</td>
</tr>
<tr>
<td>4-1</td>
<td>MCS-85 PSW Instruction</td>
<td>4-22</td>
</tr>
<tr>
<td>4-2</td>
<td>MCS-48 Stack Format</td>
<td>4-24</td>
</tr>
<tr>
<td>4-3</td>
<td>MCS-48 Program Status Word (PSW)</td>
<td>4-24</td>
</tr>
<tr>
<td>5-1</td>
<td>Intellec Shared Memory At Initialization</td>
<td>5-5</td>
</tr>
<tr>
<td>5-2</td>
<td>Memory Allocation for Dual ICE-85/4X</td>
<td>5-7</td>
</tr>
<tr>
<td>9-1</td>
<td>Components of a Dual-ICE System</td>
<td>9-1</td>
</tr>
<tr>
<td>9-2</td>
<td>A Generalized Execution Process</td>
<td>9-2</td>
</tr>
<tr>
<td>9-3</td>
<td>HOST Parser Task Status</td>
<td>9-7</td>
</tr>
<tr>
<td>9-4</td>
<td>HOST Execution Process Status and Task Status</td>
<td>9-10</td>
</tr>
<tr>
<td>9-5</td>
<td>ICE Process Status and Task Status</td>
<td>9-12</td>
</tr>
<tr>
<td>9-6</td>
<td>Dispatcher Functional Diagram</td>
<td>9-15</td>
</tr>
<tr>
<td>9-7</td>
<td>Polling Sequence for processes</td>
<td>9-20</td>
</tr>
<tr>
<td>B-1</td>
<td>Intellec Series II Model 220/230</td>
<td></td>
</tr>
<tr>
<td>B-2</td>
<td>Dual-ICE Installation</td>
<td>B-2</td>
</tr>
<tr>
<td>B-2</td>
<td>ICE Boards in Dual-Auxiliary Connector</td>
<td>B-3</td>
</tr>
<tr>
<td>B-3</td>
<td>Ribbon Cable Routing Diagram</td>
<td>B-4</td>
</tr>
</tbody>
</table>
The Multi-ICE Product

The Multi-ICE product extends the capabilities of many of Intel's In-Circuit Emulator (ICE) products. The commands described in this manual permit the asynchronous debugging of a multi-processor design by operating two ICE hardware modules from one Intellec Microcomputer Development System.

Multi-ICE also contains commands that enhance the operation of each of the two ICES combined in a multi-ICE configuration.

Additionally, Multi-ICE includes facilities for defining test suites on- and off-line using diskette files, and executing the test suites in later sessions, directly from files. Thus, you can construct 'automated' test sequences that can be stored permanently or modified at will.

The manual assumes that you are familiar with the operation of at least one of the standard ICE products such as ICE-85 or ICE-49. Most of the examples in the manual are based on ICE-85.

The Multi-ICE 'package' contains:

- The Multi-ICE software diskette (single- and double-density versions). This diskette contains the combinations of ICE software that are currently supported. Each combination includes all standard ICE commands for the two ICES, all new ICE-dependent commands, and all new ICE-independent commands. Thus, one load step is all you need to have a complete multi-ICE software package at your disposal.
- A set of three replacement PROMs for the ICE-85 component of the multi-ICE system, or for one ICE-85 in an 85/85 combination. Installation instructions for these chips are given in appendix B.
- The Multi-ICE Operating Instructions (this manual).

Multi-ICE operation requires the following hardware and documentation:

- An Intellec MDS-800 system with 64K RAM, two diskette drives, console, and room for two pairs of ICE boards, or
- An Intellec Series II Model 220 or 230 system with two diskette drives, expander chassis, and 64K RAM. Installation instructions are given in appendix B.
- Two sets of ICE hardware boards with cabling, corresponding to the multi-ICE combination you wish to use.
- A standard ICE manual for each kind of ICE you are using (see preface for a list of manuals).

Multi-ICE Program Names

The Multi-ICE software diskette contains the software for all supported combinations of ICES.
To invoke any combination, you must know the name of the (ISIS-II) file that contains it. The filename is made up of the numbers of the two ICEs, with the higher number first. Thus, for example, the filename for a combination of ICE-85 and ICE-49 is “8549”, and the invocation from ISIS-II (assuming drive 1) would be:

:`F1:8549`

For a combination of two ICE-85’s, the filename is “8585”.

All Multi-ICE filenames follow this naming convention.

**About this Manual**

The multi-ICE manual contains descriptions of all the commands and keywords that are particular to the multi-ICE software system. Refer to the standard ICE manual for other commands.

Here is a brief summary of the manual’s contents:

Chapter 2 is designed to introduce the main features that are particular to the multi-ICE ‘environment’. It demonstrates how to invoke the multi-ICE software, enter commands to the HOST process, and pass command blocks to the two ICE processes.

Chapter 3 presents the details on expressions and how they are evaluated arithmetically and as TRUE/FALSE values. Expressions in Multi-ICE can contain more kinds of operators and operands than expressions available in any standard ICE; boolean (TRUE/FALSE) expressions especially are required to permit conditional execution of commands.

Chapter 4, 5, 6, 7, and 8 present the syntax and brief discussions of the new commands offered by the multi-ICE system. Table 1-1 summarizes these commands.

The commands in chapter 4 can be executed by any kind of process, HOST or ICE, and by any ICE (85 or 49) that is part of a multi-ICE combination. Here are brief descriptions of some of these commands.

The REPEAT command establishes a loop. The commands in the loop are executed iteratively. You can use an UNTIL or WHILE clause to specify when the loop is to terminate.

The COUNT command establishes a bounded loop. With this command, you can specify how many iterations of the included command sequence are to be performed. The COUNT command can also include UNTIL and WHILE clauses to exit the loop on condition, before the count is completed.

The IF command establishes conditional execution of a command sequence. The command can include ORIF and ELSE clauses to construct a branched command or test sequence.

The REPEAT, COUNT, and IF commands can be nested, that is, they can include each other.

Macros are command blocks that can be defined on-line, that is, while the ICE software is running. You can call up any defined macro by name to cause it to begin executing. The macro definition can include formal parameters; the user supplies the actual parameters when the macro is called.
# Table 1-1. Multi-ICE Commands

<table>
<thead>
<tr>
<th>Chapter Executed By</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7 &amp; 8</th>
</tr>
</thead>
<tbody>
<tr>
<td>ANY PROCESS</td>
<td>REPEAT Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>COUNT Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IF Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MACRO Commands</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BOOL Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>IND Symbol Commands</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Symbolic Display Commands</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Multiple Display Commands</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WRITE Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Any ICE-85 Process</td>
<td>SEARCH Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>DOMAIN Commands</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>NESTING Commands</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LINES Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>MODULES Commands</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>FLAG Keyword</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>LIMIT, LOWER Keywords</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>HOST Process only (must be outside ACTIVATE block)</td>
<td>SWITCH=ENn Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>ACTIVATE PRn Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>KILL PRn</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ICE Process only (must be inside ACTIVATE block)</td>
<td>KILL Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUSPEND PRn Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>CONTINUE PRn Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>WAIT PRn Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>BREAK PRn Command</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SUSPEND Command</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>


The INCLUDE command, combined with the PUT macro command enables the user to define macros on- and off-line, save them on diskette, and call them up for modification or execution at a later time.

The commands in chapter 5 can be executed by any ICE-85 process. Here are brief descriptions of some of these commands.

The SEARCH command allows you to search memory for the address containing the value you specify.

The DOMAIN command restricts the scope of a statement-number reference to the current module, eliminating the need to include the module name in each subsequent reference.

The NESTING command displays the level of procedure nesting in the user program as executed so far.

Chapters 6, 7, 8, and 9 present the multi-ICE commands. Chapter 6 describes the three basic multi-ICE commands, SWITCH, ACTIVATE, and KILL. This chapter, along with the example in chapter 2, provides a good introduction to the "world" of multi-ICE.

Chapters 7 and 8 present the rest of the multi-ICE synchronization commands.

Chapter 9 gives an extended presentation of how the multi-ICE system operates. Refer to this chapter for details on the effects of the multi-ICE commands.

The four appendices provide information for reference.

**Notation Used in this Manual**

The manual uses a notation system to convey the basic syntax and optional features of the commands.

**Basic Definitions**

For discussion of the notation, we define the following terms; examples appear later on in the discussion.

- **token**: One or more contiguous characters delimited by blanks or by context.
- **field**: A sequence of tokens that must be included (or omitted) as a unit.
- **entry**: A token or field.
- **command**: A sequence of entries terminated by a final carriage return.
- **menu**: A choice of entries presented as a vertical arrangement.
- **keyword**: A token that is defined by the system (up to the first three characters); a command literal.
- **user-name**: A token that is defined by the user.
- **meta-term**: A word or hyphenated phrase that represents a class of entries.
Notational Forms and Examples

1. Keywords are shown in ALL CAPS. For example, in the following command syntax description:
   
   EVALUATE expression SYMBOLIC
   
   The entries ‘EVALUATE’ and ‘SYMBOLIC’ are keywords and must be entered as given (up to the first three characters, ‘EVA’ and ‘SYM’ respectively).

2. Meta-terms and user-names are shown in lower-case italic. In the example just given, the entry ‘expression’ is a meta-term representing the class of all arithmetic and boolean expressions possible in Multi-ICE. As another example:
   
   DEFINE MACRO macro-name
   
   The entry ‘macro-name’ represents a token whose form (character sequence in this case) is specified by the user.

   As special cases of this notational form, the entry ‘ENn’ means one of the tokens ‘EN1’ or ‘EN2’, and ‘PRn’ means one of the tokens ‘PR1’ or ‘PR2’. (See chapter 2 for the use of these tokens.)

   Additionally, the entry ‘cr’ represents an intermediate carriage return, ending an inner line of a multi-line command. Final carriage returns are not shown in the syntax.

3. A pair of brackets [ ] enclosing a single entry means that entry is optional. For example:
   
   REMOVE MACRO [macro-list]
   
   The field represented by ‘macro-list’ can be omitted, leaving a valid command. Of course, omitting an entry can change the effect of the command when it is executed.

4. A pair of brackets enclosing a single entry and followed by an ellipsis (three periods) means that entry may be omitted, used once, or repeated indefinitely. For example:
   
   ACTIVATE PRn
       [command cr]...
    ENDACTIVATE
   
   The entry ‘command cr’ means that this field can be omitted (‘null’ command), or one command can be entered with an intermediate carriage return, or as many such commands as desired, until the ENDACTIVATE keyword is entered to signal the termination of the command block.

5. A pair of brackets around a menu of entries means ‘select none or one—not more than one’. For example,
   
   SEARCH [SINGLE] partition [mask-field] FOR value
       [DOUBLE]
   
   In this command, the entries ‘SINGLE’ and ‘DOUBLE’ are mutually exclusive. One or the other can be used, or neither—but not both.

6. A pair of brackets around a menu of entries and followed by the ellipsis means ‘select none, one, or more than one, in any order’. There are two exceptions to this convention; both are noted below. As an example of the majority rule:
   
   REPEAT cr
       [command cr]
   WHILE boolean-expression cr
       [UNTIL boolean-expression cr]...
    ENDREPEAT
   
   In this compound command, the body can consist of any number of commands, WHILE fields, and UNTIL fields, in any order.
Exceptions: the same notational symbols are used with the WRITE command and the multiple displays. These commands require at least one entry be selected from the menu to produce a valid command. The syntaxes are:

\[
\text{WRITE} \left[ \begin{array}{c} \\
\text{string} \\
\text{expression} \\
\text{BOOL \ boolean-expression} \\
\text{keyword-reference} \\
\text{content-expression} \\
\end{array} \right] [\ldots] \\
\left[ \ldots \right]
\]

7. The symbol:

\[ [\ldots] \]

After an entry or menu means, ‘if repeated, use commas to separate entries’.
This chapter contains a brief example of multi-ICE operation, and introduces a few of the basic multi-ICE commands.

Sample Programs

The example involves ‘debugging’ two programs, PROG1 and PROG2. Each is written in PL/M-80, to be run on an MCS-85 system such as an iSBC 80/30 with multi-ICE-85.

PROG1 is listed in figure 2-1, and PROG2 is shown in figure 2-2.

The ‘user’ hardware configuration for this example could consist of two iSBC 80/30 boards (let’s call then ‘board 1’ and ‘board 2’) with an external console attached to the serial I/O port (connector J3) of board 1.

PROG1 and PROG2 demonstrate a rudimentary form of handshaking between two processors. For simplicity, we’ll assume PROG1 is running on board 1 and PROG2 on board 2; they communicate through a common external memory (RAM) board.

PROG1 reads a character from the console connected to board 1, and sets a flag when it has finished reading a character. PROG2 checks the flag, changes the character into either a lower case character or a question mark, and sets another flag. PROG1 checks this second flag; when the second flag is set, PROG1 displays the resulting character on the external console, then prompts for another character.

In our example, location 8000H stores the character both as input and as changed by PROG2. Location 8001H contains the flags; bit 1 is the "character input" flag from PROG1 to PROG2 (1=set), and bit 0 is the "output character" flag from PROG2 to PROG1.

As shown in figure 2-2, PROG 2 has “bugs” in procedure START2: P1 should be 8000H and P2 should be 8001H. We shall “find” and patch these bugs in the sample ICE session.

Other details on PROG1 and PROG2 are given as comments in the listings; these details are of no importance to the command examples.

Installation and Configuration Summary

Installation steps are given in full in Appendix B; we summarize them briefly for this example.

The multi-ICE system we are using is 85/85, that is, a combination of two ICE-85's.

- Selecting one set of ICE-85 boards “at random” to be the “first ICE”, we install the three replacement PROMs and set the device code to 10H, both on the Controller board of the first ICE.
- We install the first set of ICE boards in the main chassis of an Intellec Series II model 230.
PROG1:
DO;

/*
PROG1
THIS PROGRAM EXECUTES IN A 80/30 BOARD ENVIRONMENT. THE
PROGRAM READS INPUT FROM EXTERNAL CONSOLE AND STORES INPUT IN
OFFBOARD RAM (LOCATION 8000). A FLAG BIT IS SET (BIT 1 IN BYTE 8001H)
SIGNIFYING THAT CHARACTER HAS BEEN READ. ANOTHER PROGRAM
(PROG2) WHICH IS EXECUTING IN ANOTHER 80/30 ENVIRONMENT CHECKS
THIS FLAG WAITING FOR CHARACTER READY STATE (BIT 1 ON IN BYTE 8001).
*/

/*PROGRAM DECLARATIONS*/
DECLARE FOREVER LITERALLY "WHILE1";
DECLARE CR LITERALLY '0DH';
DECLARE LF LITERALLY '0AH';
DECLARE (P1,P2) ADDRESS;
DECLARE CHAR BASED P1 BYTE;
DECLARE CNTRL BASED P2 BYTE;

* CONSOLE OUTPUT ROUTINE
OUT: PROCEDURE(CHAR);
DECLARE CHAR BYTE;
WT: IF (INPUT(OEDH) AND 01H) = 0 THEN GOTO WT;
"WAIT UNTIL OUTPUT BUFFER IS EMPTY"/
OUTPUT (OECH) = CHAR;
END OUT;

* CONSOLE INPUT ROUTINE
IN: PROCEDURE BYTE;
"OEDH - STATUS PORT OF 8251 IN 80/30, OECH - DATA PORT"/
WT: IF (INPUT(OEDH) AND 2H) = 0 THEN GOTO WT;
"WAIT UNTIL CHAR IS PRESENT"/
RETURN (INPUT(OECH) AND 7FH); "RETURN CHAR INPUT"/
END IN;

Figure 2-1. PROG1 Program Listing


/*
 * END OF LINE PROCEDURE, OUTPUT CARRIAGE RETURN AND LINEFEED
 */
19 1 ENDLINE:
   PROCEDURE;
20 2 CALL OUT(CR);
21 2 CALL OUT(LF);
22 2 END ENDLINE;

/*
 * START OF PROGRAM PROG1
 * THE CONSOLE IS MONITORED FOR ANY USER INPUT
 */

/* THE FOLLOWING OUTPUT COMMANDS ARE 80/30 DEPENDENT
 AND ARE USED TO INITIALIZE THE BAUD RATE CLOCK AND 8251 CHIP */

23 1 DISABLE;
24 1 OUTPUT(0EDH) = 4FH; /* MODE INSTRUCTION TO USART: 64X, 8 BITS LONG
   NO PARITY, 1 STOP BIT */
25 1 OUTPUT(0EDH) = 27H; /* COMMAND TO USART */
26 1 OUTPUT(0DFH) = 0B6H; /* SET COUNTER 2 TO MODE 3 BAUD RATE
   GENERATOR */
27 1 OUTPUT(0DEH) = 08H; /* 8H WILL MAKE COUNTER 3 GENERATE CLOCK
   PULSES FOR 2400 BAUD */
28 1 OUTPUT(0DEH) = 0;  

29 1 START1:
   P1 = 8000H;
30 1 P2 = 8001H;
31 1 CNTRL = 0; /* INITIALIZE CNTRL TO 0 */
32 1 DO FOREVER;

33 2 CALL OUT('"'); /* OUTPUT TERMINAL PROMPT CHARACTER */
34 2 CHAR = IN; /* GET KEYBOARD INPUT, PUT IN LOCATION 8000H */
35 2 CNTRL = 2H; /* INPUT CHARACTER READY BIT SET IN
   HANDSHAKING BUFFER, PROG2 RUNNING IN
   SECOND 80/30 IS WAITING FOR THIS BIT TO BE SET
   SO IT CAN READ CHARACTER INPUT WHICH IS IN
   DATA BUFFER */
36 2 CALL ENDLINE 

37 2 WAIT:
   IF (CNTRL AND 1H) = 0 THEN GOTO WAIT; /* LOOP UNTIL OUTPUT BIT SET IN
   CNTRL 1 IN BIT 1 MEANS THERE IS CHARACTER TO OUTPUT */

39 2 CALL OUT(CHAR); /* OUTPUT CHARACTER PASSED FROM OTHER 80/30
   PROGRAM */
40 2 CALL ENDLINE; /* OUTPUT CARRIAGE RETURN LINEFEED */
41 2 END; /* END DO FOREVER, LOOP BACK FOR MORE INPUT */
42 1 END PROG1; /* END OF MODULE */

Figure 2-1. PROG1 Program Listing (Cont’d.)
PROG2:
DO;

/*
PROG2
THIS PROGRAM EXECUTES IN A 80/30 BOARD WHICH SENDS AND RECEIVES
DATA FROM ANOTHER 80/30 THROUGH LOCATIONS 8000H AND 8001H WHICH
ARE IN COMMON MEMORY. LOCATION 8000H CONTAINS A CHARACTER
READ FROM EXTERNAL CONSOLE BY PROG1 AND LOCATION 8001H IS A
CHARACTER FLAG BYTE BIT 0 ON (OF LOCATION 8001H) MEANS THERE IS A
CHARACTER TO OUTPUT, BIT 1 ON (OF LOCATION 8001H) MEANS THERE IS A
CHARACTER TO READ. IF CHARACTER IS UPPERCASE LETTER PROG2 CON­
VERTS CHARACTER TO LOWER CASE AND SETS BIT 0 (8001H) ON. IF
CHARACTER INPUT FROM EXTERNAL CONSOLE BY PROG1 AND IN LOCA­
TION 8000H IS NOT UPPERCASE LETTER THEN SET CHARACTER TO ?.
CHARACTER IS THEN READ BY PROG1 AND OUTPUT TO CONSOLE.
*/

/*PROGRAM DECLARATIONS*/

DECLARE FOREVER LITERALLY 'WHILE 1';
DECLARE (P1, P2) ADDRESS; /*LOCATIONS OF
CHARACTER AND
CHARACTER FLAG BIT*/
DECLARE CHARACTER BASED P1 BYTE;
DECLARE CONTROL BASED P2 BYTE;

START2:
P1 = 3000H; /*3001H
AND 3000H SHOULD BE 8000H AND 8001H BUT ARE HERE
AS BUGS*/
P2 = 3001H;

DO FOREVER;

LOOP:
IF (CONTROL AND 2H) = 0 THEN GOTO LOOP; /*WAIT FOR CHARACTER
INPUT*/
IF CHARACTER >= 41H AND CHARACTER <= 5AH THEN CHARACTER =
CHARACTER + 20H;
ELSE CHARACTER = '?';
CONTROL = 1H ; /*SET CONTROL FOR OUTPUT CHAR READY*/
END; /*END FOREVER*/

END PROG2; /*END OF MODULE*/

Figure 2-2. PROG2 Program Listing
• We set the device code of the “second ICE” to 11H, and install it in the expander chassis attached to the Intellec system.
• We turn on the power to the Intellec system and diskette drive.
• We insert an ISIS-II system diskette in drive 0 and the diskette containing both the multi-ICE software and our test programs (PROG1 and PROG2) into drive 1.

This completes the installation. Now we proceed to the debugging session.

Sample Multi-ICE Session

Boot ISIS-II by pressing the RESET button on the Intellec front panel, and obtain the hyphen prompt.

Enter the drive and filename containing the multi-ICE-85 software:

\[-:F1:8585\]

Multi-ICE signals readiness with the sign-on message:

\[ISIS-II \text{ MULTI-ICE } 85/85, \text{ Vx} \cdot \text{x}\]
\[\text{PR1 IS ICE-85 DEVICE CODE 10H, PR2 IS ICE-85 DEVICE CODE 11H}\]

The asterisk prompt tells us we can enter ICE commands. Before getting to the commands, let’s introduce the multi-ICE keywords we shall use (table 2-1).

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST</td>
<td>Default execution process.</td>
</tr>
<tr>
<td>PR1</td>
<td>Execution process for ICE with lowest device code. Executes commands within &quot;ACTIVATE PR1&quot; block.</td>
</tr>
<tr>
<td>PR2</td>
<td>Execution process for ICE with highest device code. Executes commands within &quot;ACTIVATE PR2&quot; block.</td>
</tr>
<tr>
<td>EN1</td>
<td>PR1's parsing and execution environment, available to HOST via SWITCH command.</td>
</tr>
<tr>
<td>EN2</td>
<td>PR2's parsing and execution environment, available to HOST via SWITCH command.</td>
</tr>
<tr>
<td>SWITCH</td>
<td>Keyword to display or change HOST's environment.</td>
</tr>
<tr>
<td>ACTIVATE</td>
<td>Keyword to begin block of commands to be executed by ICE process (PR1 or PR2).</td>
</tr>
<tr>
<td>ENDACTIVATE</td>
<td>End of command block for ICE process.</td>
</tr>
<tr>
<td>KILL</td>
<td>Command to halt processing by PR1 or PR2 (or both), and discard that process current command block.</td>
</tr>
<tr>
<td>ALL</td>
<td>Keyword meaning &quot;both PR1 and PR2&quot;.</td>
</tr>
</tbody>
</table>
Any commands we enter at this time are executed by the default process, called the "HOST process," or just HOST. The HOST can use hardware and software resources of either of the two ICEs.

At initialization, the HOST is in the "environment" of the first ICE-85. This ICE is to execute PROG1, and we can map and load this program for EN1, where it is available to the HOST and to PR1.

The map and load sequence for PROG1 is as follows:

```
*MAP 4000 = USER
*MAP 8000 = USER
*MAP IO 0 TO FF = USER
*LOAD :F1:PROG1
*S SYMBOL
MODU LE ..PROG1
.MEMORY=409CH
.P1=4097H
.P2=4099H
.OUT=4055H
.CHAR=409BH
.WT=4059H
.IN=406BH
.WT=406BH
.ENDLINE=407CH
.START1=4017H
.WAIT=4038H
```

We display the symbol table for PROG1 to verify that the program loaded.

Now, let's do the same thing for PROG2. First, we SWITCH to EN2, the environment of PR2, the second ICE-85; this ICE is going to execute PROG2.

```
*SWITCH = EN2
*MAP 4000 = USER
*MAP 8000 = USER
*LOAD :F1:PROG2
*S SYMBOL
MODU LE ..PROG2
.MEMORY=405BH
.P1=4057H
.P2=4059H
.START2=4003H
.Loop=400DH
```

We display PROG2's symbol table to verify program load.

