TEXAS INSTRUMENTS

*Improving Man's Effectiveness Through Electronics*

# Model 990 Computer
# TI Pascal User's Manual

MANUAL NO. 946290-9701
ORIGINAL ISSUE 1 MAY 1978
REVISED 15 JANUARY 1979

**Digital Systems Division**

## LIST OF EFFECTIVE PAGES

INSERT LATEST CHANGED PAGES DESTROY SUPERSEDED PAGES

Note: The portion of the text affected by the changes is indicated by a vertical bar in the outer margins of the page.

Model 990 Computer TI Pascal User's Manual (946290-9701)

Original Issue . . . . . . . . . . . . . . . . . . . . . . .1 May 1978
Revised . . . . . . . . . . . . . . . . . . . . . . . . . . 15 January 1979 (ECN 004356)

Total number of pages in this publication is 394 consisting of the following:

| PAGE NO. | CHANGE NO. | PAGE NO. | CHANGE NO. | PAGE NO. | CHANGE NO. |
|---|---|---|---|---|---|
| Cover | 0 | B-1 - B-6 | 0 | | |
| Effective Pages | 0 | Appendix C Div | 0 | | |
| iii - xiv | 0 | C-1 - C-6 | 0 | | |
| 1-1 - 1-6 | 0 | Appendix D Div | 0 | | |
| 2-1 - 2-4 | 0 | D-1 - D-28 | 0 | | |
| 3-1 - 3-6 | 0 | Appendix E Div | 0 | | |
| 4-1 - 4-2 | 0 | E-1 - E-16 | 0 | | |
| 5-1 - 5-22 | 0 | Appendix F Div | 0 | | |
| 6-1 - 6-36 | 0 | F-1 - F-12 | 0 | | |
| 7-1 - 7-2 | 0 | Appendix G Div | 0 | | |
| 8-1 - 8-24 | 0 | G-1 - G-2 | 0 | | |
| 9-1 - 9-10 | 0 | Appendix H Div | 0 | | |
| 10-1 - 10-12 | 0 | H-1 - H-6 | 0 | | |
| 11-1 - 11-6 | 0 | Appendix I Div | 0 | | |
| 12-1 - 12-46 | 0 | I-1 - I-8 | 0 | | |
| 13-1 - 13-6 | 0 | Alphabetical Index Div | 0 | | |
| 14-1 - 14-46 | 0 | Index-1 - Index-14 | 0 | | |
| 15-1 - 15-22 | 0 | User's Response | 0 | | |
| Appendix A Div | 0 | Business Reply | 0 | | |
| A-1 - A-8 | 0 | Cover Blank | 0 | | |
| Appendix B Div | 0 | Cover | 0 | | |

## PREFACE

This manual describes the TI Pascal programming language as it is implemented on the Model 990 Computer. This manual is intended to show a programmer familiar with another language how to use TI Pascal (TIP). It also serves as a reference to which a programmer may refer to resolve questions about the language. It is not meant to be adequate to serve as a tutorial for those unfamiliar with programming.

Throughout the manual TI Pascal is compared to the version of Pascal described in *Pascal User Manual and Report,* K. Jensen and N. Wirth, Springer-Verlag, 1975.

The manual is organized into fifteen sections and nine appendixes including:

| | |
|---|---|
| I | Introduction — Provides an overview of the language, lists the extensions and modifications specific for TI Pascal, and compares the language to FORTRAN and to PL/I. |
| II | Overview — Provides an overview of language elements and of the program and data structure of Pascal. The concept of scope is introduced, and an example program is presented and described. |
| III | Notation and Vocabulary — Describes the notation used in the manual, including BNF productions and syntax diagrams. Defines the language elements. |
| IV | The Assignment Statement — Describes the assignment statement and the expression used in the assignment statement. |
| V | Enumeration Types and Related Statements — Describes the enumeration types INTEGER, LONGINT, and BOOLEAN. Also describes machine-dependent enumerations types CHAR, REAL, FIXED, and DECIMAL. Describes the user-defined enumeration types, scalar and subrange. Describes the ASSERT, Compound, IF, WHILE, REPEAT, CASE, and FOR statements. |
| VI | Structured Data Types — Describes the structured data