Now, even though the "current SWITCH" is EN2, we can start PR1 emulating by entering the following commands.

```
*ACTIVATE PR1
.*GO
.*ENDA
PR1 EMULATION BEGUN
```

When the HOST encounters the ACTIVATE PR1..ENDA block, it passes the commands inside the block to PR1 for execution. In this case, the block had only the one "GO" command; PR1 executes this command and signals "EMULATION BEGUN".
Now, since PRI is executing a command block, the prompt is suppressed. This happens when any process is executing. You can always press the ESC key to abort processing, erase the command or command block being executed, and get the prompt.

However, when either PRI or PR2 (not the HOST) is executing or emulating, you can press the spacebar to get the prompt without erasing any commands. If either ICE process is emulating ("GO" command) when the spacebar is pressed, it continues emulating; otherwise, execution halts after the current command and remains halted until the final carriage return is entered.

We want PRI to continue emulating, so we press the spacebar to get the HOST's attention, and start PR2 emulating:

```
*ACTIVATE PR2 ;SPACE BAR WAS PRESSED TO GET THE PROMPT
  *GO
  *ENDA
PR2 EMULATION BEGUN
PR2 ERR 42:GUARDED ACCESS
```

PR2 signals "EMULATION BEGUN", then encounters an address that is outside the mapped area. The attempt to access this "GUARDED" location terminates PR2's emulation, but PRI is not affected.

However, we want to stop PRI so we don't have to press the spacebar every time we want a prompt.

We press the spacebar and obtain the prompt. Now we enter the command:

```
*KILL PR1 ;AGAIN, SPACE BAR WAS PRESSED.
PR1 EMULATION TERMINATED, PC=..PROG1#16
```

The KILL PRI command halts PRI and discards its command block. PRI signals "EMULATION TERMINATED", and displays the current PC at the end of emulation (PC is the next instruction that would be executed if emulation begins again).

Note that the PC value is displayed "symbolically", in this case by displaying the module and line number corresponding to the address currently in PC. This kind of symbolic display is the default under multi-ICE, for a number of address-type displays.

After PRI’s message, the prompt appears automatically.

We’re talking to the HOST again; we display the HOST’s environment by entering:

```
*SWITCH
EN2 (ICE-85)
```

Now to examine the trace buffer of PR2 to see where the GUARDED ACCESS occurred. Since we’re in EN2, the HOST can perform the display:

```
*;EXAMINE WHAT CAUSED THE GUARDED ACCESS.
*PRINT ALL
HOST/EN2:
  ADDR INSTRUCTION ADDR-S-DA ADDR-S-DA ADDR-S-DA
0001:  4000 LXI SP, 4057
0007:  4003 LXI H, 3000
0013:  4006 SHLD 4057 4057-W-00 4058-W-30
```
Getting Started With Multi-ICE

0023: 4009 INX H
0025: 400A SHLD 4059
0035: 400D LHLD 4059
0045: 4010 MVI A,02
0049: 4012 ANA M

*;LINE 6 IN PROG2 SHOULD BE "P1 = 8000;"

We see in frame 0049 that address 3001H appears; this is the guarded location.

Let's look at the contents of "code memory" to see where a "patch" might be inserted. Refer to figure 2-2 for the source code.

*;LINE 6 IN PROG2 SHOULD BE "P1 = 8000;"
* BYTE .START2 LENGTH 10
..PROG2.START2
4003H=21H 00H 30H 22H 57H 40H
..PROG2#7
4009H=23H 22H 59H 40H
..PROG2.Loop
400DH=2AH 59H 40H 3EH 02H A6H
* BYTE 4005 = 80

We request 16 contiguous bytes (10H = 16 decimal). The display gives the addresses in hexadecimal, unless an exact match to a symbol is encountered; exact matches are displayed as the symbol or line number, then the corresponding hex address on the next line.

The display involves three symbolic locations in PROG2 (.START2, #7, and .LOOP).

By inspection, we note the address "3000H" in bytes 4004 and 4005; we want to change this to "8000H", and do so by setting "BYTE 4005 = 80". Later on, we'll edit the source listing and recompile.

Now we can start the parallel emulation again; this time we begin from the main loop in each program.

*ACTIVATE PR1
.*GO FROM .START1
.*ENDA
PR1 EMULATION BEGUN
*ACTIVATE PR2;SPACEBAR WAS PRESSED
.*GO FROM .START2
.*ENDA
PR2 EMULATION BEGUN

This listing does not show the input to or output from PROG1, since they involve a terminal external to the Intellec system.

Assuming that the desired effect is obtained, we request the prompt, KILL both PR1 and PR2, and EXIT from multi-ICE back to ISIS-II. The session is completed.

*KILL ALL;SPACE BAR WAS PRESSED.
PR1 EMULATION TERMINATED, PC=..PROG1.IN
PR2 EMULATION TERMINATED, PC=..PROG2. LOOP +0003H
*EXIT

Refer to chapters 6, 7, 8, and 9 for the details on all the multi-ICE commands. Chapters 3, 4, and 5 describe extensions to the multi-ICE command set that are available in multi-ICE systems.
An expression is a formula that evaluates to a number. The formula can contain operands, operators, and parentheses. Depending on the command context, the resulting number is interpreted either as an integer value or as a logical state (TRUE/FALSE).

This chapter shows you how to construct expressions by presenting the types of operands and operators that can be used, and by giving rules and examples to explain how expressions are evaluated. The chapter also describes numeric and logical (boolean) command contexts and various other contexts for interpreting numeric results.

**Operands**

Operands are numeric values. Numeric values in expressions are represented by primaries; a primary is a constant or reference. ICE "resolves" each primary operand into a numeric operand by some form of "lookup" before evaluation proceeds.

The next several sections discuss numbers under Multi ICE and review the types of primaries that can be entered as operands in expressions.

**Numbers**

Under Multi ICE, all numbers are unsigned binary constants. In most cases, expressions operate on and result in 16-bit numbers, with all operations treated modulo 65536 ($2^{16}$); thus, $65535 + 1 = 0$, and $0 - 1 = 65535$.

A numeric operand obtained by looking up a primary is forced to 16 bits for evaluation. If the number has more than 16 bits, the high-order bits after the sixteenth are discarded; if the number has fewer than 16 bits, the high-order bits are filled with zeros.

**Primaries**

A primary is one of the following types of tokens:

- **constants:**
  - numeric constant
  - string constant
- **direct references:**
  - keyword-reference
  - symbolic-reference
  - statement-reference
  - process-reference

All these entities can be resolved into numeric operands without using any operators. The next several paragraphs give details on the various types of primaries.
Numeric Constants

A numeric constant is simply a way of representing a number in an ICE command. A numeric constant consists of one or more digits, and (optionally) a one-character explicit radix to identify the number base. The elements of numeric constants are summarized in table 3-1.

<table>
<thead>
<tr>
<th>Number Base</th>
<th>Valid Digits</th>
<th>Explicit Radix</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Binary (base 2)</td>
<td>0, 1</td>
<td>Y</td>
<td>11110011Y</td>
</tr>
<tr>
<td>Octal (base 8)</td>
<td>0 - 7</td>
<td>Q, O</td>
<td>363Q</td>
</tr>
<tr>
<td>Decimal (base 10)</td>
<td>0 - 9</td>
<td>T</td>
<td>243T</td>
</tr>
<tr>
<td>Hexadecimal (base 16)</td>
<td>0 - 9, A - F</td>
<td>H</td>
<td>00F3H</td>
</tr>
<tr>
<td>Decimal Multiple of 1024T</td>
<td>0 - 9</td>
<td>K</td>
<td>4K</td>
</tr>
</tbody>
</table>

Numeric constants with explicit radix K (decimal multiple of 1024T) can be used in expressions; however, the implicit radix cannot be set to K with the SUFFIX command.

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

A numeric constant without an explicit radix entered from the console 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, T, or H by using the SUFFIX command. Refer to the standard ICE manuals for details on the SUFFIX command.

Certain contexts assume an implicit decimal (T) radix. In these contexts, any numeric constant entered without an explicit radix is interpreted as a decimal number. The COUNT command described in this manual is one such context; others (for examples, the COUNT clause after STEP; PRINT; MOVE; statement-numbers; and channel-group names) are discussed in the standard ICE manuals. The SUFFIX command has no effect on these contexts. More details on command contexts appear later in this chapter.

In this manual, most numeric constants are shown with explicit radixes for clarity. Numbers without explicit radixes are decimal (for example, 65536), except for the numbers 0 and 1, to which any radix can apply.

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 number with the nine 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 000000000100001Y (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 number; the high byte is all zeros, and the low byte contains the 8-bit ASCII value with the parity bit set to 0.
Table 3-2 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 3-2. 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>@</td>
<td>40</td>
<td>a</td>
<td>61</td>
</tr>
<tr>
<td>!</td>
<td>21</td>
<td>A</td>
<td>41</td>
<td>b</td>
<td>62</td>
</tr>
<tr>
<td>&quot;</td>
<td>22</td>
<td>B</td>
<td>42</td>
<td>c</td>
<td>63</td>
</tr>
<tr>
<td>#</td>
<td>23</td>
<td>C</td>
<td>43</td>
<td>d</td>
<td>64</td>
</tr>
<tr>
<td>$</td>
<td>24</td>
<td>D</td>
<td>44</td>
<td>e</td>
<td>65</td>
</tr>
<tr>
<td>%</td>
<td>25</td>
<td>E</td>
<td>45</td>
<td>f</td>
<td>66</td>
</tr>
<tr>
<td>&amp;</td>
<td>26</td>
<td>F</td>
<td>46</td>
<td>g</td>
<td>67</td>
</tr>
<tr>
<td>(</td>
<td>27</td>
<td>G</td>
<td>47</td>
<td></td>
<td></td>
</tr>
<tr>
<td>)</td>
<td>28</td>
<td>H</td>
<td>48</td>
<td>h</td>
<td>68</td>
</tr>
<tr>
<td>+</td>
<td>29</td>
<td>I</td>
<td>49</td>
<td>i</td>
<td>69</td>
</tr>
<tr>
<td>-</td>
<td>2A</td>
<td>J</td>
<td>4A</td>
<td>l</td>
<td>6A</td>
</tr>
<tr>
<td>.</td>
<td>2B</td>
<td>K</td>
<td>4B</td>
<td>k</td>
<td>6B</td>
</tr>
<tr>
<td>/</td>
<td>2C</td>
<td>L</td>
<td>4C</td>
<td>l</td>
<td>6C</td>
</tr>
<tr>
<td>0</td>
<td>2D</td>
<td>M</td>
<td>4D</td>
<td>m</td>
<td>6D</td>
</tr>
<tr>
<td>1</td>
<td>2E</td>
<td>N</td>
<td>4E</td>
<td>n</td>
<td>6E</td>
</tr>
<tr>
<td>2</td>
<td>2F</td>
<td>O</td>
<td>4F</td>
<td>o</td>
<td>6F</td>
</tr>
<tr>
<td>3</td>
<td>30</td>
<td>P</td>
<td>50</td>
<td>p</td>
<td>70</td>
</tr>
<tr>
<td>4</td>
<td>31</td>
<td>Q</td>
<td>51</td>
<td>q</td>
<td>71</td>
</tr>
<tr>
<td>5</td>
<td>32</td>
<td>R</td>
<td>52</td>
<td>r</td>
<td>72</td>
</tr>
<tr>
<td>6</td>
<td>33</td>
<td>S</td>
<td>53</td>
<td>s</td>
<td>73</td>
</tr>
<tr>
<td>7</td>
<td>34</td>
<td>T</td>
<td>54</td>
<td>t</td>
<td>74</td>
</tr>
<tr>
<td>8</td>
<td>35</td>
<td>U</td>
<td>55</td>
<td>u</td>
<td>75</td>
</tr>
<tr>
<td>9</td>
<td>36</td>
<td>V</td>
<td>56</td>
<td>v</td>
<td>76</td>
</tr>
<tr>
<td>&lt;</td>
<td>37</td>
<td>W</td>
<td>57</td>
<td>w</td>
<td>77</td>
</tr>
<tr>
<td>=</td>
<td>38</td>
<td>X</td>
<td>58</td>
<td>x</td>
<td>78</td>
</tr>
<tr>
<td>&gt;</td>
<td>39</td>
<td>Y</td>
<td>59</td>
<td>y</td>
<td>79</td>
</tr>
<tr>
<td>?</td>
<td>3A</td>
<td>Z</td>
<td>5A</td>
<td>z</td>
<td>7A</td>
</tr>
<tr>
<td>&lt;</td>
<td>3B</td>
<td>]</td>
<td>5B</td>
<td></td>
<td></td>
</tr>
<tr>
<td>=</td>
<td>3C</td>
<td>{</td>
<td>5C</td>
<td>l</td>
<td>7C</td>
</tr>
<tr>
<td>&gt;</td>
<td>3D</td>
<td>^</td>
<td>5D</td>
<td></td>
<td>7D</td>
</tr>
<tr>
<td>?</td>
<td>3E</td>
<td>(−)</td>
<td>5E</td>
<td></td>
<td>7E</td>
</tr>
</tbody>
</table>

Direct References

A direct reference is a keyword reference, symbolic reference, statement-number
reference, or process reference. All these references produce a numeric operand
value through some form of "lookup". Direct references differ from numeric and
string constants in one important aspect. If the expression containing a direct
reference is evaluated more than once (for example, in a loop), the direct reference
can have a different value each time the expression is evaluated.

Keyword References

A keyword reference is the name of a register, buffer, or flag accessible to ICE.
When a keyword reference is entered as an arithmetic operand, the current content
or setting of the register is used in the evaluation.

Symbolic References

A symbolic reference points to an entry in some ICE symbol table. Corresponding to
each symbol table entry is a number that represents an address or other value. When
a symbolic reference is entered as an arithmetic operand, its corresponding value is
obtained from the appropriate symbol table and used in the evaluation.
Expressions

Statement-Number References

Under ICE-85, an entry such as "#56" in an expression causes the address of the first instruction at that line number in the source code to be used as an operand.

With ICE-85, statement-number information for each module of high-level code (PL/M, FORTRAN, or other high-level language) is stored in tables, one for each module of code loaded. Since the LINK process combines different modules, each with its own set of statement numbers, the reference may require a module identifier; the above example might become "..CAR#56" to distinguish the reference from line number 56 in another module. Statement numbers are interpreted in decimal radix if they have no explicit suffix. Refer to the DOMAIN command in Chapter 5, and to the ICE-85 manual for details.

Statement numbers are invalid in ICE-49 and ICE-41.

Process References

This form of primary occurs in multi-ICE operations. A process reference has the form process-status process. Process is a token of the form PRn, where n is the logical number of the ICE whose status is to be examined (examples: PR1, PR2), or the keyword HOST. Process-status means one of the keywords ACTIVE, SUSPENDED, or DORMANT. If the indicated ICE process is currently in the specified status, the operand value of the process reference is TRUE (FFFFH); otherwise it is FALSE (0).

Operators

An expression can contain any combination of unary and binary operators. Table 3-3 describes all the operators available under Multi-ICE. The operators are ranked in order of precedence from highest (1) to lowest (9); 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 of content-operators has too many details to fit in the table; see table 3-5. The table identifies each operator as unary or binary; a unary operator takes one operand, and a binary operator takes two operands. The brief descriptions of the operations in the table are supplemented by the text.

Classes of Operators

For discussion, the operators are classed as shown in table 3-4.

Arithmetic Operators

The ICE scanner distinguishes unary "+' and "-" from binary "+' and "-" by context. Unary "+' is superfluous, since all numbers are assumed to be positive.

The unary "-" has two meanings in ICE commands. In the MOVE and PRINT commands, "-" means "toward the top (earliest entry) in the trace buffer"; in this context, "-" is a command token rather than an operator. In all other contexts, a unary "-" means "2's complement modulo 65536"; in other words, (-N) evaluates to (65536 - N).

Binary "+' results in the arithmetic sum of its two operands; the result is treated modulo 65536 (any high-order bits after the sixteenth bit are dropped). Binary "-" results in the arithmetic difference of its two operands; this result is also treated modulo 65536, so that a "negative" result (-N) ends up as (65536 - N).
### Table 3-3. Enhanced-ICE Operators

<table>
<thead>
<tr>
<th>Precedence Class</th>
<th>Operator</th>
<th>Unary Binary</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>+</td>
<td>u</td>
<td>Unary plus.</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>u</td>
<td>Unary minus. ((-N)) means ((65536-N)), the 2’s complement of (N), module (2^4)</td>
</tr>
<tr>
<td>2</td>
<td>*</td>
<td>b</td>
<td>Integer multiplication.</td>
</tr>
<tr>
<td></td>
<td>/</td>
<td>b</td>
<td>Integer division. The result is the integer quotient; the remainder (if any) is lost.</td>
</tr>
<tr>
<td></td>
<td>MOD</td>
<td>b</td>
<td>Modulo reduction. The remainder after division, expressed as an integer.</td>
</tr>
<tr>
<td>3</td>
<td>+</td>
<td>b</td>
<td>Addition.</td>
</tr>
<tr>
<td></td>
<td>−</td>
<td>b</td>
<td>Subtraction.</td>
</tr>
<tr>
<td>4</td>
<td>MASK</td>
<td>b</td>
<td>Bitwise AND. Higher precedence than identical operation AND (see below).</td>
</tr>
<tr>
<td>5</td>
<td>content-operator</td>
<td>u</td>
<td>Treats operand as memory or port address, returns the content of that address.</td>
</tr>
<tr>
<td>6</td>
<td>=</td>
<td>b</td>
<td>Is equal to. Result is either TRUE (FFFFH) or FALSE (0).</td>
</tr>
<tr>
<td></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>&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>7</td>
<td>NOT</td>
<td>u</td>
<td>Unary Logical (1’s) complement. 1 becomes 0, 0 becomes 1; TRUE becomes FALSE, FALSE becomes TRUE.</td>
</tr>
<tr>
<td>8</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>9</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. 1 = highest precedence (evaluated first), 9 = lowest precedence.
2. u = unary, b = binary.
3. Refer to text for additional details.
4. *content-operator* is one of the tokens BYTE, IBYTE, WORD, IWORD, PORT, CBYTE, DBYTE, or XBYTE. See text and table 3-5.
### Table 3-4. Classes of Operators

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

Binary “*” results in the multiplication of its two operands, truncated to the low-order 16 bits.

Binary “/” divides the first operand 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 division as an integer result, and the quotient part of the division is lost. Thus, \((5 \text{ MOD } 3)\) evaluates to \((2)\), the remainder of \((5/3)\).

Binary “MASK” performs a bitwise logical AND on its two 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.

Unary “-” has 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.

### Content Operators

The content operators are keywords that refer to the contents of memory addresses and I/O ports. In expressions, they function as unary operators with precedence next lower after “MASK”. Each ICE has its own set of content operators. Table 3-5 summarizes the content operators for ICE-85, ICE-49, and ICE-41. Refer to the standard ICE manuals for more detail.

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. Refer to the standard ICE manuals for details on addresses, partitions, and memory mapping.
Table 3-5. Content Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Content Returned</th>
<th>ICEs</th>
</tr>
</thead>
<tbody>
<tr>
<td>BYTE</td>
<td>Byte at address in mapped memory.</td>
<td>85</td>
</tr>
<tr>
<td>WORD</td>
<td>Word (two consecutive bytes) at address in mapped memory.</td>
<td>85</td>
</tr>
<tr>
<td>IBYTE</td>
<td>Byte at address in Intellec shared memory.</td>
<td>85</td>
</tr>
<tr>
<td>IWORD</td>
<td>Word at address in Intellec shared memory.</td>
<td>85</td>
</tr>
<tr>
<td>PORTI</td>
<td>Byte content of addressed port, as mapped.</td>
<td>85</td>
</tr>
<tr>
<td>CBYTE</td>
<td>Byte at address in user code memory.</td>
<td>49, 41</td>
</tr>
<tr>
<td>DBYTE</td>
<td>Byte at address in user data memory.</td>
<td>49, 41</td>
</tr>
<tr>
<td>XBYTE</td>
<td>Byte at address in user external data memory.</td>
<td>49</td>
</tr>
</tbody>
</table>

Note:
1. ICE-49 and ICE-41 use keywords such as P0 and P1 to refer to I/O port contents.

Relational Operators

A relational operator calls for a comparison of the values of its two operands. The six possible comparisons are shown in Table 3-3. 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” operator results in a 1’s complement of its operand; a 16-bit operand is assumed. Here are some examples:

\[
\begin{align*}
\text{NOT } 0 & \rightarrow \text{FFFFH} \\
\text{NOT } 1 & \rightarrow \text{FFFEH} \\
\text{NOT } \text{11110110Y} & \rightarrow \text{1111111100001001Y} \\
\text{NOT } \text{FFFFH} & \rightarrow 0
\end{align*}
\]

The result of “AND” on any pair of corresponding bits in its two operands 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>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:

\[
\begin{align*}
0 \text{ AND } 0 & \rightarrow 0 \\
1010Y \text{ AND } 1001Y & \rightarrow 1000Y \\
\text{FFFFH AND } 0 & \rightarrow 0 \\
\text{FFFFH AND FFFFH} & \rightarrow \text{FFFFH} \\
1 \text{ AND } 0 & \rightarrow 0
\end{align*}
\]
The result of an "OR" operation on any pair of corresponding bits in its two operands 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>1</td>
</tr>
</tbody>
</table>

Examples:

0 OR 0 → 0
1 OR 0 → 1
1010Y OR 1001Y → 1011Y
FFFFH OR 0 → FFFFH
FFFFH OR FFFFH → FFFFH

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:

0 XOR 0 → 0
1 XOR 0 → 1
1010Y XOR 1001Y → 11Y
FFFFH XOR 0 → FFFFH
FFFFH XOR FFFFH → 0

**Environment Controls**

An environment control is one of the keywords EN1 or EN2. When an environment control is used in an expression, ICE uses that environment for all references. There are two ways to use an environment control:

1. **ENn primary**

2. **ENn (expression)**

With the first form, the environmental control applies only to the primary (reference) that immediately follows. Thus, the scope of an environmental control without parenthesis is the next primary.

Parentheses can be used to extend the scope of an environmental control to an expression. The parentheses also control the order of evaluation.

Environmental controls are useful when one ICE wants to access a reference to the other ICE from within an ACTIVATE list (see Chapter 6 for more details).

An environmental control is treated like a unary operator with highest precedence.
How Expressions are Evaluated

Here is a simple conceptual model of how ICE evaluates an expression. The model involves a loop that scans the expression iteratively (figure 3-1). The loop terminates in two ways:

- When nothing remains except a single number.
- When a syntax error (or other error) occurs.

ICE 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 3-3)
- 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 numeric 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 number; that is:

\[(\text{number}) \rightarrow \text{number}\]

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. In some examples, the steps in evaluation are shown, but most show just the overall result. Table 3-6 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 3-7. This table also describes the special notation used in some of the examples. The examples also assume SUFFIX = T; that is, any number without an explicit radix is decimal.

Case 1: EVALUATE primary

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

Examples:

```
EVA 10
1010Y 12Q 10T AH ""

EVA PC
1000000000000Y 10000Q 4096T 1000H ""

EVA.AA
10000000000000Y 20000Q 8192T 2000H ""
```
Figure 3-1. A Simple Model of Evaluation
Case 2: EVALUATE unary-operator primary

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

Examples:

```
EVA -2
1111111111111110Y 177776Q 65534T FFFEH '†'
```

```
EVA BYTE .AA
100011Y 43Q 35T 23H '#'
```

```
EVA NOT PC
1110111111111111Y 167777Q 61439T EFFFH '0'
```

**Table 3-6. 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>primary</td>
<td>None</td>
<td>number</td>
</tr>
<tr>
<td>2</td>
<td>unary-operator primary</td>
<td>Any</td>
<td>number</td>
</tr>
<tr>
<td>3</td>
<td>primary binary-operator primary</td>
<td>Any</td>
<td>number</td>
</tr>
<tr>
<td>4</td>
<td>primary b1 primary b2 primary</td>
<td>b1 &gt;= b2</td>
<td>number b2 number (case 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b2 &gt;&gt; b1</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td>5</td>
<td>primary b1 (primary b2 primary)</td>
<td>b1 &gt;&gt; b2</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b2 &gt;&gt; b1</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td>6</td>
<td>u1 primary b1 primary</td>
<td>u1 &gt;&gt; b1</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b1 &gt;&gt; u1</td>
<td>u1 number (case 2)</td>
</tr>
<tr>
<td>7</td>
<td>primary b1 u1 primary</td>
<td>u1 &gt;&gt; b1</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b1 &gt;&gt; u1</td>
<td>ERROR (See case 8)</td>
</tr>
<tr>
<td>8</td>
<td>primary b1 (u1 primary)</td>
<td>u1 &gt;&gt; b1</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b1 &gt;&gt; u1</td>
<td>number b1 number (case 3)</td>
</tr>
<tr>
<td>9</td>
<td>u1 u2 primary</td>
<td>u2 &gt;= u1</td>
<td>u1 number (case 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>u1 &gt;&gt; u2</td>
<td>ERROR (See case 10)</td>
</tr>
<tr>
<td>10</td>
<td>u1 (u2 primary)</td>
<td>u2 &gt;&gt; u1</td>
<td>u1 number (case 2)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>u1 &gt;&gt; u2</td>
<td>u1 number (case 2)</td>
</tr>
</tbody>
</table>

**Table 3-7. Conditions and Notations for Examples**

**Conditions**

- All memory locations are accessible (none are GUARDED).
- SUFFIX = T (implicit radix is decimal).
- PC = 1000H
- DEFINE .AA = 2000H
- DEFINE .BB = FFFFH
- BYTE 1000H = 3EH
- BYTE 2000H = 23H

**Notation**

- → evaluates to.
- >> has higher precedence than.
- >>>= has higher or equal precedence.
- u1, u2,... unary operators
- b1, b2,... binary operators
Case 3: EVALUATE primary binary-operator primary

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

Examples:

\[
\begin{align*}
\text{EVA} & \ 10 + 20 \\
& \quad 11110Y \ 36Q \ 30T \ 1EH \ " \\
\text{EVA} & \ .AA > 10 \\
& \quad 1111111111111Y \ 177777Q \ 65535T \ FFFFH \ " \\
\text{EVA} & \ .AA \ OR \ PC \\
& \quad 1100000000000Y \ 30000Q \ 12288T \ 3000H "0"
\end{align*}
\]

Case 4: EVALUATE primary b1 primary b2 primary

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

A. b1 >= b2

Examples:

\[
\begin{align*}
\text{EVA} & \ 10 + .AA - PC \\
& \quad 1000000001010Y \ 10012Q \ 4106T \ 100AH " \\
\text{EVA} & \ 10 * .AA - PC \\
& \quad 11000000000000Y \ 30000Q \ 12288T \ 3000H "0" \\
\text{EVA} & \ PC = .AA \ OR \ .BB \\
& \quad 11111111111111Y \ 177777Q \ 65535T \ FFFFH " \\
\text{EVA} & \ 1 + 2 - 3 \\
& \quad 0Y \ 0Q \ 0T \ 0H " \\
\text{EVA} & \ 3 * 2 + 1 \\
& \quad 111Y \ 7Q \ 7T \ 7H "
\end{align*}
\]

B. b2 >> b1

Examples:

\[
\begin{align*}
\text{EVA} & \ 2 + 3 * 4 \\
& \quad 1110Y \ 16Q \ 14T \ EH " \\
\text{EVA} & \ .BB \ OR \ .AA \ AND \ PC \\
& \quad 11111111111111Y \ 177777Q \ 65535T \ FFFFH " \\
\text{EVA} & \ 1 \ OR \ 2 \ AND \ 3 \\
& \quad 11Y \ 3Q \ 3T \ 3H "
\end{align*}
\]
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</td>
<td>.BB OR PC = .AA AND AFAFH XOR .AA MOD 277</td>
</tr>
<tr>
<td>1</td>
<td>Lookup</td>
<td>FFFFH OR 1000H = 2000H AND AFAFH XOR 200H MOD 277</td>
</tr>
<tr>
<td>2</td>
<td>MOD</td>
<td>FFFFH OR 1000H = 200H AND AFAFH XOR 9FH</td>
</tr>
<tr>
<td>3</td>
<td>=</td>
<td>FFFFH OR 0 AND AFAFH XOR 9FH</td>
</tr>
<tr>
<td>4</td>
<td>AND</td>
<td>FFFFH OR 0 XOR 9FH</td>
</tr>
<tr>
<td>5</td>
<td>OR</td>
<td>FFFFH XOR 9FH</td>
</tr>
<tr>
<td>6</td>
<td>XOR</td>
<td>FF60H</td>
</tr>
</tbody>
</table>

More examples:

EVA 2 XOR 32 MASK 41 MOD 33
10Y 2Q 2T 2H ""

EVA 2 * 3 + 5 / 3 / 4 + 7
1101Y 15Q 13T DH ""

EVA 2 + 3 * 5 + 7
11000Y 30Q 24T 18H ""

Case 5: EVALUATE primary b1 (primary b2 primary)

Binary operator b2 is evaluated first, even if it has lower precedence than b1. Use parentheses when b2 must be evaluated before b1.

Examples:

EVA 2 * (3 + 5)
10000Y 20Q 16T 10H ""

EVA .BB / (.AA MASK AFAFH)
111Y 7Q 7T 7H ""

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

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

Precedence decides which operator is evaluated first.

A. ul \texttt{>>} b1

Examples:

\begin{verbatim}
EVA \texttt{-10 + 22}
1100Y 14Q 12T CH ''
\end{verbatim}

\begin{verbatim}
EVA \texttt{BYTE .AA OR .BB}
1111111111111111Y 177777Q 65535T FFFFH ''
\end{verbatim}

\begin{verbatim}
EVA \texttt{NOT .BB AND AFAFH}
0Y 0Q 0T 0H ''
\end{verbatim}

B. b1 \texttt{>>} ul

Examples:

\begin{verbatim}
EVA \texttt{BYTE .AA - 1000H}
111110Y 76Q 62T 3EH ''
\end{verbatim}

\begin{verbatim}
EVA \texttt{NOT .BB /23}
1111010011011110Y 172336Q 62686T F4DEH 'Tt'
\end{verbatim}

Case 7: EVALUATE \textit{primary} b1 ul \textit{primary}

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

A. ul \texttt{>>} b1 is valid.

Examples:

\begin{verbatim}
EVA \texttt{10 * -2}
1111111111111111Y 177754Q 65516T FFECH 'L'
\end{verbatim}

\begin{verbatim}
EVA \texttt{.AA AND NOT .BB}
0Y 0Q 0T 0H ''
\end{verbatim}

B. b1 \texttt{>>} ul produces an error. Operator b1 must be evaluated next, and requires two operands, but \textit{ul primary} has not yet been evaluated to a numeric result.

Examples:

\begin{verbatim}
EVA \texttt{10 + BYTE .AA}
HOST/EN1 ERR 80: SYNTAX ERROR
\end{verbatim}

\begin{verbatim}
EVA \texttt{.AA MASK NOT .BB}
HOST/EN1 ERR 80: SYNTAX ERROR
\end{verbatim}
Case 8: EVALUATE primary b1 (ul primary)

Unary operator ul is evaluated first, even if it has lower precedence than binary operator b1. Parentheses must be used when ul has lower precedence than b1.

Examples:

EVA 10 + (BYTE .AA)
101101Y 55Q 45T 2DH ‘-’

EVA .AA MASK (NOT .BB)
0Y 0Q 0T 0H ‘’

Case 9: EVALUATE ul u2 primary

Unary operator u2 must have higher precedence than ul to evaluate without an error.

A. u2 >> ul is valid

Examples:

EVA BYTE –EFFFH
10000000Y 200Q 128T 80H ‘’

EVA NOT BYTE .AA
11111111011100Y 177734Q 65500T FFDCH ‘’

B. ul >>= u2 results in an error.

Examples:

EVA BYTE NOT .AA
HOST/EN1 ERR 80: SYNTAX ERROR

EVA– BYTE .AA
HOST/EN1 ERR 80: SYNTAX ERROR

EVA BYTE BYTE 1000H
HOST/EN1 ERR 80: SYNTAX ERROR

EVA – – 5
HOST/EN1 ERR 80: SYNTAX ERROR

Case 10: EVALUATE ul (u2 primary)

Unary operator u2 is evaluated first, even if it has lower precedence than ul. Parentheses must be used when u2 has lower precedence than ul.

Examples:

EVA BYTE (NOT .AA)
111101Y 75Q 61T 3DH ‘=’

EVA – (BYTE .AA)
111111111011101Y 177735Q 65501T FFDCH ‘]’

EVA BYTE (BYTE 1000H)
11111110Y 376Q 254T FEH ‘‘

EVA – (– 5)
101Y 5Q 5T 5H ‘’

3-15
Two other "cases" can be diagrammed as:

\[
\text{primary } b_1 b_2 \text{primary} \\
\text{primary } u_1 b_2 \text{primary}
\]

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 numbers as results. The interpretation or use of the result depends on 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 integer value; all bits are important. In the standard ICE manuals, all expressions are numeric.

The term *boolean-expression* means an expression in a boolean command context. 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. In this manual, the command keywords IF, ORIF, WHILE, UNTIL, and BOOL establish boolean contexts; all other commands establish numeric contexts.

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:

**NOT:**

\[
\text{NOT FALSE } \rightarrow \text{ TRUE} \\
\text{NOT TRUE } \rightarrow \text{ FALSE}
\]

**AND:**

\[
\text{TRUE AND TRUE } \rightarrow \text{ TRUE} \\
\text{TRUE AND FALSE } \rightarrow \text{ FALSE} \\
\text{FALSE AND TRUE } \rightarrow \text{ FALSE} \\
\text{FALSE AND FALSE } \rightarrow \text{ FALSE}
\]

**OR:**

\[
\text{TRUE OR TRUE } \rightarrow \text{ TRUE} \\
\text{TRUE OR FALSE } \rightarrow \text{ TRUE} \\
\text{FALSE OR TRUE } \rightarrow \text{ TRUE} \\
\text{FALSE OR FALSE } \rightarrow \text{ FALSE}
\]

**XOR:**

\[
\text{TRUE XOR TRUE } \rightarrow \text{ FALSE} \\
\text{TRUE XOR FALSE } \rightarrow \text{ TRUE} \\
\text{FALSE XOR TRUE } \rightarrow \text{ TRUE} \\
\text{FALSE XOR FALSE } \rightarrow \text{ FALSE}
\]
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 3-8 for reference.

**Table 3-8. Command Contexts**

<table>
<thead>
<tr>
<th>Type of Entry</th>
<th>Contexts</th>
<th>Interpretation</th>
<th>Limitations</th>
<th>Example of Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Numeric expression</td>
<td>Set and change commands, etc.</td>
<td>16-bit unsigned number; may be forced to fit destination.</td>
<td>All primaries and operators allowed. Numeric constants without suffix is interpreted in current default radix.</td>
<td>PC = .AA * 256T + 10FFH</td>
</tr>
<tr>
<td>Boolean expression</td>
<td>BOOL, WHILE, UNTIL, IF, OR IF</td>
<td>LSB = 0 → FALSE LSB = 1 → TRUE</td>
<td>All primaries and operators allowed. Numeric constants without suffix is interpreted in current default radix.</td>
<td>WHILE PC &lt; .BB + .OFFSET</td>
</tr>
<tr>
<td>Count</td>
<td>COUNT command, STEP COUNT, clock rates</td>
<td>16-bit positive number</td>
<td>All primaries and operators allowed. All constants without suffix are decimal.</td>
<td>COUNT 10</td>
</tr>
<tr>
<td>Address</td>
<td>FROM, context-operator, partition, MAP, SAVE, DUMP</td>
<td>16-bit (or lower) address in memory or I/O.</td>
<td>Only arithmetic operators are allowed outside parentheses. Constants without suffix are interpreted in current default radix.</td>
<td>GO FROM .CC + .OFFSET</td>
</tr>
<tr>
<td>Decimal number</td>
<td>statement-number, MOVE, PRINT</td>
<td>positive number</td>
<td>No operators are allowed outside parentheses. All constants without suffix are decimal.</td>
<td>PRINT 10</td>
</tr>
<tr>
<td>Decimal constant</td>
<td>channel-number (ICE-85)</td>
<td>Positive decimal constant.</td>
<td>Single constant, no suffix allowed.</td>
<td>DEFINE GROUP X = 1, 9, 17</td>
</tr>
</tbody>
</table>
The ICE-independent commands described in this chapter can be executed by the HOST in either environment (EN1 or EN2) and by either ICE process (PR1 or PR2).

The commands are as follows:

**Loop and Branch Commands**

- REPEAT command
- COUNT command
- IF command

**Macro Commands**

- DEFINE MACRO command
- Call macro command
- Display macro command
- Display macro directory command
- REMOVE MACRO command
- PUT macro command
- INCLUDE command

**Display Commands**

- BOOL command
- WRITE command
- Multiple displays
- Symbolic displays
- IND symbol commands
- Define IND symbol command
- Remove IND symbol command
- Display IND symbol commands

**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, IF, and DEFINE MACRO commands. Two other compound commands, ACTIVATE and LOCK, apply to multiple-ICE operations and are discussed in later chapters. As the command titles indicate, REPEAT and COUNT are looping commands, IF establishes conditional execution, and DEFINE MACRO establishes a named command block.

**Local and Global Defaults**

Several of the system defaults can have "local" settings within a compound command; these defaults are 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>
<tr>
<td>SWITCH</td>
<td>Default ICE environment to be used for any reference without an explicit environment operator.</td>
</tr>
<tr>
<td>LOCK</td>
<td>The process that currently is executing a LOCK command block.</td>
</tr>
</tbody>
</table>
When a compound command executes, the current "global" settings of SUFFIX, SWITCH, and LOCK are saved so that they can be restored after the command finishes executing. Each of these global defaults continues in effect within the block unless and until a new (local) default is set with a SUFFIX, SWITCH, or LOCK command in the compound command. Defaults other than these are changed globally when they are set within a block.

When the command block finishes executing, the previous SUFFIX, SWITCH and LOCK defaults are restored. Thus, any of these three defaults that is set within a block has no effect after that block has terminated.

Here is an (artificial) example of a macro block with a local default:

```
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.

**REPEAT Command**

```
REPEAT cr
  [command cr]
  [WHILE boolean-expression cr]
  [UNTIL boolean-expression cr] ...
ENDREPEAT
```

Examples:

```
REPEAT
  GO FROM .START TILL .BR0
ENDREPEAT

REPEAT
  WHILE .VAR < .TOTAL
  STEP .VAR 1
  PRINT -1
ENDR

REPEAT
  .COUNTER = .COUNTER + 1
  WRITE 'COUNTER = ', BYTE .COUNTER
  UNTIL .COUNTER > .MAXIMUM
END
```

- **REPEAT** Command keyword identifying the beginning of the command block to be repeated.
- **cr** Intermediate carriage return.
- **command** Any ICE command, simple or compound, except DEFINE MACRO.
WHILE Command keyword introducing a "while true" clause.

\textit{boolean-expression} An expression that is TRUE when the least significant bit (LSB) = 1 and FALSE when the LSB = 0.

UNTIL Command keyword introducing an "until true" clause.

ENDREPEAT Command keyword identifying the end of the command block.

Discussion

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 \textit{cr} in the command syntax).

After each intermediate carriage return, ICE 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. If no elements are entered, the REPEAT is a "null" command.

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 loading up the values of any references in the expression. Thus, the result can change with each evaluation. Refer to Chapter 3 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-exp" is equivalent to "UNTIL NOT (bool-exp)".
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 when no ICE process is ACTIVE.

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
PC = .START
REPEAT
    UNTIL PC = 1000H
    STEP COUNT 1
ENDR
```

In this command the condition PC = 1000H is tested after every STEP. If the sequence of STEPs reaches PC = 1000H as the next instruction, the loop will terminate. By contrast, consider this example:

```plaintext
PC = .START
REPEAT
    UNTIL PC = 1000H
    STEP COUNT 10
ENDR
```

In the second example, the condition PC = 1000H is tested after every ten STEPs. The loop exits only if PC = 1000H occurs at the end of some group of ten instructions. If PC = 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
PC = 0
REPEAT
    UNTIL PC > 10H
    STEP COUNT 10
    PRINT -10
    WHILE .ZZ = 0
       .ZZ = .ZZ + 1
ENDR
```

Assume for this example that the code being emulated (with STEP) contains only one-byte instructions. Then, after the first time through the loop, PC = 0AH (10T) and .ZZ = 1. On the second iteration, the test PC > 10H is FALSE when it is encountered, so the STEP and PRINT commands are executed again. At this point, PC > 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.
Here are some brief examples of the REPEAT command.

Example 1: Generate an ASCII table similar to Table 3-2.

```
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
PC = .START
REPEAT
  UNTIL PC = .DELAY
  STEP COUNT 1
  PRINT -1
ENDR
```

Example 3: Using a complex combination of conditions in the boolean expression. Note that a combination like this cannot appear in a TILL clause after STEP, since parentheses and the XOR operator are not allowed with TILL.

```
REPEAT
  UNTIL (PC > .END XOR BYTE .VAR1 = 0) AND (.TEMP > 0 XOR .VAR2 = 1)
  STEP COUNT 1
  REGISTERS
ENDR
```

Example 4: Emulate from the start of the program (.START) until a breakpoint (LOCATION 1000H EXECUTED) is reached, display status registers, then continue emulating, halting, and displaying status until a terminating condition (BYTE .VAR = 2) is reached.

```
PC = .START
REPEAT
  GO TILL LOCATION 1000H EXECUTED
  REGISTERS
  UNTIL BYTE .VAR = 2
ENDR
```

COUNT Command

```
COUNT count cr
  command cr
  [WHILE boolean-expression cr ] ...
  [UNTIL boolean-expression cr ]
ENDCOUNT
```
Examples:

```
COUNT 10
  GO FROM.START TILL BR0
ENDCOUNT

COUNT .TESTTIMES + 50T
  WHILE .VAR < .TOTAL
  STEP COUNT 1
  PRINT -1
ENDC

COUNT BYTE.COUNTER
  WRITE 'COUNTER = ' BYTE.COUNTER
END
```

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>Command keyword marking the beginning of a bounded loop.</td>
</tr>
<tr>
<td>count</td>
<td>A number (implicitly decimal if entered without radix) or expression specifying the maximum number of iterations of the commands in the block.</td>
</tr>
<tr>
<td>cr</td>
<td>Intermediate carriage return.</td>
</tr>
<tr>
<td>command</td>
<td>Any simple or compound ICE command, except DEFINE MACRO.</td>
</tr>
<tr>
<td>WHILE</td>
<td>Command keyword introducing a &quot;while true&quot; exit condition.</td>
</tr>
<tr>
<td>boolean-expression</td>
<td>A number or expression that evaluates to TRUE if its least significant bit (LSB) = 1, and to FALSE if its LSB = 0.</td>
</tr>
<tr>
<td>UNTIL</td>
<td>Command keyword introducing an &quot;until true&quot; exit condition.</td>
</tr>
<tr>
<td>ENDCOUNT</td>
<td>Command keyword marking the end of the command block in the loop. May be abbreviated to END.</td>
</tr>
</tbody>
</table>

Discussion

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 after COUNT controls the (maximum) number of iterations to be performed. If a numeric constant is used (for example, COUNT 10), ICE 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 4-1 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
PC = .START
COUNT 10
  UNTIL PC = .DELAY
  STEP COUNT 1
  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 or COUNT command that includes one or more exit-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 4-1 as a display (or LIST file output) you could use the following sequence.

```
BASE = T
DEFINE .ITER = 0
DEFINE .XX = 2
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 BOOL display command is discussed later in this chapter.

The following example emulates to a breakpoint, displays status registers, then continues emulating, breaking, and displaying status for a definite number of iterations:

```
PC = .START
COUNT 10
 GO TILL LOCATION 1000H EXECUTED
 REGISTERS
END
```

**IF Command**

```
IF boolean-expression [THEN] cr
 [command cr] ...
 ORIF boolean-expression [THEN] cr ] ...
 [command cr] ...
 ELSE cr
 [command cr] ...
ENDIF
```

Examples:

```
IF PC > 1000H THEN
 GO TILL BR0
ENDIF
IF PC >= 0 AND PC < 1000H
 GO TILL BR0
ENDIF
IF PC >= 0 AND PC < 1000H
 GO TILL BR0
 ORIF PC >= 1000 AND PC < 2000H
 GO TILL BR1
 ELSE
 GO TILL SY0
END
```
Multi-ICE

IF

Command keyword marking the beginning of the first block of commands in the conditional command.

boolean-expression

A number or expression that evaluates to TRUE when its least-significant bit (LSR) is 1, or to FALSE when its LSB is 0.

THEN

Optional command keyword.

cr

Intermediate carriage return.

command

Any simple or compound ICE command except DEFINE MACRO. The block of commands in the first IF or ORIF clause with a TRUE boolean expression are executed.

ORIF

Command keyword marking the beginning of an additional block of commands to be executed when the ORIF condition is TRUE.

ELSE

Command keyword marking the beginning of a block of commands to be executed when none of the IF or ORIF conditions is TRUE.

ENDIF

Command keyword marking the end of the IF command. May be abbreviated to END.

Discussion

The IF command permits conditional execution in a command sequence. 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 ENDIF keyword is required to close off the IF command; it can be abbreviated to END.

Here is an example of the IF command.

BASE = T
PC = 1
IF PC < 1
   EVALUATE 1
ORIF PC < 2
   EVALUATE 2
ORIF PC < 3
   EVALUATE 3
ELSE
   EVALUATE 4
END
This example displays the result of EVALUATE 2 and then terminates. The first condition (IF PC < 1) is FALSE, so EVALUATE 1 is skipped. The second condition (ORIF PC < 2) is TRUE, so EVALUATE 2 is executed and the IF command terminates. The third condition (ORIF PC < 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:

\[
\begin{align*}
\text{BOOL PC} & < 1 \quad \text{(Displays FALSE)} \\
\text{BOOL PC} & < 2 \quad \text{(Displays TRUE)} \\
\text{EVALUATE 2}
\end{align*}
\]

**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 ENDIF, to help you keep straight which command you intend to close off. ICE 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 several times during program execution. One way to achieve this effect (in ICE-85) is with the following command sequence:

\[
\begin{align*}
\text{TRACE} & = \text{INSTRUCTION} \\
\text{PC} & = \text{.START} \\
\text{REPEAT} \quad & \text{IF PC} = \text{.DELAY} \\
& \quad \text{PC} = \text{WORD SP} \\
& \quad \text{SP} = \text{SP + 2} \\
& \quad \text{ENDIF} \\
& \quad \text{STEP COUNT 1} \\
& \quad \text{PRINT -1} \\
\text{ENDR}
\end{align*}
\]

At each CALL to .DELAY in the program, the return address for the call is pushed on the stack. The keyword SP refers to the stack pointer, the address of the top of the stack where the return address is stored. The effects of the commands PC = WORD SP and SP = SP + 2 are to load the return address back into PC and reset the stack pointer just as if the RET (return) instruction at the end of .DELAY had been executed. (This command sequence for “popping” the stack works with ICE-85 only; a sample sequence to accomplish the same thing for ICE-49 is given as a macro later in this chapter.)
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):

```
GO FROM .START TILL #21 EXECUTED
REPEAT
  IF BYTE .MARK > 0
    WORD .PTR = (WORD .PTR) + 2
  ENDIF
  GO FROM #23 TILL #21 EXECUTED
ENDR
```

As a last example of nesting, the following sequence keeps track of the procedure level in the user code, displays any CALL or RET (return) instruction, and terminates when the outermost procedure returns.

```
PC = .START
SR = COUNT 1
DEFINE .LEVEL = 1
REPEAT
  STEP
    IF OPCODE = 0CDH OR OPCODE MASK 3070 = 0C4H ;CALL
      .LEVEL = .LEVEL + 1
      PRINT -1 ;display level
    ORIF OPCODE = 0C9H OR OPCODE MASK 3070 = 0C0H ;RETURN
      .LEVEL = .LEVEL -1
      PRINT -1 ;display level
    ENDIF
  WHILE .LEVEL >= 1
ENDR
```

Macro Commands

A macro is a named 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.
- Bring one or more macro definitions (or other commands) in from a file for use in the current test sequence.

The syntax summaries of the macro commands are presented as a group, followed by a discussion with examples.
DEFINE MACRO Command

```
DEFINE MACRO macro-name cr
   [command cr]  ...
EM
```

Examples:

```
DEFINE MACRO LOOK
   WRITE '.VAR1 = ', BYTE .VAR1
   WRITE '.VAR2 = ', BYTE .VAR2
EM

DEF MAC EMUL
   GO FROM .START TILL .BR0
   PRINT -10
   REPEAT
      STEP COUNT %0
      PRINT - %0
   UNTIL PC = %1
   ENDR
EM
```

- **DEFINE MACRO** Command keywords for macro definition block.
- **macro-name** The user-assigned name of the macro being defined. May not duplicate an existing macro-name.
- **cr** Intermediate carriage return.
- **command** Any simple or compound ICE command except DEFINE MACRO and REMOVE MACRO.
- **EM** Command keywords marking the end of the definition block.

Macro Call Command

```
:macro-name  [actual-parameter-list]
```

Examples:

```
:LOOK
:EMUL 10, .END
```

- **:** Command token for macro call.
- **macro-name** The name of the macro to be expanded and executed.
- **actual-parameter-list** A list of parameter values or strings to be substituted for formal parameters (%0 .. %9) in the macro definition to form the expanded (executable) macro.
Display Macro Definition Command

MACRO \textit{[macro-name]}

Examples:

MACRO LOOK
MACRO EMUL

MACRO \quad \text{Command keyword for displaying the definition block of the macro named in the command.}

\textit{macro-name} \quad \text{The name of the macro to be displayed. If \textit{macro-name} is omitted, all macro definitions are displayed.}

Display Macro Directory Command

DIRECTORY MACRO

Examples:

DIRECTORY MACRO
DIR MAC

DIRECTORY MACRO \quad \text{Command keywords requesting display of the names of all macros correctly defined, in the order they were defined.}

REMOVE MACRO Command

REMOVE MACRO \ [\textit{macro-list}]

Examples:

REMOVE MACRO
REM MAC LOOK
REM MACRO LOOK, EMUL

REMOVE MACRO \quad \text{Command keywords to remove (delete) one, several, or all macro names and definitions.}

\textit{macro-list} \quad \text{The names of one or more macros separated (if more than one) by commas. Macros in the list are deleted; if \textit{macro-list} is omitted, all macros are removed.}
**PUT Macro On File Command**

```
PUT :drive:filename MACRO  [macro-list]
```

Examples:

```
PUT :F1:UTIL.MAC MACRO
PUT :F1:INIT1.INC MACRO LOOK, EMUL
```

**PUT..MACRO**

Command keywords to open named file for input and copy one, several, or all macro definitions to that file.

**:drive:filename**

The diskette drive number and user filename for the file that is to receive the macro definitions. If `filename` already exists on the given `drive`, the previous content of that file are overwritten (lost) when the current macros are put in the file.

**macro-list**

One or more macro names, separated by commas if more than one. If `macro-list` is omitted, all macros currently defined are put in the file.

**INCLUDE From File Command**

```
INCLUDE :drive:filename
```

Examples:

```
INCLUDE :F1:UTIL.MAC
INCLUDE :F1:INIT.INC
```

**INCLUDE**

Command keywords to read commands (especially but not limited to macro definitions) from the named file, up to the end-of-file.

**:drive:filename**

The diskette drive number and filename of the file that contains the commands to be included.

**Defining and Invoking Macros**

Each macro is defined once in the test session.

Once it is defined, you can invoke (call) a macro as often as desired.

The macro definition command causes the macro name and the 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).
If you have a file on the ICE diskette named MAC.TMP it will be lost when any macros are defined during the test session.

A macro-name must begin with an alphabet letter, or with one of the characters "?" 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, ACTIVATE, LOCK, or another macro definition). The command block in the macro definition can include any command except another DEFINE MACRO 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 commands that define user symbols and other identifiers such as channel group names. Macros that include user definitions can be used for initialization purposes. For example, suppose you have the eighteen ICE-85 user probes attached to a 40-pin clip that is to be moved from one chip to another during the test session. You can define a macro for each pin configuration; each time you move the clip, you call the appropriate macro to define groups for trace display. Each such macro begins with the REMOVE GROUP command to avoid an error. The following macros serve as illustrations:

```
DEF MAC CHIP1
  REMOVE GROUP
  DEF GROUP INPUT = 8,7,6,5,4,3,2,1 IN Y
  DEF GROUP OUTPUT = 16,15,14,13,12,11,10 IN Y
EM

DEF MAC CHIP2
  REMOVE GROUP
  DEF GROUP INPUT = 16,15,6,5,4,10,9,8 IN Y
  DEF GROUP OUTPUT = 14,13,1,3,2,11,7,12 IN Y
EM
```

The fact that the groups are defined in each macro makes it possible to use the macros in any order.

You should use some caution in placing user definitions within macro definitions, however, since multiple definitions cause errors.
A macro definition can include calls to other macros, but a macro cannot call itself recursively. A macro that calls itself in its command block expands indefinitely when the outer macro is called, without ever executing any commands (press ESC to terminate such an infinite expansion). 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 defaults (SUFFIX, SWITCH, LOCK) are saved in case new defaults are 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 defaults are restored.
- The macro exits. Control returns to the console (asterisk prompt).

The next several sections provide details on these operations; system defaults are discussed earlier in this chapter.

**Formal and Actual Parameters**

A formal parameter marks a place in a macro definition 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:

```
%0
```

where \( n \) 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 parameters (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, ICE supplies nulls for the extra formal parameters. If too many actual parameters are entered, the extra actual parameters are ignored. However, if more than ten actual parameters are entered, an error occurs and the call is aborted.
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
   %0BYTE %1
EM
```

In the call to this macro, parameter %0 can become "C", "D", or "X" (under ICE-48), or "I" or null (under ICE-85). Parameter %1 can be any valid address or partition. Examples of calls to this macro:

*ICE-48*

<table>
<thead>
<tr>
<th>Macro call</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>:MEM X,20H</td>
<td>XBYTE 20H</td>
</tr>
<tr>
<td>:MEM D,20H LEN 5H</td>
<td>DBYTE 20H LEN 5H</td>
</tr>
</tbody>
</table>

*ICE-85*

<table>
<thead>
<tr>
<th>Macro call</th>
<th>Expansion</th>
</tr>
</thead>
<tbody>
<tr>
<td>:MEM I, 1000H</td>
<td>IBYTE 1000H</td>
</tr>
<tr>
<td>:MEM, 1000H TO 100FH</td>
<td>BYTE 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).
Example 3:

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 (ICE-85):

:BRS 0, LOCATION 1000H EXECUTED

This parameter is valid, but this one:

:BRS 0, FFH, 101Y ON ADDR, STS

results in the expansion:

BR0 = FFH

To obtain the correct expansion, make the parameter a string:

:BRS 0, 'FFH, 101Y ON ADDR, STS'

This results in the expansion:

BR0 = FFH, 101Y ON ADDR, STS

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, and 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 macro 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 command adds the macro defined to the end of the table.

The REMOVE MACRO command removes one or more macro definitions from the table.
If the list of macro-names is omitted, all macros are removed. The REMOVE MACRO 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. 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.

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 (command)
   CHIP1
   CHIP2
   MEM
   RPT
   BRS (display)
```

**Example 2:**

```
*MAC GOER (command)
   DEF MAC GOER (display)
   REPEAT
      GO FROM .START TILL BR0
   END
   EM
```

**Example 3:**

```
*REM MAC BRS (command)
```

**Example 4:**

```
*DIR MAC (command)
   GOER
   CHIP1
   CHIP2
   MEM
   RPT
```

**Example 5:**

```
*DEF MAC NULL (command)
*EM
```

**Example 6:**

```
*DIR MAC (command)
   GOER (display)
   CHIP1
   CHIP2
   MEM
   RPT
   NULL
```
Saving Macros

The PUT MACRO command causes one or more macro definitions to be copied from the temporary file to a “permanent” file.

If any macro names are entered, those macro definitions are saved. If no list of macro names is given, all macros in the macro table are saved. The definitions in the temporary file MAC.TMP containing 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 below.

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.

INCLUDING Commands From Files

The macro definition files created with the PUT command can be read into the ICE temporary macro table with the INCLUDE command. For example, suppose we had defined a macro INIT as follows:

```
*DEFINE MACRO INIT
*MAP 3000 LEN 4K = USER
*LOAD :F1:PROG1
*EM
```

Suppose further that we had saved this macro definition with a PUT command as follows:

```
*PUT :F1:INIT.INC MACRO INIT
```

Then, in a subsequent session we bring this macro definition from the file with the command:

```
*INCLUDE :F1:INIT.INC
```

The result of the INCLUDE is to read in the definition of INIT as given earlier. ICE issues a prompt at the beginning of each input line (i.e., the prompts are not saved on the file).

Although the PUT command can be used only to save macro definition, the INCLUDE command can refer to a file containing any valid ICE commands. For example, the file could contain macro invocations as well as definitions. To have an INCLUDE file contain commands other than macro definitions, edit the file with the ISIS text editor. Each command that you “edit in” should start at the left margin (do not “edit in” the prompt), and should terminate with a carriage return.

The INCLUDE command can be nested within any other command, including a macro definition.
Further Examples

Here are a few more examples of macros. These macros simulate stack operations, calls, and returns in ICE-85 and ICE-49.

A stack is an area of memory, indexed (addressed) by a register called the stack pointer (SP). The stack is used to save status information required for an orderly return from a procedure call.

ICE-85 Macro Examples

In ICE-85, 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 (byte) at the top of the stack; this address contains the last item pushed on the stack. As each new byte 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 (SP - 1), and the low byte is stored in the next lower address (equivalent to SP - 2).

The MCS-85 assembly language PUSH rp instruction sets SP to the next available pair of bytes, then stores the content of the given register pair in adjacent addresses at that position. We can simulate this action with a macro, as follows:

```assembly
DEF MAC PUSH85
    SP = SP - 2T ; decrement SP.
    WORD SP = %0 ; low byte in low address, high byte in high address.
EM
```

The formal parameter %0 lets us use PUSH85 to save any register pair or other 16-bit value; for examples:

- :PUSH85 PC ; save program counter.
- :PUSH85 RBC ; save register pair BC.
- :PUSH85 RDE ; save register pair DE
- :PUSH85 RHL ; save register pair HL
- :PUSH85 PSW ; save RA and RF

**NOTE**

The keyword PSW was inadvertently omitted from the ICE-85 Operator’s Manual. Please refer to figure 4-1 for a description of the MCS-85 Push PSW instruction.

The complementary MCS-85 POP rp instruction copies the contents of the two top bytes back into the given register pair, then increments SP to the new top of the stack. A macro for this function is:

```assembly
DEF MAC POP85
    %0 = WORD SP
    SP = SP + 2T
EM
```

Here are some calls to POP85, corresponding to the PUSH85 calls given earlier:

- :POP85 RHL
- :POP85 RDE
- :POP85 RBC
- :POP85 PC
- :POP85 PSW
Here are some macros that use PUSH85 and POP85.

1. Macro to "call" a procedure:
   ```
   DEF MAC CALL85
   :PUSH85 PC
   GO FROM %0
   EM
   
   This macro can be invoked with or without a halt condition:
   :CALL85 .PROC
   :CALL85 .PROC TILL BR0
   ```

2. Macro to "return" from a procedure:
   ```
   DEF MAC RET85
   :POP85 PC
   GO %0
   EM
   
   To invoke this macro:
   :RET85
   :RET85 TILL BR0
   ```

3. Macro to single-step through user code, skipping over a specified procedure whenever that procedure is called from the user program.
   ```
   DEF MACRO SKIP
   SR = COUNT 1
   REPEAT
   IF PC = %0
   :POP85 PC
   ENDF
   STEP
   PRINT -1
   ENDR
   EM
   ```
Suppose the user program contains a repetitive timer routine named DELAY that is called from several places in the program. The following macro invocation causes ICE to step through the program without emulating the timer routine:

:SKIP .DELAY

**ICE-49 Macro Examples**

In ICE-49, the stack is located in the on-chip DATA memory. Accessing the stack in ICE-4X is somewhat more involved than in ICE-85, due to the way stack data is formatted.

The stack is 16 bytes long, allowing up to eight levels of nesting. The stack pointer (SP) takes values from 1 to 7, corresponding to data memory locations as shown in table 4-2. Each SP value points to the next available pair of locations in data RAM; the pair consists of a low (even) byte and a high (odd) byte. We want to access each of these locations separately, given a current value of SP. The decimal location of the low (EVEN) byte equals \( SP \times 2 + 8 \), and the decimal location of the high (odd) byte equals \( SP \times 2 + 9 \), the low location plus one. For example, when \( SP \) equals 5, the locations are \( (5 \times 2 + 8 = 18) \) and \( (5 \times 2 + 9 = 19) \). We'll use this approach in our "push" and "pop" macros, but there are a few more details to consider.

The MCS-48 assembly language CALL instruction pushes the 12-bit program counter (PC) and bits 4 to 7 of the program status word (PSW) on the stack, using the format shown in figure 4-2. The full structure of the PSW is shown in figure 4-3.

The content of the high (odd) byte is bits 8 to 11 of the PC and bits 4 to 7 of the PSW. The following command achieves this bit assignment:

\[
\text{DBYTE } (SP \times 2 + 9) = (\text{PSW MASK F0H}) + (\text{PC/100H})
\]

<table>
<thead>
<tr>
<th>SP</th>
<th>Data Memory Locations (Decimal)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>8-9</td>
</tr>
<tr>
<td>1</td>
<td>10-11</td>
</tr>
<tr>
<td>2</td>
<td>12-13</td>
</tr>
<tr>
<td>3</td>
<td>14-15</td>
</tr>
<tr>
<td>4</td>
<td>16-17</td>
</tr>
<tr>
<td>5</td>
<td>18-19</td>
</tr>
<tr>
<td>6</td>
<td>20-21</td>
</tr>
<tr>
<td>7</td>
<td>22-23</td>
</tr>
</tbody>
</table>

The formula \((\text{PSW MASK F0H})\) produces a byte consisting of zeros in bits 0 to 3 and bits 4 to 7 of PSW unchanged. The formula \((\text{PC/100H})\) shifts the 12-bit PC to the right a total of eight bit positions; thus bits 8 to 11 of PC are shifted into bits 0 to 3 of the result. When these two partial results are added together, the overall result is as shown in figure 4-2.

The content of the low (even) byte is the low 8 bits of the PC. The following command performs this assignment:

\[
\text{DBYTE } (SP \times 2 + 8) = \text{PC}
\]

The DBYTE assignment takes the low eight bits of PC and ignores the rest.
We can now write a macro to handle the ICE-4X push operation, as follows:

```asm
DEF MAC PUSH4X
SUFFIX = H
DBYTE (SP*2+8) = PC
SP = SP + 1
EM
```

The complementary “pop” macro is simply the reverse of `PUSH4X`:

```asm
DEF MAC POPR4X
SUFFIX = H
SP = SP - 1
PSW = (PSW MASK 0F) + ((DBYTE SP*2+9) MASK F0)
PC = (DBYTE SP*2+8) + (((DBYTE SP*2+9) MASK 0F) * 100)
EM
```

To simulate the MCS-48 CALL instruction:

```asm
DEF MAC CALL4X
:PUSH4X
GO FROM %0
EM
```

Sample invocations:

```asm
:CALL4X .ADDR
:CALL4X .ADDR TILL BR0
```
The MCS-48 RETR instruction returns with the PSW restored. To obtain this effect:

```
DEF MACRO RETR4X
  :POPR4X
  GO %0
  EM

Sample invocations:
  :RETR4X
  :RETR4X TILL BR0
```

To simulate the MCS-48 RET instruction (return without restoring PSW):

```
DEF MAC RET4X
  SUFFIX = H
  SP = SP - 1
  PC = (DBYTE SP*2+8) + (((DBYTE SP*2+9) MASK 0F) * 100)
  GO %0
  EM

Sample invocations:
  :RET4X
  :RET4X TILL BR0
```

**BOOL Display Command**

```
BOOL boolean-expression
```

Examples:

```
BOOL PC > 1000H
BOOL ACTIVE PR1
BOOL PC > 1000 OR ACTIVE PR1
```

**Discussion**

The BOOL command is a display command, parallel to the EVALUATE command.

To display the result of any expression as boolean TRUE or FALSE, enter a command of the form:

```
BOOL expression
```

ICE evaluates the expression, and displays TRUE if the least significant bit (LSB) of the result is a 1 or displays FALSE if the LSB is a 0. Refer to Chapter 3 for details on how expressions are evaluated.
WRITE Command

```
WRITE [string] expression [BOOL boolean-expression] [...]
```

Examples:

- WRITE 'THE VALUE IS', .VALUE
- WRITE 'THE NEW VALUE IS', .VALUE * 3/2
- WRITE 'IT IS', BOOL ACTIVE PR1, 'THAT PR1 IS ACTIVE'

| WRITE Command keyword to display text or data on the Intellec console device. |
| string | A sequence of characters enclosed in single quotes. |
| expression | Any number or expression; the result after evaluation is displayed. |
| BOOL boolean-expression | The keyword BOOL followed by any number, or expression. The boolean value “TRUE” is displayed when the number or result has its least significant bit (LSB) equal to 1, and “FALSE” is displayed when the result has its LSB equal to 0. |

Discussion

The WRITE command displays one or more elements on the console. If a list device (printer file) is enabled, the elements are sent to the list file as well.

For example, if the content of the variable .TIME is 15, the following command:

```
WRITE BYTE .TIME ' SECONDS ELAPSED.'
```

produces the following display:

```
15 SECONDS ELAPSED
```

The WRITE command can display a string, a number, the result of evaluating a numeric expression, or the result of evaluating a boolean expression; or a combination of any of these kinds of elements.

All the elements following WRITE are displayed on one line if possible; if the next element doesn’t fit the remaining character space on the line, ICE inserts a carriage return/line feed. No spaces are inserted between elements on the same line; if you want spaces before or after a text message, put spaces in the string.

A string is displayed just as you enter it.

A numeric constant is displayed as entered, using the current BASE when the WRITE command is created.
Multi-ICE

ICE-Independent Commands

An expression is evaluated, and the result is displayed in the current BASE.

An element of the form "BOOL boolean-expression" produces a display of "TRUE" when the least-significant bit of the expression (after evaluation) is a 1, and "FALSE" when the LSB is a 0.

Multiple Display Commands

Keyword references and content expressions can be combined in one-line multiple display commands. The form of this command can be shown as:

\[
\text{display-reference} \ [\ldots]
\]

A display reference is either a keyword reference or a content expression. A keyword reference is the name of any processor register (such as PC), or an ICE status register (such as BUFFERSIZE). Emulation registers (such as BR0, CR0, or QRO) cannot be used in multiple displays. Symbolic references (such as .TEMP) are invalid in multiple displays (except in content-expressions).

A content expression consists of a content-operator (see table 3-5), followed by an expression that evaluates to a single valid address. A partition (using TO or LENGTH) cannot appear in a multiple display.

Here is an example of a multiple display; the values shown are arbitrary.

\[
\begin{align*}
\text{PC}, & \text{ SP}, \text{ BYTE .AA} \quad \text{(Command)} \\
\text{PC}=36C3H & \text{ SP}=5000H \text{ 1000H}=1FH \quad \text{(Display)}
\end{align*}
\]

ICE-Independent Symbol Table Commands

Under Multi-ICE, each of the two ICEs has its own symbol table. In addition, there is an independent symbol table that can be accessed by the HOST and either ICE using the keyword IND.

To define a symbol in the IND symbol table:

\[
\text{DEFINE IND .symbol-name} = \text{address/value}
\]

To refer to an IND symbol (that is, to obtain the address corresponding to that symbol):

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

To remove a symbol from the IND symbol table:

\[
\text{REMOVE IND .symbol-name}
\]

To display the IND symbol table:

\[
\text{SYMBOL IND}
\]

To remove the entire IND symbol table:

\[
\text{REMOVE SYMBOL IND}
\]

The IND symbol table forms one "module". New modules cannot be defined for the IND symbol table.
## Symbolic Display of Addresses

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENABLE SYMBOLIC</td>
<td>Command keywords to have addresses that are displayed by certain ICE commands appear as symbolic displays rather than the default radix.</td>
</tr>
<tr>
<td>DISABLE SYMBOLIC</td>
<td>Command keywords to have all addresses displayed by ICE commands appear in the default radix.</td>
</tr>
<tr>
<td>EVALUATE expression SYMBOLIC</td>
<td>After the EVALUATE command, the keyword SYMBOLIC causes the result to appear as a symbolic display rather than in the four radices.</td>
</tr>
</tbody>
</table>

### Discussion

When ENABLE SYMBOLIC is in effect (SYMBOLIC is initially ENABLED), addresses displayed by many types of commands are displayed as symbolic displays rather than in the default radix.

DISABLE SYMBOLIC restores all address displays to their default radixes.

Examples of commands affected by SYMBOLIC are GO and STEP (display of PC after emulation termination), and displays of memory contexts. Displays of symbol tables are not affected.

A symbolic display has the format:

```
symbolic-reference [+ numeric-constant H]
```

where `symbolic-reference` can be any of the following:

- `symbol-name`
- `# statement-number`
- `..module-name.symbol-name`
- `..module-name # statement-number`

The symbolic reference displayed is the earliest entry in the symbol (or statement-number) table whose corresponding value is CLOSEST TO BUT NOT GREATER THAN the address-value given. Only the table accessible to the current SWITCH are searched; the IND symbol table is not searched.

Where the display involves a sequence of addresses, the symbolic displays appear only where an address **exactly matches** a symbol value. Otherwise, the difference is displayed as a **numeric constant**, always in hexadecimal radix.
If no symbol has a value lower than or equal to the target address, the default radix is used.

When SYMBOLIC appears after EVALUATE, the result of evaluating to expression is shown as a symbolic display, or a single hexadecimal value if no symbol has a value lower than or equal to the target value.
The commands and keywords in this chapter extend the capability of the ICE-85 component of a multi-ICE system; they are valid with ICE-85 only. The commands and keywords are as follows:

Commands:

- SEARCH command
- DOMAIN command
- NESTING command
- LINES command
- MODULES command

Keywords

- FLAG
- LOWER and LIMIT

SEARCH Command

The SEARCH (SEA) command displays addresses that contain a target value. The command is valid only with ICE-85. The syntax is:

```
SEARCH [DOUBLE] partition [WITH MASK mask-value] FOR target-value
[WITH LIMIT limit]
```

ICE searches the `partition` looking for an address whose contents (when masked) match the `target-value`. The address and its content are displayed. If more than one address has matching contents, all such addresses and their contents are displayed, one per line of the display.

The examples in this section assume memory contents as follows. SUFFIX = H is also assumed (numbers in commands entered are hexadecimal without explicit radix); The explicit radix appears in the display as shown in the examples.

<table>
<thead>
<tr>
<th>Address</th>
<th>Content</th>
<th>Address</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>0100H</td>
<td>0H</td>
<td>0106H</td>
<td>0H</td>
</tr>
<tr>
<td>0101H</td>
<td>11H</td>
<td>0107H</td>
<td>11H</td>
</tr>
<tr>
<td>0102H</td>
<td>22H</td>
<td>0108H</td>
<td>02H</td>
</tr>
<tr>
<td>0103H</td>
<td>33H</td>
<td>0109H</td>
<td>FFH</td>
</tr>
<tr>
<td>0104H</td>
<td>44H</td>
<td>010AH</td>
<td>FFH</td>
</tr>
<tr>
<td>0105H</td>
<td>55H</td>
<td>0106H</td>
<td>0H</td>
</tr>
</tbody>
</table>

If DOUBLE is omitted (or if the optional keyword SINGLE is used), the search looks at single addresses, matching on byte (8-bit) contents.

```
SEARCH 100 TO 109 FOR 55
0105H=55H
```

```
SEA 100 TO 109 FOR 0
0100H=00H
0106H=00H
```

5-1
SEARCH DOUBLE looks at overlapping pairs of addresses, matching on a 16-bit target value. The high byte of the target value is compared to the content of the higher of the pair of addresses, and the low byte to the content of the lower of the pair of addresses.

```plaintext
SEARCH DOUBLE 100 TO 109 FOR 5544
0104H=5544

SEA DOU 100 TO 109 FOR 1100
0100H=1100H
0106H=1100H
```

SEARCH DOUBLE looks one address past the last address in the partition.

```plaintext
SEA DOU 100 TO 108 FOR FF02
0108H=FF02
```

The WITH MASK `mask-value` clause allows you to mask certain bits in the target value. The mask value is ANDed both with the target value and with the content of each address examined. Any “0” bits in the mask value thus represent “don’t-care” bits in the match. The display gives all addresses whose (masked) contents match the (masked) target value, and their (unmasked) contents.

```plaintext
SEA 100 TO 109 WITH MASK OF FOR 2
0102H=22H
0108H=02H

SEA 100 TO 109 WITH MASK OF FOR 22
0102H=22H
0108H=02H

SEA 100 TO 109 WITH MASK OF FOR F2
0102H=22H
0108H=02H

SEA DOU 100 TO 109 WITH MASK F0F0 FOR 1100
0100H=1100H
0106H=1100H

SEA DOU 100 TO 109 WITH MASK 1 FOR 1
0101H=2211H
0103H=4433H
0105H=0055H
0107H=0211H
0109H=FFFFH

SEA DOU 100 TO 109 WITH MASK 1 FOR F
0101H=2211H
0103H=4433H
0105H=0055H
0107H=0211H
0109H=FFFFH

SEA DOU 100 TO 109 WITH MASK 1 FOR 0
0100H=1100H
0102H=3322H
0104H=5544H
0106H=1100H
0108H=FF02H
```
The target value and mask value can be numeric constants as shown in the examples; they can also be any reference or expression.

**DOMAIN Commands**

The `DOMAIN` (DOM) command allows you to specify the module to use for looking up statement-number references that are entered without an explicit module name. (Module names and statement numbers are valid only ICE-85.) The syntax is:

```
DOM ..module-name
```

The `module-name` is entered with two leading periods. For example, to restrict the search for statement numbers to a module named CARS, use:

```
DOM ..CARS
```

The module named must exist in the table or an error results.

To reset the domain to the entire statement-number table, use the command:

```
RESET DOMAIN
```

In addition the domain is reset to the entire table when the module that is the current domain is removed.

The `DOMAIN` command is useful where two or more modules with identical or overlapping statement numbers have been loaded. Note that `LINK` combines different modules, each with its own set of statement numbers. The `DOMAIN` command allows you to enter statement number references for a selected module without repeating the module name on each reference; this facility is very useful when your attention is focused on debugging a particular module.

**NESTING Command**

If the user code contains any call and return instructions (including conditional calls and returns), you can display the calls that were in effect when emulation last halted by entering the display keyword `NESTING` (NES). The NES command is valid only in ICE-85.

The display shows the address of each call instruction for which a matching return has not yet been executed, and the address of each called procedure. The most recent call (that is, the most deeply nested call) is the top line of the display, and the earliest call (least nested but still active) is the lowest line of the display. As each return instruction is executed, the most recent call is removed from the display. If no calls were active at the last halt in emulation, nothing is displayed.

The format of each display line is:

```
address of call   CALL   address of procedure
```

**LINES Command**

Under ICE-85, the display command `LINES` (LIN) causes a display of the statement number table.
**MODULES Command**

Under ICE-85, the display command MODULES (MOD) causes a display of all module names with symbol tables currently loaded.

**FLAG Keyword**

The display keyword FLAG displays the 8085 status flags. The format of the display is:

\[ \text{SN}=b \quad Z=b \quad XX=0 \quad ACY=b \quad XX=0 \quad PY=b \quad XX=0 \quad CY=b \]

where: SN is the sign flag, Z is the zero flag, ACY is the auxiliary carry flag, PY is the parity flag, CY is the carry flag, XX are inactive bits, and b is the current value (0 or 1) of each of the flags.

**REMOVE MODULE Command**

Under ICE-85, to remove one or more modules (symbols and statement numbers) from the tables, use the command:

```
REMOVE MODULE ..module-name [ , ..module-name] ...
```

The modules named in the list must exist in the tables, or an error results. Modules not named in the list of module-names are not affected. The REMOVE MODULE command affects the symbol and line number tables only; code is not affected.

**LIMIT and LOWER**

To assist in mapping to Intellec shared memory, Multi-ICE ICE-85 includes three keywords (UPPER, LOWER, LIMIT) that refer to significant locations in the shared memory space. They are summarized in table 5-1. Refer to figure 5-1 for diagram, and to table 5-2 for a summary of memory blocks.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Location Referenced</th>
<th>How Set or Changed</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPPER</td>
<td>Lowest address in ICE workspace.</td>
<td>Set and changed by ICE; Read-only to user.</td>
</tr>
<tr>
<td>LOWER</td>
<td>Lowest block name higher than highest location occupied by ICE software. (i.e., lowest block free for mapping.)</td>
<td>Set by ICE and held constant for any software combination; Read-only to user.</td>
</tr>
<tr>
<td>LIMIT</td>
<td>Lowest address available to ICE for expanding workspace. An error occurs if LIMIT is greater than UPPER.</td>
<td>Set by ICE initially equal to LOWER; can be reset by the user to protect user space.</td>
</tr>
</tbody>
</table>
### Multi-ICE ICE-8S-Dependent Commands

**Figure 5-1. Intellec® Shared Memory At Initialization**

Table 5-2. ICE-85™ Memory Blocks for Mapping

<table>
<thead>
<tr>
<th>Block No.</th>
<th>Lowest Address</th>
<th>Highest Address</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Hex</td>
<td>Hex</td>
</tr>
<tr>
<td>31</td>
<td>62K</td>
<td>F800H</td>
</tr>
<tr>
<td>30</td>
<td>60K</td>
<td>F000H</td>
</tr>
<tr>
<td>29</td>
<td>58K</td>
<td>E800H</td>
</tr>
<tr>
<td>28</td>
<td>56K</td>
<td>E000H</td>
</tr>
<tr>
<td>27</td>
<td>54K</td>
<td>D800H</td>
</tr>
<tr>
<td>26</td>
<td>52K</td>
<td>D000H</td>
</tr>
<tr>
<td>25</td>
<td>50K</td>
<td>C800H</td>
</tr>
<tr>
<td>24</td>
<td>48K</td>
<td>COOH</td>
</tr>
<tr>
<td>23</td>
<td>46K</td>
<td>B800H</td>
</tr>
<tr>
<td>22</td>
<td>44K</td>
<td>B000H</td>
</tr>
<tr>
<td>21</td>
<td>42K</td>
<td>A800H</td>
</tr>
<tr>
<td>20</td>
<td>40K</td>
<td>A000H</td>
</tr>
<tr>
<td>19</td>
<td>38K</td>
<td>0800H</td>
</tr>
<tr>
<td>18</td>
<td>36K</td>
<td>0000H</td>
</tr>
<tr>
<td>17</td>
<td>34K</td>
<td>8000H</td>
</tr>
<tr>
<td>16</td>
<td>32K</td>
<td>8000H</td>
</tr>
<tr>
<td>15</td>
<td>30K</td>
<td>7000H</td>
</tr>
<tr>
<td>14</td>
<td>28K</td>
<td>7000H</td>
</tr>
<tr>
<td>13</td>
<td>26K</td>
<td>6800H</td>
</tr>
<tr>
<td>12</td>
<td>24K</td>
<td>6000H</td>
</tr>
<tr>
<td>11</td>
<td>22K</td>
<td>5800H</td>
</tr>
<tr>
<td>10</td>
<td>20K</td>
<td>5000H</td>
</tr>
<tr>
<td>9</td>
<td>18K</td>
<td>4800H</td>
</tr>
<tr>
<td>8</td>
<td>16K</td>
<td>4000H</td>
</tr>
<tr>
<td>7</td>
<td>14K</td>
<td>3800H</td>
</tr>
<tr>
<td>6</td>
<td>12K</td>
<td>3000H</td>
</tr>
<tr>
<td>5</td>
<td>10K</td>
<td>2800H</td>
</tr>
<tr>
<td>4</td>
<td>8K</td>
<td>2000H</td>
</tr>
<tr>
<td>3</td>
<td>6K</td>
<td>1800H</td>
</tr>
<tr>
<td>2</td>
<td>4K</td>
<td>1000H</td>
</tr>
<tr>
<td>1</td>
<td>2K</td>
<td>0800H</td>
</tr>
<tr>
<td>0</td>
<td>0K</td>
<td>0000H</td>
</tr>
</tbody>
</table>
ICE issues a warning under the following conditions:

1. A MAP command refers to an address higher than UPPER (for example, “MAP F800H = INTELLEC F800H”); you are mapping into current ICE workspace or over Monitor.

2. A MAP command refers to an address lower than LOWER (for example, “MAP 0 = INT 0”); you are mapping over ICE or over ISIS.

3. A MAP command refers to an address lower than UPPER and higher than LIMIT (for example, “MAP 3000H = INT E000H”); you are mapping into the area to be used for expanding the ICE workspace.

Initially, any MAP memory command receives a warning. The MAP command is executed in any event. Note that you cannot write to any location higher than UPPER or lower than LOWER; mapping these areas gives you read-only access to ISIS or Monitor routines. You can also reset LIMIT to avoid the warning message and protect your user code from being overwritten by the expanding ICE workspace.

The ICE workspace contains symbol and statement number tables and space for processing ICE commands, including macro expansions. It grows dynamically to accommodate larger symbol tables and command structures. As workspace grows, ICE resets UPPER to lower and lower addresses. UPPER always reflects the largest space required for command processing during each test session; it does not “shrink” dynamically to accommodate smaller commands. UPPER is reset to higher locations when you remove any symbols, modules, or lines from the tables.

ICE workspace can expand until UPPER equals LIMIT. Whenever an expansion of workspace results in UPPER less than LIMIT, an error occurs, the command is not executed, and control returns to the ICE command level.

If LIMIT remains equal to LOWER, as at initialization, the workspace can expand into the area mapped for user code without producing any error or warning message. This expansion may result in user code being overwritten.

You can set LIMIT so as to prevent the workspace from overwriting any user code. If LIMIT is reset before the user area is mapped, the warning message does not appear since this area is no longer part of the system area.

Sample Mapping Sequences

Here are some sample MAP command sequences to demonstrate some of the ways to use UPPER, LOWER, and LIMIT. As shown in figure 5-2, the sample multi-ICE configuration leaves three 2K memory blocks (6K in all) for the user space.
Example 1:

The user program is LOCATEd so as to begin at location 36C0H and is about 2K in length. It thus requires two consecutive memory blocks. This can be mapped very simply with the following command:

```
* MAP 3000H LENGTH 4K = INT LOWER
HOST/EN1 WARN C1: MAPPING OVER SYSTEM
```

When you don't think the workspace will ever expand down to the top of your code space and you don't mind receiving the warning message you can simply map your code to Intellec memory starting at the location given by the keyword LOWER. After this, you load your code in the normal manner.

Example 2:

The user program is LOCATED to start at address 2000H and is about 6K in length. The file containing the code is named "PROG". The commands are:

```
*LOAD :F1:PROG NOCODE ;load symbol tables only
*LIMIT = (UPPER/2K)*2K ;set LIMIT to the bottom of workspace
*MAP 2000H LEN 6K = INT LOWER
*LOAD :F1:PROG NOSYMBOLS NOLINES ;load the code
```
This kind of sequence is the one to use when you want to maximize the amount of user space available, since the command

\[ \text{LIMIT} = \text{UPPER/2K} \times 2K \]

limits the workspace to the rest of the block that it partially occupies at present (UPPER/2K \times 2K is the number of the block that contains the UPPER address). For example, consider the following command sequence:

* \text{UPPER}
  UPP=F5FCH ; lowest address in workspace.
* \text{LOWER}
  LOW=D800H ; lowest free block.
* \text{LIMIT}
  LIM=D800H ; lowest address available for expanding workspace.
* \text{LOAD} : F1:PROG NOCODE ; load symbol tables into workspace.
* \text{UPPER}
  UPP=F450H ; the workspace has expanded.
* \text{EVALUATE} \text{UPPER/2K} \times 2K
  1111000000000000Y 1700000 61440T FOOOH 'p' ; lowest address in current block
* \text{LIMIT} = F000H ; limit workspace expansion to current block.
* \text{MAP} 2000 \text{LEN} 6K = \text{INT LOWER}

Example 3:

The user program starts at 3000H and is about 4K in length. The file containing the code is named "PROG1". The following sequence allocates the lowest 4K (two blocks) to user code and the remaining 2K to workspace expansion:

* \text{LIMIT} = \text{LOWER} + 4K
* \text{MAP} 3000H \text{LEN} 4K = \text{INT LOWER}
* \text{LIMIT}
  LIM=E800H
* \text{LOWER}
  LOW=D800H
* \text{LOAD}: F1:PROG1

The new LIMIT (E800H in our example) is the number of the lowest block above the area mapped for user code (see figure 5-2).

Example 4:

The user program starts at 3000H and is just less than 3K in length; the file is "PROG2". The following sequence maps the lower two blocks to user code as in example 3, setting LIMIT just above the user code space. In contrast to example 3, that point is now in the middle of a block instead of at a block boundary.

* \text{LIMIT} = \text{LOWER} + 3K
* \text{MAP} 3000H \text{LEN} 4K = \text{INT LOWER}
* \text{HOST/EN1 WARN C1: MAPPING OVER SYSTEM}
  \text{LIMIT}
  LIM=E800H
* \text{LOWER}
  LOW=D800H
* \text{LIMIT}
  LIM=E400H

Note that we still receive the warning message since we are mapping higher than LIMIT. You must map in terms of the 2K blocks; however, LIMIT can be set to any address. In this case, LIMIT has been set so as to maximize the amount of workspace available to ICE.
Chapters 6, 7, 8, and 9 present the multi-ICE commands. In chapter 6 we describe the three basic multi-ICE commands (SWITCH, ACTIVATE, and KILL), and give a simple model of multi-ICE operation to help explain what the commands do. Chapters 7 and 8 introduce the synchronization commands (SUSPEND, CONTINUE, WAIT, BREAK, and LOCK), and add refinements to the basic model to help describe the effects of these commands. The full model is presented in chapter 9.

Chapter 6 is organized as follows:

- Syntax descriptions of the SWITCH, ACTIVATE, and KILL commands
- A simple model of multi-ICE operation
- Details on the SWITCH command
- Details on the ACTIVATE command
- Details on the KILL command
- System Displays and Messages

**SWITCH Commands**

```
SWITCH = ENn
SWITCH

Examples:

SWITCH = EN1
SWITCH = EN2
SWITCH

SWITCH
```

**ACTIVATE Command**

```
ACTIVATE PRn cr

[ ice-command cr] ...

ENDACTIVATE
```
### Example:

<table>
<thead>
<tr>
<th>Command</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVATE PR1</td>
<td></td>
</tr>
<tr>
<td>REPEAT</td>
<td></td>
</tr>
<tr>
<td>STEP COUNT 1</td>
<td></td>
</tr>
<tr>
<td>PRINT -1</td>
<td></td>
</tr>
<tr>
<td>UNTIL OPCODE = .RET</td>
<td></td>
</tr>
<tr>
<td>ENDREPEAT</td>
<td></td>
</tr>
<tr>
<td>REGISTERS</td>
<td></td>
</tr>
<tr>
<td>ENDACTIVATE</td>
<td></td>
</tr>
</tbody>
</table>

**ACTIVATE**

HOST-only command keyword; compound command block becomes PRn's *command list*, and PRn is set ACTIVE READY.

**PRn**

One of the keywords PR1 or PR2.

**cr**

Intermediate carriage return.

**ice-command**

Any simple or compound command that is valid for PRn; (refer to table –).

**ENDACTIVATE**

Command keyword that terminates the compound command block; may be abbreviated END.

### KILL Commands

**By HOST:**

1) KILL PRn
2) KILL ALL

**By ICE:**

KILL

**Examples:**

KILL PR1

KILL PR2

KILL ALL

ACTIVATE PR1

REPEAT

IF PC = .DELAY THEN

KILL

ENDIF

STEP COUNT

PRINT -1

UNTIL OPCODE = .RET

ENDREPEAT

REGISTERS

ENDACTIVATE

**KILL**

Command keyword; sets designated ICE process(es) to DORMANT (erases command list).

**PRn**

One of the keywords PR1 or PR2.

**ALL**

Keyword meaning “both PR1 and PR2”.
Discussion

The multi-ICE system consists of three processes, the HOST process and the two ICE processes PR1 and PR2, and two HOST parsing and execution environments, EN1 and EN2.

All three processes can execute commands. The HOST controls the entry of all commands from the console (or file).

PR1 always executes in environment EN1, and PR2 always executes in EN2. The HOST executes in either EN1 or EN2 as set by the current SWITCH.

The HOST process accepts commands through a parser. The HOST parser uses the keywords and syntax rules from either EN1 or EN2, as set by the current SWITCH.

The SWITCH commands set or display the parsing and execution environment for the HOST process. Details on SWITCH appear later in this chapter and in chapter 9.

Each process has a command buffer that can contain one or more commands called the "command list" for that process; the buffer can also be empty (has no commands).

A process is ACTIVE when it has a command list to execute and DORMANT when its buffer is empty.

The HOST executes any commands that are not within an ACTIVATE block; commands within an ACTIVATE block are executed by the ICE process (PR1 or PR2) named in the ACTIVATE command, as discussed later on.

The HOST process can execute any of the standard ICE emulation and interrogation commands. By SWITCHing the HOST back and forth you can operate the two ICES in sequence through the HOST.

When the HOST is ACTIVE, (executing or emulating), the prompt is suppressed and you cannot enter any commands. When the HOST process finishes executing its command list, the HOST becomes DORMANT. You can also abort the HOST’s command list by pressing the ESC key; the HOST becomes DORMANT.

The ACTIVATE command is used to pass a command list to the command buffer of one of the two ICE processes, making that process ACTIVE. The ACTIVATE command is a compound command, as shown in the syntax description earlier in the chapter; the keywords ACTIVATE and ENDACTIVATE mark the beginning and end of the block.

An ACTIVATE is executed by the HOST process. When you enter the command ACTIVATE PRI cr; the following occurs:

- The current SWITCH is saved, so that it can be restored after the end of the ACTIVATE block.
- PR1’s parsing and execution environment are automatically in effect.
- The prompt is issued, with a period to show that the subsequent commands are inside a block (that is, nested).

You can now enter the commands to be executed by PR1. The commands must be valid for PR1; details are given later in the chapter. The block terminates with ‘ENDACTIVATE’ (can be abbreviated to ‘END’).
When you enter ‘ENDACTIVATE’, the HOST transfers the command list that is within the ACTIVATE command (we can also call this the ‘ACTIVATE list’) to the command buffer of PR1; PR1 becomes ACTIVE and begins to execute the commands in its ACTIVATE list.

While either PR1 or PR2 is ACTIVE, the prompt is suppressed. You can interrupt an ICE process (not the HOST) without aborting its command list by pressing the spacebar. After the spacebar is pressed, the prompt is issued and you may enter a command. When you enter the final carriage return to terminate the entering of the command, the process that was interrupted resumes executing where it left off. Pressing the spacebar while an ICE process is emulating obtains the prompt without breaking emulation.

Using ACTIVATE commands, you can start both ICEs emulating at the same time. There are two ways to do this; the following two sequences produce equivalent results (note use of spacebar SP to obtain prompt):

1. *ACTIVATE PR1
   *GO FOREVER
   *ENDA
   PR1 EMULATION BEGUN
   SP *ACTIVATE PR2
   *GO TILL BR0
   *ENDA
   PR2 EMULATION BEGUN
2. *ACTIVATE PR1
   *GO FOREVER
   *ENDA
   PR1 EMULATION BEGUN
   SP *SW1=EN2
   *GO TILL BR0
   HOST/EN2 EMULATION BEGUN

Thus, when two processes are emulating, the two emulations are simultaneous (but asynchronous).

When two or three processes are executing commands other than GO (but including single steps), they alternate execution; after each process finishes its current simple command, the next process starts executing the next simple command in its command list. (Refer to chapter 9 for details.)

**Details on SWITCH Commands**

The initial environment is EN1.

The current SWITCH remains in effect until one of the following occurs:

- Another SWITCH command is entered to the HOST, not nested within any block.
- An ACTIVATE block is entered. ACTIVATE brings in the environment of the ICE process named; that environment remains in effect until ENDACTIVATE.
- An environment control is encountered in an expression. An environment control has one of the following two forms (see chapter 3):
  
  ENn primary
  ENn (expression)

  An environment control sets the parsing environment for the primary or parenthesized expression it precedes.
• If a SWITCH command appears in a block (REPEAT COUNT, IF, DEFINE MACRO, LOCK), it remains in effect until the end of the block or until another SWITCH command is encountered in the same block.

Details on ACTIVATE

The ACTIVATE command may not contain any of the following HOST-only commands:

- SWITCH = ENn
- ACTIVATE PRn
- KILL PRn
- KILL ALL
- SUSPEND PRn
- SUSPEND ALL
- CONTINUE PRn
- CONTINUE ALL
- WAIT PRn
- WAIT ANY
- BREAK PRn
- BREAK ALL

If you try to activate a process that is already ACTIVE, an error occurs and the command is ignored.

Details on KILL

Initially, both PR1 and PR2 are DORMANT; their command buffers have no commands. An ICE process becomes DORMANT (from another state) when one of the following occurs:

• The process kills itself by executing the last commands in its command list.
• The process kills itself by executing a KILL command in its command list.
• The user kills the process through the HOST by entering a KILL PRn or KILL ALL command.
• The user kills the process by pressing ESC and answering "Y" to the query "KILL PRn?".

Using the ESC Key to Kill a Process

An ESC entered when the prompt is displayed aborts the command that is being entered. ESC does not terminate the parser’s task, however, and another prompt is issued automatically. Pressing the ESC key while any process is ACTIVE (prompt is suppressed) invokes an abort routine to perform the following actions:

• The HOST process is set DORMANT if it is not already DORMANT.
• The message “HOST PROCESSING ABORTED” is displayed.
• If PR1 is ACTIVE (not SUSPENDED or DORMANT), the system displays the query “KILL PR1?” To set PR1 DORMANT, enter “Y” followed by cr. To allow PR1 to remain ACTIVE, press cr immediately. (Entering any character other than “Y” also allows PR1 to remain ACTIVE.)
• If PR2 is ACTIVE, the system asks “KILL PR2?”. Enter “Y” cr to set PR2 DORMANT, or enter cr to allow PR2 to remain ACTIVE.
Summary of Multi-ICE Messages

ERRORS
Error message from HOST (parser or process)

HOST/ENn ERR xx: description of error

ERROR message from ICE process

PRn ERR xx: description of error

EXECUTION—when commands produce console displays (except for EMUL).

Header is displayed when the current process is different from the one that produced
the previous display.

HOST/ENn: cr/lf if HOST executes display command

PRn: cr/lf if ICE executes display command

EMULATION:

HOST/ENn EMULATION BEGUN
HOST/ENn EMULATION TERMINATED, DC = address

PRn EMULATION BEGUN
PRn EMULATION TERMINATED, PC = address

ABORT:

HOST PROCESSING ABORTED
KILL PRn?
This chapter presents the SUSPEND, CONTINUE, and WAIT commands. These commands control the synchronization of two or three ACTIVE processes by inserting a SUSPENDED state that remains in effect until the process is continued by command or by condition.

When a process (HOST or ICE) is SUSPENDED, it retains its command list but cannot execute any commands. If the process was EMULATING when suspended, emulation breaks on the next instruction. When the process is subsequently continued, emulation begins again at the current PC.

The two ICE processes can be suspended with the SUSPEND commands, and continued with the CONTINUE command. The HOST process can be suspended with a WAIT command; the HOST continues when the process named in the WAIT command is no longer ACTIVE.

**SUSPEND Commands**

<table>
<thead>
<tr>
<th>By HOST:</th>
<th>1) SUSPEND PRn</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2) SUSPEND ALL</td>
</tr>
</tbody>
</table>

By ICE: SUSPEND

Examples:

SUSPEND PR1
SUSPEND PR2
SUSPEND ALL
ACTIVATE PR1
SUSPEND
GO TILL BR0
PRINT ALL
ENDACTIVATE

<table>
<thead>
<tr>
<th>SUSPEND</th>
<th>Command keyword that halts execution or emulation, sets PRn to SUSPENDED status.</th>
</tr>
</thead>
<tbody>
<tr>
<td>PRn</td>
<td>One of the keywords PR1 or PR2.</td>
</tr>
<tr>
<td>ALL</td>
<td>Command keyword meaning “both PR1 and PR2”.</td>
</tr>
</tbody>
</table>
CONTINUE Commands

1) CONTINUE PRn
2) CONTINUE ALL

Examples:
CONTINUE PR1
CONTINUE PR2
CONTINUE ALL

CONTINUE  HOST-only command keyword; causes designated ICE process(es) to resume executing or emulation from SUSPENDED status.

PRn  One of the keywords PR1 or PR2.

ALL  Keyword meaning “both PR1 and PR2”.

WAIT Command

1) WAIT PRn
2) WAIT ANY

Examples:
WAIT PR1
WAIT PR2
WAIT ANY

WAIT  HOST-only command keyword; causes HOST to become SUSPENDED until designated ICE process is no longer ACTIVE.

PRn  One of the keywords PR1 or PR2.

ANY  Keyword meaning “either PR1 or PR2”.
SUSPEND Commands and CONTINUE Command

The SUSPEND command has two forms: one form can be executed only by the HOST and the other can be executed only by an ICE process as part of its ACTIVATE list. The two forms have the following syntaxes:

**SUSPEND by HOST:**

1. SUSPEND PR\(n\)
2. SUSPEND ALL

**SUSPEND by ICE:**

SUSPEND

The effect of either form of SUSPEND command is to set the ICE process (both processes with SUSPEND ALL) to the SUSPENDED status. The process cannot execute any more of the commands in its ACTIVATE list until the HOST executes a CONTINUE command. The syntax of the CONTINUE command is:

1. CONTINUE PR\(n\)
2. CONTINUE ALL

If the designated process in a SUSPEND command is in any process status other than ACTIVE, an error message is displayed, and no action is taken. Similarly, the process named in a CONTINUE command must be SUSPENDED for the command to be valid.

WAIT Command

The WAIT command has one of two forms:

WAIT PR\(n\)
WAIT ANY

Both forms are HOST-only (must be outside ACTIVATE).

When WAIT PR1 (for example) is executed, the HOST process becomes SUSPENDED if PR1 is ACTIVE, and remains SUSPENDED until PR1 becomes DORMANT or SUSPENDED.

With WAIT ANY, the HOST continues as soon as either PR1 or PR2 is no longer ACTIVE.

If WAIT is entered while the named process is not ACTIVE, the command has no effect.

Example with SUSPEND, CONTINUE, and WAIT

To check the 8 bit wide data bus in a system with two iSBC 80/30 boards, we can command the 1st iSBC 80/30 system to write 00H, 01H, 03H, 07H, 0FH, 1FH, 3FH, 7FH and 1FFH into location 8000H (the 1st byte in the iSBC 032 memory card accessible from both 80/30s) and command the 2nd iSBC 80/30 system to read that location after each value is written and verify it. PR1 and PR2 will be synchronized by the host process.
MAP 8000 = USER
SWITCH = EN2
MAP 8000 = USER
SWITCH = EN1

DEFINE IND. I = 0
DEFINE IND. J = 0
DEFINE IND. K = 0

ACT PR1
  COUNT 9
  SUSPEND
  BYTE 8000 = IND. I
ENDC
ENDA

* ACT PR2
  COUNT 9
  SUSPEND
  IND. J = BYTE 8000 ; PR2 READS COMMON MEMORY LOCATION 8000
  IF IND. I <> IND. J
    WRITE 'SHARED MEMORY DATA ERROR:'
    WRITE 'EXPECTED = ', IND. I, '; ACTUAL = ', IND. J
  ENDIF
  IND. K = IND. K + 1 ; IF ERROR INCREMENT ERROR COUNTER K
ENDIF
  IND. I = IND. I * 2 + 1
ENDC
ENDA

REPEAT
  CONTINUE PR1
  WAIT PR1
  CONTINUE PR2
  WAIT PR2
  UNTIL DORMANT PR1 AND DORMANT PR2
ENDR
WRITE IND. K, ' ERRORS OCCURRED'
CHAPTER 8
BREAK AND LOCK COMMANDS

The BREAK and LOCK commands presented in this chapter allow you to break emulation without aborting the command list, and to have a process gain exclusive control of the console. These commands can be combined with the multi-ICE commands presented in chapters 6 and 7 to allow close control of the multi-ICE operation.

**BREAK Command**

1) BREAK PRn
2) BREAK ALL

Examples:

```
BREAK PR1
BREAK PR2
BREAK ALL
```

**BREAK**

HOST-only command keyword; causes designated ICE process to break emulation, resume execution with next command in command-list.

**PRn**

One of the keywords PR1 or PR2.

**ALL**

Keyword meaning “both PR1 and PR2”.

**LOCK Command**

```
LOCK or
[command or] ...
ENDLOCK
```

Examples:

```
LOCK
REPEAT
STEP COUNT 1
PRINT -1
ENDREPEAT
REGISTERS
ENDLOCK

ACTIVATE PR2
LOCK
REPEAT
STEP COUNT 1
PRINT -1
ENDREPEAT
ENDLOCK
ENDACTIVATE
```
LOCK Command

Command keyword; compound command block cannot be pre-empted by another process until all commands in the block have been executed.

cr

Intermediate carriage return.

command

Any simple or compound command except WAIT and SUSPEND.

ENDLOCK

Command keyword terminating the compound command block; can be abbreviated to END.

Discussion

BREAK Commands

The BREAK commands are HOST-only (must appear outside ACTIVATE). They cause the named ICE process (or both ICE processes with 'BREAK ALL') to terminate emulation and continue executing the next command in its command list.

Consider the following sequence:

```
ACTIVATE PRI
   GO FOREVER
   REGISTERS
END ACTIVATE
```

When this block executes, PRI enters emulation and continues in emulation "FOREVER", that is until terminated manually. During PRI's emulation, you can press the spacebar to obtain the prompt.

After obtaining the prompt, you can issue the KILL command to abort both the emulation and the rest of PRI's command list or, you can SUSPEND PRI halting emulation; when you subsequently CONTINUE PRI, however, emulation resumes at the current PC. Either way, the REGISTERS command in the ACTIVATE block is never executed.

The BREAK command allows you to break emulation through the HOST and continue executing the commands in the process' command list. Using the previous example, when the ACTIVATE block begins, we can request the prompt, then enter the command 'BREAK PRI'. PRI breaks emulation, then executes the REGISTERS command.

If BREAK is entered when the designated ICE process is ACTIVE but not emulating, the command has no effect.

An ICE process can be SUSPENDED while emulating; when such a process is subsequently CONTINUED, emulation resumes with the current PC. However, if you enter a BREAK command before CONTINUE, the process resumes executing the next command in its list after the emulation command.

LOCK Command

LOCK is a compound command; the block begins with LOCK and ends with 'ENDLOCK'.

The LOCK command block can contain any commands other than SUSPEND or WAIT commands.

LOCK can appear at any level (in or out of ACTIVATE).

The commands within the current LOCK block are executed exclusively; the LOCK command suppresses the console-sharing that normally occurs under Multi-ICE when more than one process is ACTIVE.
This chapter describes the concepts and commands that allow you to control and coordinate the operations of two ICEs from one Intellec system.

To provide a framework for understanding how the commands work, we provide a model of a multi-ICE system such as 85/85 or 85/49. This model is conceptual in nature; it does not describe exactly how a given feature is implemented in the ICE software. However, the model is consistent with the software implementation, and for that reason can be used to predict what will happen under most conditions that arise during system operation.

Refer to Chapters 6, 7, and 8 for the details on the multi-ICE commands (syntax, examples).

**Components of a Dual-ICE System**

A dual-ICE system has the following software components (figure 9-1):

- **The HOST parser** issues the console prompt, receives commands from the console or from a file, parses the commands into command code using the parse tables from the HOST's current environment (EN1 or EN2), and loads the command code into the HOST's command buffer.
- **The HOST execution process** (keyword: HOST) executes commands from its command code buffer using the execution software and hardware of the HOST's current environment (EN1 or EN2) as required.
Multi-ICE Theory of Operation

- **The two ICE execution processes** (keywords: PR1, PR2) execute commands from their command code buffers in their own environments (PR1 in EN1, PR2 in EN2).
- **The dispatcher** polls available tasks (commands to parse or execute) in a fixed sequence and uses the current status of each task when polled to decide whether to allow that task to be performed or to skip it and poll the next task.

Table 9-1 summarizes the keywords used to refer to the HOST and ICE processes and to the two parsing and execution environments. The parser and the dispatcher do not have any associated keywords.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST</td>
<td>HOST execution process</td>
</tr>
<tr>
<td>PR1</td>
<td>First ICE execution process (e.g., ICE-85 in dual ICE 85/49)</td>
</tr>
<tr>
<td>PR2</td>
<td>Second ICE execution process (e.g., ICE-49 in 85/49)</td>
</tr>
<tr>
<td>EN1</td>
<td>Parsing and execution environment of the first ICE.</td>
</tr>
<tr>
<td>EN2</td>
<td>Parsing and execution environment of the second ICE.</td>
</tr>
</tbody>
</table>

### Processes and Process Status

**HOST and ICE Processes**

A process is a conceptual entity that can directly execute certain ICE commands ("software commands") and can call upon designated ICE hardware through a hardware interface to execute other ("hardware") commands. There are three processes in a dual-ICE system: the HOST process and the two ICE processes (table 9-1). In this manual the term "PRn" is used to mean "either PR1 or PR2"; PRn is not a keyword.

A process uses a command code buffer and an execution environment. Figure 9-2 diagrams a generalized execution process.

![Figure 9-2. A Generalized Execution Process](762-4)

Each process has its own command code buffer. A command code buffer can contain one or more commands. Commands are parsed into command codes by the parser, loaded into the HOST's command code buffer, and then are either executed by the HOST or transferred to the buffer of one of the two ICE processes for execution. The contents of a command code buffer are called a "command list." For an ICE process the contents are also called an "ACTIVATE list" because they are passed to the ICE process with an ACTIVATE command, as discussed later in this chapter.
An execution environment includes ICE software registers, a set of ICE “firmware registers”, the hardware interface, and the ICE hardware. ICE firmware is ROM-based ICE code that is on the ICE boards. “Firmware registers” are RAM-based registers used directly by the firmware to control the hardware interface; many of these are copies of corresponding software registers. In most situations the details of the execution environment are invisible to the user.

Each of the two ICE processes operates within its own execution environment. The environment of PR1 is identified by the keyword EN1, and keyword EN2 identifies the environment of PR2. The HOST process can operate in the execution environment of one of the two ICE processes. The HOST’s shared execution environment can be switched by command from one ICE to the other; switching the HOST’s execution environment also switches the HOST’s parsing environment, as discussed later on.

**Process Status**

Associated with each process is a *process status*; refer to table 9-2 for a summary of process status keywords. The HOST execution process is ACTIVE when its command buffer contains one or more commands; it is DORMANT when its buffer is empty; it is SUSPENDED when a WAIT command is pending.

An ICE process is DORMANT when its command code buffer is empty; it is ACTIVE when its buffer contains one or more commands and that ICE process has not been suspended by command; it is SUSPENDED when its buffer contains one or more commands and the process has been suspended by command.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>DORMANT</td>
<td>Process’ command buffer empty.</td>
</tr>
<tr>
<td>ACTIVE</td>
<td>Process’ command buffer has commands and process is not suspended by command.</td>
</tr>
<tr>
<td>SUSPENDED</td>
<td>Process is suspended by command while emulating or executing.</td>
</tr>
</tbody>
</table>

**Querying Process Status**

The process status keywords ACTIVE, DORMANT, and SUSPENDED can be used to query the system regarding the status of any process. A BOOL command with the form:

```
BOOL process-status process
```

displays “TRUE” if the given process (HOST, PR1, PR2) is currently in the given process-status (ACTIVE, SUSPENDED, DORMANT); otherwise, the command displays “FALSE”.

**HOST Parsing and Execution Environment**

The HOST parser and HOST execution process can use the resources of either of the two ICEs. These resources include the execution environment discussed earlier, and the parsing environment: a set of parse tables used to specify the valid keywords and syntax for any ICE-dependent commands.
The SWITCH command (described in detail in Chapter 6) sets the parsing and execution environment for the HOST parser and process. Both aspects (parser and execution process) of the HOST are always in the same environment, termed the “current SWITCH”.

Initially (after initial program load from ISIS), the current SWITCH is EN1. Commands are parsed using the parse tables of the first ICE and the HOST executes its commands in the execution environment of the first ICE. From the initial state, if you enter the command:

```
SWITCH = EN2
```

the parse tables of the second ICE are read in, unless the two ICEs are the same kind (for example 85/85). The next HOST execution will use the execution resources of that ICE; even if the two ICEs are of the same kind, their execution resources are independent.

If you subsequently enter:

```
SWITCH = EN1
```
you are back in the parsing and execution environment of the first ICE.

## Tasks and Task Status

A *task-slice* is the time required for one of the following actions to be completed:

- A process executes one simple command.
- A process emulates one instruction in single-step mode.
- A process enters real-time emulation.
- The parser parses one complete simple or compound command and encounters the final *cr*.

A *task* is a process (or parser) that requires a task-slice to perform its next action.

At any given time, there may be one or more tasks requiring scheduling. The dispatcher polls the possible tasks in a particular sequence; the polling sequence is described in detail later in this chapter.

Associated with each task is its *task status*. At the moment the dispatcher polls a task, that task is either READY or NOT READY. If an execution process is READY when polled, it is allowed to perform its next action. If the parser is READY when polled, a prompt is issued. We now describe the conditions under which a given task is READY or NOT READY.

### NOTE

The task statuses READY and NOT READY are shown in upper case in this manual, following the common convention for logic states that is also used for TRUE and FALSE. However, READY and NOT READY are *not* ICE keywords; they cannot be used in commands. The status of any task cannot be queried directly, but must be inferred from the sequence of commands previously entered and executed.
The HOST Parser

The HOST parser processes all input from the console or from files. The parser signals its readiness to accept a command by issuing an asterisk prompt (*).

Obtaining a Prompt

Here are some guidelines for obtaining a prompt. Details are given in succeeding paragraphs.

- The prompt is issued automatically if the HOST process is DORMANT and the ICE processes are either DORMANT or SUSPENDED.
- If the HOST process is DORMANT and one or both ICE processes are ACTIVE, press the spacebar to obtain a prompt after the current task is completed.
- If the HOST process is ACTIVE, and both ICE processes are either DORMANT or SUSPENDED, press the ESC key to abort the HOST (HOST becomes DORMANT); the prompt is now issued automatically.
- If the HOST process is ACTIVE and one or both ICE processes are ACTIVE:
  1. Press ESC to set the HOST process DORMANT.
  2. The system asks you if you want to kill the ACTIVE ICE process(es); (killing them sets them DORMANT).
  3. If you kill all ACTIVE ICE processes, the prompt is issued automatically.
  4. If you don't kill all ACTIVE processes, press the spacebar to obtain the prompt after the current task is completed.

NOTE

If you press any character keys while the parser is suppressed, the characters are ignored. The system outputs a "BELL" character (producing a "beep" at most CRT terminals) to inform you that a character key is being ignored; you must request a prompt to enter your command.

Entering Commands

When the parser is dispatched, it issues a prompt and waits for a command to be entered. While the parser is thus waiting, no process can begin executing any command. If a process is emulating when the parser is dispatched, the process keeps emulating; however, if the process breaks emulation (reaches a breakpoint) it cannot display "EMULATION TERMINATED" at the console until the parser completes its action.

The parser completes its action when you enter a final cr (carriage return), or when a parser error (such as SYNTAX ERROR or INVALID TOKEN) occurs.

At the completion of a parser action, the encoded command is loaded into the HOST's command buffer (HOST process becomes ACTIVE), the HOST process' task status is set READY, and the parser's task status is set NOT READY.
Intermediate and Final Carriage Return

A command can contain intermediate cr’s in addition to the final cr. A final cr means one of the following:

- A cr entered immediately after the prompt, or with only blanks preceding the cr (in other words, a “null” command).
- A cr entered after a single-line simple command that is not inside any compound command.
- A cr entered after the END keyword of a compound command that is not inside any other compound command.
- A cr entered at the end of a continuation line (that is, a cr not preceded by the continuation character “&” on the same line).

An intermediate cr means one of the following:

- A cr preceded by the continuation character “&” on the same line. The system acknowledges the continuation by displaying a double prompt (**) on the next line.
- A cr that ends a simple or compound command that is nested inside a compound command (REPEAT, COUNT, IF, ACTIVATE, LOCK or DEFINE MACRO command). The parser informs you of the nesting level by prefixing the prompt with one period (.) for each level of nesting. The outer level is level zero (no period).

Parser Task Status

Refer to figure 9-3 and table 9-3. In the preceding sections, we described how to obtain a prompt (get the parser dispatched) and how to terminate a command (complete the parser's action). These discussions emphasized the user's point of view. This section describes the parser as "seen" by the dispatcher.

The dispatcher periodically polls the parser. If the parser is READY when polled, it is dispatched (given control of the console). If the parser is NOT READY when polled, the dispatcher skips the parser and looks for a process to dispatch. The next paragraphs describe the conditions under which the parser is READY or NOT READY.

<table>
<thead>
<tr>
<th>Task Status</th>
<th>HOST Execution Process Status</th>
<th>ICE Process Status</th>
<th>Spacebar Pressed</th>
<th>INCLUDE or SUBMIT</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOT READY</td>
<td>ACTIVE or SUSPENDED</td>
<td>—</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td>DORMANT</td>
<td>One or both ACTIVE</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>READY</td>
<td>DORMANT</td>
<td>One or both ACTIVE</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td></td>
<td>DORMANT</td>
<td>One or both ACTIVE</td>
<td>—</td>
<td>YES</td>
</tr>
<tr>
<td></td>
<td>DORMANT</td>
<td>No PRn ACTIVE</td>
<td>—</td>
<td>—</td>
</tr>
</tbody>
</table>
READY
The parser is set READY under any one of the following conditions:

- The HOST process is DORMANT and both ICE processes are either DORMANT or SUSPENDED.
- The HOST process is DORMANT and one or both ICE processes are ACTIVE, and the user has requested the prompt by pressing the spacebar.
- The HOST process is DORMANT and one or both ICE processes are ACTIVE, and commands are being read (input) from a diskette file using the ISIS-II SUBMIT command.

NOT READY
The parser is NOT READY when any process is ACTIVE and either the user has not requested any service (spacebar or ESC) or input is not coming from a file.

Thus, the user always receives the prompt after initialization and at any other time that no execution process is ACTIVE.

The prompt is suppressed whenever the HOST process is ACTIVE or SUSPENDED (buffer has commands), since any new input would erase the commands currently in the buffer. To get a prompt when the HOST is ACTIVE, you must set the HOST DORMANT by pressing the ESC key.
If the HOST process is DORMANT but one or both ICE processes are ACTIVE, the prompt is suppressed. However, you can obtain the prompt after the current task is completed by pressing the spacebar; this does not change the ACTIVE status of the process that was pre-empted.

The ISIS SUBMIT command causes ICE to take its input from a diskette file rather than from the keyboard. Under file control, the prompt is issued after each task executed by an ICE process unless the HOST process is active. When the HOST process is ACTIVE the prompt is suppressed, even under file control. The remainder of this chapter assumes that input is from the console rather than from a file.

**HOST Execution Process**

The HOST process executes commands in its command list. The HOST’s command list contains any commands transferred by the parser into the HOST’s command code buffer that are not inside an ACTIVATE block. As discussed later, the HOST executes an ACTIVATE command by passing the commands inside the ACTIVATE block to the given ICE process for execution.

**Commands to the HOST**

The HOST can execute three kinds of commands, as follows:

1. *HOST-only multi-ICE commands*:
   - ACTIVATE PR1
   - ACTIVATE PR2
   - ENDACTIVATE
   - SUSPEND PR1
   - SUSPEND PR2
   - SUSPEND ALL
   - CONTINUE PR1
   - CONTINUE PR2
   - CONTINUE ALL
   - BREAK PR1
   - BREAK PR2
   - BREAK ALL
   - KILL PR1
   - KILL PR2
   - KILL ALL
   - WAIT PR1
   - WAIT PR2
   - WAIT ANY
   - SWITCH = EN1
   - SWITCH = EN2

Details on these commands appear later in this chapter.

2. *ICE-independent single-ICE commands*, specifically:
   - REPEAT commands
   - COUNT commands
   - IF commands
   - BOOL expression
   - MACRO commands
   - LOCK command
   - Commands that reference the IND symbol table
   - WRITE command

REPEAT, COUNT, IF, BOOL, MACRO, and WRITE commands are described in chapter 6.
3. *ICE-dependent single-ICE commands*, if the ICE on which the commands depend is the current SWITCH.

For example, if the current SWITCH is an ICE-85 environment, the HOST can execute any of the commands described in Chapter 7 (SEARCH, DOMAIN, NESTING, LINE, MODULE, FLAG, IMASK), as well as any of the standard ICE-85 commands from the ICE-85 operator's manual.

The HOST can perform the same debugging operations that an ICE process can perform. One main difference noted so far is that an ICE process can be pre-empted with the spacebar without erasing its command list, whereas the HOST process can only be interrupted with the ESC key, erasing the HOST's command list.

**HOST Process Status and Task Status**

Refer to figure 9-4 and table 9-4.

<table>
<thead>
<tr>
<th>Process Status</th>
<th>Task Status</th>
<th>Command Buffer</th>
<th>In Real-Time Emulation</th>
<th>Waiting On ACTIVATE PRn</th>
</tr>
</thead>
<tbody>
<tr>
<td>DORMANT</td>
<td>NOT READY</td>
<td>Empty</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>ACTIVATE</td>
<td>NOT READY</td>
<td>Has Commands</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>READY</td>
<td>Has Commands</td>
<td>NO</td>
<td>NO</td>
<td>NO</td>
</tr>
<tr>
<td>SUSPENDED</td>
<td>NOT READY</td>
<td>Has Commands</td>
<td>NO</td>
<td>YES</td>
</tr>
</tbody>
</table>

**DORMANT**

Initially, the HOST process is DORMANT; its buffer is empty of commands. When the HOST process is DORMANT, its task status is NOT READY.

The HOST process becomes DORMANT when it finishes executing its command list or when the user presses the ESC key.

**ACTIVE READY (Executing)**

When the parser completes its input task, it passes the commands to the HOST's command buffer and sets the HOST process ACTIVE READY. (The parser becomes NOT READY and remains so as long as the HOST is ACTIVE.) When the HOST process is subsequently polled by the dispatcher, it executes the first command in its buffer; that task can be one of the following:

- Execute one simple command other than STEP or GO.
- Emulate one instruction in single-step mode (that is, one COUNT of a STEP command).
- Begin real-time emulation. A GO command begins execution by issuing a "begin emulation" directive to the hardware controlled by the current SWITCH. A GO command completes execution when the hardware reports "emulation terminated".

After performing any task other than entering real-time emulation, the HOST process remains ACTIVE READY until it has exhausted its command list or the user presses ESC.
ACTIVE NOT READY (Emulating)

When the HOST begins executing a GO command, it remains ACTIVE but becomes NOT READY; it is waiting for the hardware to signal "emulation terminated" and cannot be dispatched for another task. The HOST remains in this state until a breakpoint is reached (assuming the user has specified a halt condition other than FOREVER) or the user presses ESC. After a breakpoint, the HOST becomes ACTIVE READY; after ESC, the HOST becomes DORMANT.

If the halt condition is FOREVER (or if for some reason the breakpoints are never reached), the ESC key is the only way to terminate the HOST's emulation. Pressing the ESC key has the "side effect" of erasing the HOST's entire command buffer. If the GO FOREVER command is in a block with other commands, the commands that appear later than the GO command can never be executed. Thus for the HOST process, GO FOREVER should be the last command in a block. By contrast, emulation by an ICE process can be broken from the console using the BREAK command without erasing anything, thus allowing execution to continue.
SUSPENDED
The HOST process becomes SUSPENDED NOT READY when it executes a WAIT PRn or WAIT ANY command. Under WAIT PRn, the HOST remains SUSPENDED NOT READY until PRn is no longer ACTIVE; under WAIT ANY, the HOST remains SUSPENDED NOT READY until either process becomes SUSPENDED or DORMANT.

The ICE Processes
The two ICEs in a multi-ICE system are referred to with the keywords PR1 and PR2. In effect, each ICE process is assigned a "logical" number; the assignment is fixed for a given multi-ICE module. For example, in the dual-ICE system consisting of ICE-85 and ICE-49, PR1 always refers to the ICE-85 and PR2 always refers to the ICE-49.

Commands to the ICE Processes
An ICE process executes any commands within the ACTIVATE list it receives from the HOST. ICE process PR1 uses the EN1 execution environment, and PR2 uses the EN2 environment. An ICE process can execute three kinds of commands, as follows:
1. *ICE-only multi-ICE commands*, specifically:
   SUSPEND
   KILL
   As discussed later in this chapter, SUSPEND causes the ICE to "pause" until it is continued by the HOST, and KILL erases the ICE's command buffer, setting the ICE process to DORMANT.
2. *ICE-independent single-ICE commands*; these commands are the same as the ones given for the HOST process in this category.
3. *ICE-dependent single-ICE commands*, if they depend on the given ICE. Actually, the parser detects invalid commands in an ACTIVATE list and issues an error message immediately; thus an ICE process never tries to execute any commands that are not valid for it.

ICE Process Status and Task Status
Refer to figure 9-5 and table 9-5.

<table>
<thead>
<tr>
<th>Process Status</th>
<th>Task Status</th>
<th>Command Buffer</th>
<th>In Real-Time Emulation</th>
<th>Suspended By Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>DORMANT</td>
<td>NOT READY</td>
<td>Empty</td>
<td>—</td>
<td>—</td>
</tr>
<tr>
<td>SUSPENDED (Executing)</td>
<td>NOT READY</td>
<td>Has Commands</td>
<td>NO</td>
<td>YES</td>
</tr>
<tr>
<td>SUSPENDED (Emulating)</td>
<td>NOT READY</td>
<td>Has Commands</td>
<td>Was Emulating When Suspended</td>
<td>YES</td>
</tr>
<tr>
<td>ACTIVE (Emulating)</td>
<td>NOT READY</td>
<td>Has Commands</td>
<td>YES</td>
<td>NO</td>
</tr>
<tr>
<td>ACTIVE (Executing)</td>
<td>READY</td>
<td>Has Commands</td>
<td>NO</td>
<td>NO</td>
</tr>
</tbody>
</table>
DORMANT (NOT READY)

Initially, both PR1 and PR2 are DORMANT; their command buffers have no commands. An ICE process becomes DORMANT (from another state) when one of the following occurs:

- The process kills itself by executing the last command in its command list.
- The process kills itself by executing a KILL command in its command list.
- The user kills the process through the HOST by entering a KILL PRn or KILL ALL command.
- The user kills the process by pressing ESC and answering "Y" to the query "KILL PRn?".

An ICE process can be set DORMANT from any other state (ACTIVE READY, ACTIVE NOT READY, or SUSPENDED). However, the actions that can set a process DORMANT depend to some degree on its current state, and on whether the ICE process is currently executing commands from within a LOCK block. Table 9-6 summarizes the conditions that can set an ICE process DORMANT.
Table 9-6. How an ICE™ Process Becomes DORMANT

<table>
<thead>
<tr>
<th>Current State Of Process</th>
<th>Actions That Set Process DORMANT From Current State</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVE READY</td>
<td>• Process executes last command in buffer.</td>
</tr>
<tr>
<td></td>
<td>• Process executes KILL command.</td>
</tr>
<tr>
<td></td>
<td>• HOST executes KILL PRn or KILL ALL command.</td>
</tr>
<tr>
<td></td>
<td>• User kills process through ESC.</td>
</tr>
<tr>
<td>ACTIVE NOT READY</td>
<td>• Process breaks emulation, finds buffer empty.</td>
</tr>
<tr>
<td></td>
<td>• HOST executes KILL PRn or KILL ALL command.</td>
</tr>
<tr>
<td></td>
<td>• User kills process through ESC.</td>
</tr>
<tr>
<td>SUSPENDED</td>
<td>• HOST executes KILL PRn or KILL ALL command.</td>
</tr>
<tr>
<td>Executing LOCK Block (ACTIVE)</td>
<td>• Process executes last command in buffer.</td>
</tr>
<tr>
<td></td>
<td>• Process executes KILL command.</td>
</tr>
<tr>
<td></td>
<td>• Process breaks emulation, finds buffer empty.</td>
</tr>
<tr>
<td></td>
<td>• User kills process through ESC.</td>
</tr>
</tbody>
</table>

**ACTIVE READY (Executing)**

When the HOST process executes an ACTIVATE PRn command, the commands in the ACTIVATE list are transferred to PRn’s command buffer and PRn is set ACTIVE READY. The next time the dispatcher polls PRn, the first task from its list is executed; the task can be one of the following:

- Execute one simple command other than STEP or GO.
- Emulate one instruction in single-step mode.
- Enter real-time emulation.

After executing any task other than entering emulation, the ICE process remains ACTIVE READY if it still has commands in its buffer.

**ACTIVE NOT READY (Emulating)**

When an ICE process executes a GO command, it remains ACTIVE but is set NOT READY; it is waiting for the hardware to report "emulation terminated". The process cannot execute any further commands until emulation terminates. If a break condition has been enabled, emulation can terminate on the break condition. If no breakpoints have been set (or if none are ever encountered during emulation), emulation continues until one of the following occurs:

- An error is encountered during emulation (for example, GUARDED ACCESS).
- The user breaks emulation by entering a BREAK PRn or BREAK ALL command through the HOST; the BREAK command sets the ICE process ACTIVE READY if the process has any commands left, or DORMANT if no commands remain.
- The user breaks emulation by entering a SUSPEND PRn or SUSPEND ALL command; the process becomes SUSPENDED. If the process is continued by the HOST, emulation resumes where it broke off and the process becomes ACTIVE NOT READY again.
- The user kills the process by entering a KILL PRn or KILL ALL command; the process becomes DORMANT.
- The user kills the process by pressing ESC and answering "Y" to the query "KILL PRn?"; the process becomes DORMANT.
SUSPENDED (NOT READY)

You can suspend an ACTIVE process through the HOST by entering a SUSPEND PRn command. If both ICE processes are ACTIVE, you can suspend them both with a SUSPEND ALL command. An ICE process can suspend itself by executing a SUSPEND command in its ACTIVATE list.

SUSPENDED (Executing)

If the process was executing (ACTIVE READY) when suspended, it finishes its current action, then is set SUSPENDED (Executing). It can resume executing when you enter a CONTINUE PRn or CONTINUE ALL command through the HOST.

SUSPENDED (Emulating)

If the process was emulating (ACTIVE NOT READY) when suspended, emulation is terminated between instructions and the "EMULATION TERMINATED" message is displayed. The process is set SUSPENDED (Emulating). If the process is subsequently continued, emulation resumes with the instruction pointed to by the program counter and the "EMULATION BEGUN" message is displayed.

A suspended process can be killed (set to DORMANT) by entering a KILL PRn or KILL ALL command to the HOST. A process that is SUSPENDED (Emulating) can be set to SUSPENDED (Executing) by the BREAK command.

The Dispatcher

The dispatcher is a piece of software that is called when one of the following occurs:

- A process (HOST or ICE) completes the execution of a simple command, emulates one instruction in single-step mode, or enters real-time emulation.
- The parser encounters a final cr.
- An error occurs; the dispatcher is called after the error message is displayed.

The dispatcher has two main functions:

- Check for interrupts (ESC key, spacebar, hardware emulation terminated, hardware error).
- Allocate a task slice to the next READY task, and exit the dispatcher.

The operation of the system depends on two sequences:

- The sequence in which interrupts occur, certain interrupts are processed, multi-ICE commands are executed, and the dispatcher is called. This sequence is external to the dispatcher.
- The dispatcher’s internal sequence of checking and processing certain interrupts and selecting the next task to dispatch.

The external sequence of interrupt processing and command execution can affect the dispatcher by altering a "dispatch table" between polls. The dispatch table is discussed in the next section.

The dispatcher’s internal sequence of interrupt processing and task selection is diagrammed in figure 9-6. The next several sections give the details on this sequence and how it interacts with the external sequence of interrupts and commands.
Figure 9-6. Dispatcher Functional Diagram
Multi-ICE Theory of Operation

Dispatch Table

The dispatcher refers to a table that contains the current task status (READY/NOT READY) and LOCK status of the HOST process, PR1, and PR2, and the task status of the HOST parser. The conditions that govern the task status of these four components are discussed earlier in this chapter; LOCK status must now be introduced to complete the picture of the dispatcher’s decision mechanism.

A process is LOCKed when it is executing or emulating from within a LOCK command block. The parser cannot be locked. When a locked process is dispatched, the process retrains control of the dispatching mechanism and no other process can be dispatched until the locked process becomes unlocked again. Details on how the dispatcher treats a locked process are given later in this chapter.

Table 9-7 summarizes the conditions that affect task status and LOCK. You can imagine that the table used by the dispatcher does not contain the conditions themselves as shown in table 9-7, but can contain TRUE or FALSE values depending on which combinations of conditions are currently TRUE when the dispatcher checks the table (polls a task).

<table>
<thead>
<tr>
<th>Task</th>
<th>READY</th>
<th>NOT READY</th>
<th>LOCKED</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST</td>
<td>• ACTIVE and not</td>
<td>• DORMANT.</td>
<td>• ACTIVE and executing or emulating from within a LOCK block.</td>
</tr>
<tr>
<td>Process</td>
<td>emulating.</td>
<td>• ACTIVE and emulating.</td>
<td></td>
</tr>
<tr>
<td>PR1</td>
<td>• ACTIVE and not</td>
<td>• DORMANT.</td>
<td>• ACTIVE and executing or emulating from within a LOCK block.</td>
</tr>
<tr>
<td></td>
<td>emulating.</td>
<td>• ACTIVE and emulating.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SUSPENDED.</td>
<td></td>
</tr>
<tr>
<td>PR2</td>
<td>• ACTIVE and not</td>
<td>• DORMANT.</td>
<td>• ACTIVE and executing or emulating from within a LOCK block.</td>
</tr>
<tr>
<td></td>
<td>emulating.</td>
<td>• ACTIVE and emulating.</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>• SUSPENDED.</td>
<td></td>
</tr>
<tr>
<td>Parser</td>
<td>• No process ACTIVE.</td>
<td>• Any process ACTIVE.</td>
<td>—</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Console and Hardware Interrupts

The system is prepared to process two categories of interrupts:
1. Interrupts from the console (spacebar, ESC, CTRL S, CTRL Q).
2. Interrupts from the hardware (emulation terminated or hardware error report).

Console Interrupts

Interrupts from the console are enabled unless an interrupt is currently being processed.

Spacebar

A space entered when the prompt is displayed is of course treated as a space character. To produce an interrupt, the spacebar must be pressed when the parser is suppressed by one or both ACTIVE ICE processes. Under this condition, pressing the spacebar invokes an interrupt routine that sets a “prompt” flag to TRUE. The
parser is set READY when “prompt” is TRUE and the HOST process is DORMANT. The dispatcher is not involved in setting the “prompt” flag or in setting the parser READY; these actions update the dispatch table without calling the dispatcher. However, the next time the dispatcher polls the parser and finds it READY, the parser is dispatched and the prompt is issued.

CTRL S, CTRL Q

Control S (CTRL S) and Control Q (CTRL Q) are also handled as console interrupts; they do not affect the dispatch table.

CTRL S halts all processing except emulation-in-progress and causes any display to pause between characters. CTRL Q continues any processing that was halted with CTRL S and lets the display continue with the next characters.

While the system is halted on CTRL S, the dispatcher cannot be called. As a result, no task can be dispatched. In addition, recognition of hardware interrupts is suppressed; thus, if an emulating process breaks emulation during a CTRL S halt, the “EMULATION TERMINATED” message cannot be displayed until you continue the system with CTRL Q.

ESC Key

While the prompt is displayed, the ESC key aborts the command that is being entered; ESC does not terminate the parser’s task, however, and another prompt is issued automatically. Pressing the ESC key while the parser is suppressed (some process is ACTIVE) invokes an interrupt routine that sets an “aborted” flag to TRUE. When the dispatcher is called, it checks the flag; if “aborted” is TRUE, the dispatcher calls an abort routine to perform the following actions:

- The HOST process is set DORMANT (NOT READY) if it is not already DORMANT.
- The message “HOST PROCESSING ABORTED” is displayed.
- If PR1 is ACTIVE (not suspended or dormant), the system displays the query “KILL PR1?”. To set PR1 DORMANT, enter “Y” followed by cr. To allow PR1 to remain ACTIVE press cr immediately (entering any character other than “Y” also allows PR1 to remain ACTIVE).
- If PR2 is ACTIVE, the system asks “KILL PR?” Enter “Y” cr to set PR2 DORMANT, or enter cr to allow PR2 to remain ACTIVE.

After performing these actions, the abort routine returns control to the dispatcher. The dispatch table has been updated to include the effects of the abort (HOST DORMANT) and of the user’s responses to the “KILL PRn?” queries. Specifically, if no process remains ACTIVE upon return, the parser has been set READY.

Hardware Interrupts

When an ICE hardware module breaks emulation or encounters an error condition, it informs the software of the condition through an interrupt.

Hardware interrupts are enabled only when the software is able to handle interrupt smoothly. One of these places is during the dispatcher’s internal sequence, since the dispatcher is called when a task is completed and the next task has not yet been dispatched. The effect is the same no matter where the interrupt is checked. This discussion assumes that the dispatcher is checking the hardware interrupts; figure 9-6 shows the place in the dispatcher’s internal sequence where the hardware interrupts are checked.
Emulation Terminated

Emulation terminates on a breakpoint, an error, or after a BREAK, SUSPEND, or KILL command. When the dispatcher checks the interrupt and detects any break other than a hardware error, it calls an interrupt routine to take the following actions:

- The message "EMULATION TERMINATED, PC = address" is displayed. The value of PC points to the address of the next instruction to be emulated; the address is displayed in the current BASE, or symbolically if SYMBOLIC displays are enabled (see chapter 4).
- The process and task status of the process that broke emulation are changed to new settings. The new setting depends on the condition that broke emulation and on the commands that remain to be executed in the process' command list, as discussed earlier in this chapter.

After these actions have been completed, control returns to the dispatcher; the dispatch table has been updated to reflect the new status of the process that broke emulation.

Hardware Errors

If a hardware error occurs during emulation or while any other command is being executed, the error is detected when the dispatcher checks for hardware interrupts. In this case, an error message is displayed to identify the error type to the user. When the error occurs during emulation, emulation terminates but no termination message is displayed; the status of the process is also updated to reflect the break in emulation. When a command other than emulation produced the error, the status of the process that executed the command is changed to DORMANT if its command list is exhausted; if it still has commands, it remains ACTIVE READY.

After displaying the error message and updating the dispatch table as necessary, the system returns control to the dispatcher to resume its internal sequence.

Referring to figure 9-6, the dispatcher reaches this point when it has finished checking for interrupts. Its next function is to determine whether any tasks are ready to be dispatched, and if so which task to dispatch.

Allocating Task Slices

A task slice is the segment of time required for a process or the parser to complete a task. When a process (HOST or ICE) is dispatched (allocated a task slice), it can execute one simple command, emulate one instruction in single-step mode, or enter real-time emulation. At the completion of any of these actions the process calls the dispatcher to allocate the next task slice. When the parser is dispatched, it issues a prompt and waits for a final cr. When the parser encounters a final cr, its action is completed and it calls the dispatcher to dispatch the next task. The dispatcher is also called after a parsing or execution error.

Current Process

At the time the dispatcher is called, the current process is the process (HOST, PR1, or PR2) most recently dispatched. The "value" of current process is not changed by any actions external to the dispatcher.

The dispatcher uses current process in two ways:

- As a marker indicating where the dispatching sequence left off, so that the dispatcher can check for a LOCK condition.
• As a moveable pointer into the first three rows of the dispatch table. If the current process is not LOCKed and the parser is not READY, the dispatcher continues its poll with the next process in its polling sequence for processes; we shall examine that sequence in a moment.

Current Process LOCKed
The Dispatcher first checks to see if the current process is LOCKed; a process is locked when it is executing commands in a LOCK command block. The exact syntax of the LOCK command is given in chapter 5. Its effect on the dispatcher is easy to describe: if the locked current process is READY, it is dispatched; if it is NOT READY (that is, emulating), the dispatcher loops back to begin checking for interrupts again. Thus, no other processes can be dispatched until the END of the lock command block is reached. If the locked process is emulating, nothing can occur until either that process breaks emulation or the user presses the ESC key. Pressing the spacebar has no effect, since the LOCK condition is checked before the parser can be dispatched (figure 9-6).

If the current process is not locked, the dispatcher proceeds to check the parser.

Parser READY
If the parser is READY when polled by the dispatcher, the parser is dispatched. If the parser is NOT READY, the dispatcher proceeds to check the three execution processes.

Polling Sequence for Processes
The dispatcher polls the three processes in a fixed sequence, diagramed in figure 9-7. For any given current process, the “next process” is defined as shown in table 9-8.

<table>
<thead>
<tr>
<th>If the current process is:</th>
<th>Then the next process is:</th>
</tr>
</thead>
<tbody>
<tr>
<td>HOST process</td>
<td>PR1</td>
</tr>
<tr>
<td>PR1</td>
<td>PR2</td>
</tr>
<tr>
<td>PR2</td>
<td>HOST process</td>
</tr>
</tbody>
</table>

Current Process READY
The dispatcher first updates current process as a pointer into the dispatch table; the process that was the next process becomes the current process. The dispatcher then looks at the dispatch table entry corresponding to that current process. If the current process is READY, it is dispatched. If the current process is NOT READY, the dispatcher updates the current process again (next process becomes current process) and repeats the READY test. If the process that is now the current process is READY, it is dispatched. If the current process is NOT READY, the dispatcher again updates the current process to point to the remaining process (the first two were NOT READY). If this process is READY, it is dispatched. If it is NOT READY (that is, no processes were READY) current process is updated once more, so that it now points to the process that was the current process when the dispatcher was called. Thus if no process is READY, current process in effect remains unchanged.

If nothing was dispatched, the dispatcher jumps back and begins checking for interrupts again.
Multi-ICE Theory of Operation

Summary

Dual-ICE operation involves the simultaneous testing of many conditions, as shown in the several status tables presented thus far. Rather than restating these conditions, we can summarize the main effects as follows (neglecting interrupts and LOCK for simplicity):

1. When no process is emulating or executing, the prompt is issued.
2. When only one process is executing, it receives every available task slice and, except for a slight delay for the dispatcher, behaves like a standard single ICE. When the process runs out of commands it stops and the system issues a prompt.
3. When one process is emulating, the system waits for emulation to break, again like a standard single ICE.
4. A process can be pre-empted after it enters real-time emulation so that another task can be dispatched.
5. Two different processes can emulate simultaneously (parallel emulation) if they are using different sets of hardware.
6. Two or more processes can have tasks to execute. Each process executes one action as it is assigned a task slice; after each action is completed, the next process in the sequence is assigned the next task slice.
This appendix contains a summary of the syntax of expressions in multi-ICE, the syntax of each of the commands, and an alphabetical list of all multi-ICE keywords.

Expressions

\[\text{expression} = \text{operand} \ [\text{binary-operator} \ \text{operand}]\ldots\]

\[\text{operand} = \text{primary} \ | \ (\text{expression} \ | \ \text{process-status} \ \text{process}) \ | \ \text{unary-operator} \ \text{operand} \ | \ (\text{operand}) \ | \ \text{environment-control} \ \text{primary} \ | \ \text{environment-control} \ (\text{expression})\]

\[\text{primary} = \text{numeric-constant} \ | \ \text{string-constant} \ | \ \text{symbolic-reference} \ | \ \text{statement-reference} \ | \ \text{keyword-reference}\]

\[\text{binary-operator} = + \ | \ - \ | \ * \ | \ MOD \ | \ MASK \ | \ = \ | \ > \ | \ \geq \ | \ < \ | \ \leq \ | \ AND \ | \ OR \ | \ XOR\]

\[\text{unary-operator} = + \ | \ - \ | \ \text{BYTE} \ | \ \text{IBYTE} \ | \ \text{CBYTE} \ | \ \text{DBYTE} \ | \ \text{XBYTE} \ | \ \text{WORD} \ | \ \text{IWORD} \ | \ \text{NOT}\]

\[\text{process-status} = \text{ACTIVE} \ | \ \text{DORMANT} \ | \ \text{SUSPENDED}\]

\[\text{process} = \text{HOST} \ | \ \text{PR1} \ | \ \text{PR2}\]

\[\text{environment-control} = \text{EN1} \ | \ \text{EN2}\]

Commands

1. Multi-ICE Commands
1A. Host-only (outside ACTIVATE block)
   - ACTIVATE \text{PRn cr}
   - \text{[command cr]...}
   - ENDACTIVATE
   - BREAK ALL
   - BREAK \text{PRn}
   - CONTINUE ALL
   - CONTINUE \text{PRn}
   - KILL ALL
   - KILL \text{PRn}
   - SUSPEND ALL
   - SUSPEND \text{PRn}
   - SWITCH = \text{ENn}
   - WAIT ANY
   - WAIT \text{PRn}
1B. ICE-only (inside ACTIVATE block)
   KILL
   SUSPEND

1C. Any process
   LOCK
   [command cr]...
   ENDLOCK
   SWITCH

2. Single-ICE commands, any process
2A. Block commands
   REPEAT cr
   [command cr
       UNTIL boolean-expression cr]
       WHILE boolean-expression cr
   ENDREPEAT

   COUNT count cr
   [command cr
       UNTIL boolean-expression cr]
       WHILE boolean-expression cr
   ENDCOUNT

   IF boolean-expression [THEN] cr
   [command cr]...
   [ORIF boolean-expression [THEN] cr]
   [command cr]...
   [ELSE cr]
   [command cr]...
   ENDIF

2B. Macro commands
   DEFINE MACRO macro-name
   [command cr]...
   EM
   :macro-name

   MACRO macro-name (macro-list?)
   DIRECTORY MACRO
   REMOVE MACRO [macro-list]
   PUT :drive:filename MACRO [macro-list]
   INCLUDE :drive:filename

2C. Display Commands
   ENABLE SYMBOLIC
   DISABLE SYMBOLIC
   EVALUATE expression SYMBOLIC
Multi-ICE

Summary of Multi-ICE Keywords and Commands

**BOOL boolean-expression**

**WRITE**

\[
\begin{align*}
\text{string} & \quad \text{expression} \\
\text{BOOL boolean expression} & \quad [\ldots]
\end{align*}
\]

\[
\begin{align*}
\text{keyword-reference} & \quad [\ldots] \\
\text{content-expression} & \quad [\ldots]
\end{align*}
\]

2D. IND Symbol-Table Commands

**DEFINE IND** .symbol-name = address / value

IND .symbol-name

SYMBOL IND

REMOVE IND .symbol-name

REMOVE SYMBOL IND

3. ICE-85-Only Commands and Keywords

**SEARCH** [DOUBLE] partition [WITH MASK mask-value] FOR target-value

3A. Commands

**DOMAIN**

DOMAIN = ..module-name

RESET DOMAIN

NESTING

LINES

MODULES

REMOVE MODULES

3B. Keywords

**FLAG**

**LIMIT**

**LOWER**
**Keywords**

Any keyword can be abbreviated to its first three characters.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTIVATE</td>
<td>LEVEL</td>
<td></td>
</tr>
<tr>
<td>ACTIVE</td>
<td>LIMIT</td>
<td>%</td>
</tr>
<tr>
<td>ALL</td>
<td>LINES</td>
<td>=</td>
</tr>
<tr>
<td>AND</td>
<td>LOCK</td>
<td>&gt;</td>
</tr>
<tr>
<td>ANY</td>
<td>LOWER</td>
<td>&lt;</td>
</tr>
<tr>
<td>BOOL</td>
<td>MACRO</td>
<td>&lt;=</td>
</tr>
<tr>
<td>BREAK</td>
<td>MASK</td>
<td>&gt;=</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>MODULES</td>
<td></td>
</tr>
<tr>
<td>COUNT</td>
<td>NESTING</td>
<td></td>
</tr>
<tr>
<td>DEFINE</td>
<td>NOT</td>
<td></td>
</tr>
<tr>
<td>DIRECTORY</td>
<td>OR</td>
<td></td>
</tr>
<tr>
<td>DOMAIN</td>
<td>ORIF</td>
<td></td>
</tr>
<tr>
<td>DORMANT</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td>PR1</td>
<td></td>
</tr>
<tr>
<td>EDGE</td>
<td>PR2</td>
<td></td>
</tr>
<tr>
<td>ELSE</td>
<td>PUT</td>
<td></td>
</tr>
<tr>
<td>ENDACTIVATE</td>
<td>REMOVE</td>
<td></td>
</tr>
<tr>
<td>ENDCOUNT</td>
<td>REPEAT</td>
<td></td>
</tr>
<tr>
<td>ENDF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENDLock</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENREPEAT</td>
<td>SEARCH</td>
<td></td>
</tr>
<tr>
<td>EN1</td>
<td>SINGLE</td>
<td></td>
</tr>
<tr>
<td>EN2</td>
<td>SUSPEND</td>
<td></td>
</tr>
<tr>
<td>FOR</td>
<td>SWITCH</td>
<td></td>
</tr>
<tr>
<td>FOR</td>
<td>SYMBOLIC</td>
<td></td>
</tr>
<tr>
<td>HOST</td>
<td>UNTIL</td>
<td></td>
</tr>
<tr>
<td>IF</td>
<td></td>
<td></td>
</tr>
<tr>
<td>INCLUDE</td>
<td>WAIT</td>
<td></td>
</tr>
<tr>
<td>IND</td>
<td>WHILE</td>
<td></td>
</tr>
<tr>
<td>KILL</td>
<td>XOR</td>
<td></td>
</tr>
</tbody>
</table>
APPENDIX B
INSTALLATION PROCEDURES FOR
INTELLEC SERIES II SYSTEMS

This appendix contains procedures for installing two ICE hardware modules (ICE-85, ICE-49, or ICE-41A) in one Intellec Series II Microcomputer Development Systems, Models 220 and 230.

NOTE
To install dual-ICE in an Intellec MDS-800 system, follow the procedures for each ICE given in the standard ICE manual for that product. The MDS-800 chassis has enough extra slots for the two pairs of circuit boards.

The procedure for Intellec Models 220 and 230 is as follows:
1. Install expansion chassis following procedure in the Intellec Series II Installation and Service Manual. Do not replace the front panels yet.
2. Figure B-1 shows the recommended locations for the two ICEs (two boards for each ICE) in a Model 220 or 230 system. One ICE module is in the main chassis and the other is in the expander chassis. Both occupy the middle two slots so that the flat ribbon cables can fit inside the chassis with the front cover installed.
3. Set the device codes on the two ICE controller boards to identify PRI and PR2. Table B-1 shows the device codes to use for various combinations of ICEs. The ICE with the lower numbered device code is PRI in all combinations.

Table B-1. Multi-ICE™ Device Codes

<table>
<thead>
<tr>
<th>Combination</th>
<th>PR1</th>
<th>PR2</th>
</tr>
</thead>
<tbody>
<tr>
<td>85/85</td>
<td>10H</td>
<td>11H</td>
</tr>
<tr>
<td>85/49</td>
<td>10H</td>
<td>23H</td>
</tr>
<tr>
<td>85/41A</td>
<td>10H</td>
<td>24H</td>
</tr>
</tbody>
</table>

In general, when the two ICEs are different (e.g., 85/49 or 85/41A) the standard device code for each ICE should be used. You should verify the device code settings; refer to the standard ICE manuals for details.

4. The multi-ICE package includes three replacement PROMs for one ICE-85 Controller Board. Install the three PROMs as shown in table B-2. Do not replace the PROM in socket A11.

Table B-2. ICE-85™ Replacement PROM Locations

<table>
<thead>
<tr>
<th>Socket Number</th>
<th>Old Part Number</th>
<th>New Part Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>A8</td>
<td>9100139</td>
<td>9100229</td>
</tr>
<tr>
<td>A9</td>
<td>9100140</td>
<td>9100230</td>
</tr>
<tr>
<td>A10</td>
<td>9100141</td>
<td>9100231</td>
</tr>
</tbody>
</table>
Figure B-1. Intellec® Series II Models 220/230 Dual-ICE™ Installation
5. Insert a pair of ICE boards into an iSBC dual auxiliary connector (P/N 1000751). Refer to figure B-2 for a diagram showing the orientation of the two boards and the locations of the four ribbon cable connectors X, Y, V, and T.

a. For a permanent installation, the dual auxiliary connector can be bolted to the backplane before installing the ICE boards.

6. Insert the boards and connector into the main chassis, in the middle two slots as shown in figure B-1.

NOTE

The cable slots at the right side of the main chassis are larger than the corresponding slots on the expander chassis. Inspect the flat ribbon cables and install the ICE with the wider cables in the main chassis.

7. Attach the ribbon cables from the cable modules (X to X, V to V, Y to Y, and T to T) as shown in figure B-3. Hold cables X, V, and Y together with the ribbed side toward the main chassis. Insert the connectors, then fold the cables to the right. Guide the ribbon connector with connector T (from the trace module) around the left end of cable Y and make the connection.

8. Guide the flat cables through the slot at the right side of the main chassis and replace the front panel on the main chassis.

9. Repeat steps 5 through 9 to install the other ICE in the expander chassis. Curl the flat cables as necessary to fit the exit slot at the right of the chassis.

10. Apply power to the system, load diskettes, boot ISIS-II, and invoke the multi-ICE software. The installation is complete.

---

Figure B-2. ICE™ Boards in Dual Auxiliary Connector
Figure B-3. Ribbon Cable Routing Diagram
This appendix contains a listing and explanation of the ICE error messages that are particular to the commands in this manual or that have a different interpretation under these commands from that given for the standard ICE commands. Refer to the standard ICE manuals for explanations of other ICE error messages.

**ERR 88: MACRO PARAMETER ERROR**

A macro call contained more than ten actual parameters. Enter the command with ten or fewer actual parameters.

**ERR 90: MEMORY OVERFLOW**

ICE workspace has expanded to the maximum permitted by the value of LIMIT. This can happen when the symbol table grows very large and when a macro expansion requires more workspace than that available. The command that produced the overflow is aborted, but the memory already written in the Intellec by that command is not restored. Memory below LIMIT is not changed.

**ERR 96: INVALID WITHIN ACTIVATE**

An ACTIVATE block may not contain a HOST-only command (WAIT PRn, WAIT ANY, CONTINUE PRn, CONTINUE ALL, SUSPEND PRn, SUSPEND ALL, KILL PRn, KILL ALL, BREAK PRn, BREAK ALL), another ACTIVATE command, a macro definition command, or a SWITCH command that refers to the other ICE environment (e.g., SWITCH = PR2 inside an ACTIVATE PRI block). The invalid command is ignored, but the ACTIVATE block is not aborted. Enter another command.

**ERR 9F: PROCESS ALREADY ACTIVE**

The process named in an ACTIVATE command is already executing or emulating a command list from an ACTIVATE block. The extra ACTIVATE command is ignored. Enter another command, or KILL the active process before entering another ACTIVATE command for that process.

**ERR A3: PROCESS DORMANT**

The process named in a CONTINUE command is DORMANT. The CONTINUE command is ignored. Enter another command.

**ERR A4: MACRO FILE FULL**

The temporary file MAC.TMP on the multi-ICE diskette has used all the available space on that diskette, and there is no room for any more macro definitions. Save and remove one or more macros to make room for more, using the PUT MACRO and REMOVE MACRO commands in multi-ICE.
**ERR AF: ILLEGAL HOST COMMAND**

A SUSPEND or CONTINUE command was entered to the HOST process (that is, not within an ACTIVATE block) that either had no process name or had an unrecognizable process name following the initial command keyword. The command is ignored; enter another command.

**ERR B0: LIMIT HIGHER THAN UPPER**

LIMIT, the lowest address available to the ICE memory manager for expanding ICE workspace, cannot be set to a higher address than the value of UPPER, the lowest address currently used by ICE workspace. The command that attempted to change LIMIT was ignored. Refer to chapter 5, ICE-85 Dependent Commands, for details on LIMIT and UPPER.

**ERR B1: INVALID WITH LOCK ON**

A LOCK block may not contain a SUSPEND or WAIT command. The invalid command was ignored, but the LOCK block is not aborted. Enter another command.

**ERR B4: POTENTIAL BUS LOCKOUT**

In a dual ICE-85 system both ICE-85’s may not be mapped to Intellec memory. Two ICE-85’s emulating (GO FOREVER) from Intellec memory can produce a bus lockout condition requiring a hardware reset in the Intellec system, unless one of them can break emulation. This condition can be avoided by mapping at least one ICE-85 to USER.

**WARN C1: MAPPING OVER SYSTEM**

Under ICE-85, you are warned when memory mapped to INTELLEC contains addresses in one of the following areas of memory:

- Monitor (highest block)
- ICE workspace (UPPER points to lowest address in ICE workspace).
- Potential ICE workspace (LIMIT points to the lowest address available for expanding ICE workspace).
- ICE software (LOWER points to the lowest address in the next free block higher than the ICE software).
- ISIS-II (the lowest blocks).

Initially, LIMIT = LOWER; any memory mapped to INTELLEC at this time receives a warning. You can reset LIMIT to the highest address occupied by user code; if LIMIT is reset before the area is mapped, the warning is not issued. The warning has no effect on the command.

**WARN C2: HARDWARE MISSING**

This message is preceded by the device code that Multi-ICE was looking for. This warning indicates that Multi-ICE cannot communicate with the indicated ICE hardware. Most commonly, the hardware is not installed in the Intellec chassis. Alternatively, check the device code setting for the ‘missing’ ICE (see Appendix B). The warning is not fatal. The ICE software for the missing ICE can still be run, but of course no hardware commands can execute.
WARN C3: MULTIPLE HARDWARE

Both ICEs have the same device code setting. Reset the device code setting (see Appendix B) of one of the ICEs and re-install.
This appendix contains suggestions on operating multi-ICE, procedures for backing up the multi-ICE PROMs, and brief descriptions of known limitations of this product when compared to standard ICE systems.

1. Perform all LOAD, SAVE, LIST, and MAP commands through the HOST rather than within ACTIVATE, SWITCH environments as needed.
2. Enter all macro definitions through the HOST. Macros can be called by any process, and are not limited to the current environment at definition time.
3. While an ICE process is in emulation the HOST can be in interrogate mode in the same environment. Avoid commands that affect emulation and trace while in this situation; these commands include those that affect the MAP, the GO-register, breakpoints, the trace buffer, the program counter, qualifier registers, code memory, and hardware registers. The result of any of these commands in this situation will not be useful to you.

4. **Backing up Replacement PROMs for Testing**

Using an Intel UPP PROM programmer and 2716 personality module attached to the Intellec system you can transfer the contents of the three replacement PROMs furnished with multi-ICE to diskette for comparison in case of a suspected PROM failure.

To ISIS-II enter the command sequence (prompts are furnished by the system):

```
-UPM
*TYPE*2716
```

Insert each PROM in the programmer socket in turn, and transfer its contents with the appropriate transfer command as shown in table D-1.

<table>
<thead>
<tr>
<th>PROM Installed</th>
<th>Transfer Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>9100229</td>
<td>TRANSFER FROM 0 TO 7FFH</td>
</tr>
<tr>
<td>9100230</td>
<td>TRANSFER FROM 800H TO 0FFH</td>
</tr>
<tr>
<td>9100231</td>
<td>TRANSFER FROM 1000H TO 17FFH</td>
</tr>
</tbody>
</table>

Finally, enter the command:

```
*WRITE FILE :F1:85 PROM.BAK FROM 0 TO 17FFH HEX
```

This completes the backup procedure. To make the comparison test, enter the command:

```
*READ FILE :F1:85PROM.BAK INTO 0
```

Insert each PROM in the programmer socket in turn, and compare its contents with the backup as shown in table D-2.

<table>
<thead>
<tr>
<th>PROM Installed</th>
<th>Compare Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>9100229</td>
<td>COMPARE FROM 0 TO 7FFH</td>
</tr>
<tr>
<td>9100230</td>
<td>COMPARE FROM 800H TO 0FFH</td>
</tr>
<tr>
<td>9100231</td>
<td>COMPARE FROM 1000H TO 17FFH</td>
</tr>
</tbody>
</table>
This completes the comparison procedure.

5. Under ICE-85, use of the monitor routines CO and C1 for console display from within user code interferes with Multi-ICE’s interrupt handling routines. Thus, you should not map I/O to the Intellic system.

Instead, use the WRITE command to have ICE produce simulated output whenever a display is required.

6. Under ICE-85, the EXECUTE command is omitted in Multi-ICE. Use the INCLUDE command instead.

7. Under ICE-85, the performance characteristics of the SY1 line do not allow you to synchronize trace collection between two ICES.

8. Under ICE-85, the replacement PROMs furnished with Multi-ICE change the performance of SY0 OUT by adding two control keywords, EDGE and LEVEL. The syntax of the ENABLE SY0 OUT command becomes:

   \texttt{ENABLE SY0 OUT [EDGE] \[LEVEL\]}

LEVEL is the default, and is the performance condition for the standard ICE-85. Under this condition, SY0 goes from high to low within 1.3-2.3 ms after a breakpoint register matches.

When EDGE is specified, the SY0 line goes from high to low within 30-200 \(\mu\)s after a breakpoint match, stays low for a brief period, returns high to finish the last instruction (and some internal ‘bookkeeping’), then goes low and stays low. EDGE is useful when the external device to be controlled by SY0 OUT is edge-triggered, or when you wish to have a faster response than LEVEL can give. Specifically, EDGE should be used when the external device is another ICE; the external ICE halts emulation immediately on receiving the first high-to-low edge. (That is, halts 30-200 \(\mu\)s after the sending ICE has halted emulation.)

When SY0 is used to control the start of emulation, emulation begins approximately 800 \(\mu\)s after SY0 goes high. This brief delay is characteristic of both the sending ICE and the receiving ICE; thus, SY0 does not produce a simultaneous start of emulation by two ICES.

When SY0 out is enabled in the sending ICE-85, a reference using BYTE, WORD, or PORT, or a single STEP emulation produces a momentary pulse on SY0 OUT. If the receiving ICE has SY0 IN enabled, it will start emulation then halt on the first falling edge.

With SY0 OUT enabled, a RESET HARDWARE command by the sending ICE-85 causes SY0 to go high and remain high for 1.8 seconds.

9. Under ICE-85, the CAUSE command does not give the correct cause of breaking emulation.

10. If the Multi-ICE software is invoked from a SUBMIT file under ISIS-II V3.4, the control E feature for switching input between the console and the SUBMIT file is not fully supported.

11. ICE-85 containing Multi-ICE firmware now supports a Hold/Hold Acknowledge protocol while not in emulation. The EMUL (active HIGH) signal provided in the buffer assembly indicates when the Hold/Hold Acknowledge protocol is fully supported. When EMUL is false (LOW), ICE-85 may not respond to a Hold Request with a Hold Acknowledge for up to 1 msec. Thus, if ICE-85 requires the use of the user system bus to retrieve or restore emulation data, it sets EMUL low indicating that the Hold/Hold Acknowledge protocol is not fully supported.

12. Under ICE-85, the operations of the EMUL output differs from the standard ICE-85 output. In the standard ICE-85 EMUL indicates emulation in progress, while in Multi-ICE-85 EMUL indicates that ICE-85 does not have control of the user system bus.
ACTIVATE command, 2-5, 2-6, 2-7, 6-1, 6-2 to 6-4, 6-5, 9-12
ACTIVATE list
See Command list
ACTIVE status, 6-3, 9-3, 9-9, 9-10, 9-11, 9-12
ALL keyword, 2-5, (see SUSPEND, KILL, CONTINUE, and BREAK commands)
AND operator, 3-5, 3-7
ANY keyword
See WAIT command
Arithmetic operators, 3-4, 3-5, 3-6
Binary Operators, 3-4, 3-6, 3-11ff
BOOL command, 4-1, 4-25, 9-3
Boolean expression, 3-16, 3-17
BREAK command, 8-1, 8-2, 9-12, 9-13, 9-14
Classes of operators, 3-4, 3-5, 3-6
Command buffer, 9-1, 9-2, 9-5, 9-10
Command code buffer
See Command buffer
Command contexts, 3-16, 3-17
Command list, 9-2, 9-11, 9-12, 9-13
Compound commands, 4-1, 9-6
Content-operator, 3-5, 3-6, 3-7
CONTINUE commands, 7-2, 7-3, 7-4, 9-14
COUNT command, 4-1, 4-5 to 4-8
CTRL Q, 9-17
CTRL S, 9-17
Current process, 9-18 to 9-20
Current SWITCH, see Environment
DEFINE MACRO command, 4-12, 4-14 to 4-16
Direct references, 3-3, 3-4
Dispatcher, 9-14 to 9-20
Dispatch table, 9-16
Display macro command, 4-1, 4-13, 4-18, 4-19
Display macro directory command, 4-1, 4-13, 4-19
DOMAIN commands, 5-1, 5-3
DORMANT status, 6-3, 6-5, 9-3, 9-9, 9-11, 9-12
DOUBLe
See SEARCH command
EDGE keyword, D-2
EN1, EN2
See Environment
Environment, 2-5, 6-1, 6-3, 6-4, 9-1, 9-3, 9-4, 9-11
Environment controls, 3-8, 6-4
Error messages, C-1 to C-3
ESC key, 2-7, 6-3, 6-5, 9-9, 9-17
Evaluating expressions, 3-9 to 3-16
Execution process, 9-2
Expression, 3-1
FLAG keyword, 5-4
Hardware interrupts, 9-16, 9-17, 9-18
HOST execution process
See HOST process
HOST-only commands, 1-3, 9-8
HOST parser, 9-1, 9-3, 9-5 to 9-8, 9-15, 9-16, 9-19
HOST process, 2-5, 2-6, 6-3 to 6-6, 9-1, 9-2, 9-3, 9-8 to 9-11, 9-19, 9-20
ICE-independent commands, 4-1, 9-8, 9-11
ICE process, 2-5, 6-3, 6-4, 6-5, 9-2, 9-3, 9-11 to 9-14, 9-18, 9-19, 9-20
IF command, 4-1, 4-8 to 4-10
INCLUDE command, 4-1, 4-14, 4-20
IND symbol commands, 4-1, 4-27
Installation, 2-1, 2-2, 2-5, Appendix B
Intermediate carriage return, 9-6
Invoking ICE software, 1-1, 2-5
Keyword reference, 3-3
KILL commands, 2-5, 2-7, 2-8, 6-2, 6-5, 9-11, 9-12, 9-13
LEVEL keyword, D-2
LIMIT keyword, 5-4 to 5-8
LINES command, 5-3
Local and global defaults, 4-1, 4-2
LOCK command, 8-1, 8-2, 8-3, 9-15, 9-16, 9-19
LOCKED status, 9-15, 9-16, 9-19
Logical operators, 3-5, 3-6, 3-7, 3-8
LOWER keyword, 5-4 to 5-8
Macro call command, 4-1, 4-12, 4-14ff
MACRO command (display macro), 4-1, 4-13, 4-18 to 4-19
Macro commands, 4-11 to 4-25
Macro invocation, see Macro call command
MASK operator, 3-5, 3-6
Messages, 6-6
MOD operator, 3-5, 3-6
MODULES command, 5-4
Multiple displays, 4-1, 4-27
NESTING command, 5-3
Nesting compound commands, 4-10, 4-11
NOT operator, 3-5, 3-7
NOT READY
See Task Status
Numbers, 3-1
Numeric constants, 3-2
Numeric expression, 3-16, 3-17
Operands, 3-1
Operators, 3-4 to 3-8, 3-9
OR operator, 3-5, 3-8

Parsing and execution environment
See Environment
Parser
See HOST parser
PR1, PR2
See ICE process
Precedence of operators, 3-4, 3-5, 3-6, 3-9 to 3-16
Primaries, 3-1
Process references, 3-4
Process status, 3-4, 6-3, 9-3, 9-9, 9-10, 9-11 to 9-14
PROGI (sample user program), 2-1, 2-2, 2-3, 2-6
PROG2, 2-1, 2-4, 2-6, 2-8
Prompt, 9-5; see HOST parser
PUT macro command, 4-1, 4-14, 4-20

READY
See Task status
Relational operators, 3-5, 3-6, 3-7
REMOVE MACRO command, 4-1, 4-13, 4-18, 4-19
REMOVE MODULE command, 5-4

REPEAT command, 4-1, 4-2 to 4-5
Replacement PROMs, 1-1, B-1, D-1

SEARCH command, 5-1 to 5-3
SINGLE
See SEARCH command
Spacebar, 2-7, 6-4, 9-5, 9-16, 9-17
Statement-number references, 3-4, 4-28
String constants, 3-2, 3-3
SUSPEND command, 7-1, 7-3, 7-4, 9-11, 9-12, 9-13, 9-14
SUSPENDED status, 7-1, 7-2, 7-3, 9-3, 9-11, 9-12, 9-13, 9-14
SWITCH commands, 2-5, 6-1, 6-3, 6-4, 6-5
Symbolic displays, 4-1, 4-28, 4-29
Symbolic references, 3-3, 4-27, 4-28

Tasks, 9-4, 9-14, 9-18
Task-slice, 9-4, 9-14, 9-15, 9-18 to 9-20
Task status, 9-4, 9-6, 9-7, 9-8, 9-9, 9-10 to 9-14, 9-19

Unary operators, 3-4, 3-6, 3-11ff

WAIT command, 7-2, 7-3, 7-4, 9-10, 9-11
WRITE command, 4-1, 4-26, 4-27

XOR operator, 3-5, 3-6, 3-8
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. □
LIKE YOUR COMMENTS...

document is one of a series describing Intel products. Your comments on the back of this form will produce better manuals. Each reply will be carefully reviewed by the responsible person. All ent and suggestions become the property of Intel Corporation.

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1040 SANTA CLARA, CA

POSTAGE WILL BE PAID BY ADDRESSEE

Intel Corporation
Attn: Technical Publications M/S 6-2000
3065 Bowers Avenue
Santa Clara, CA 95051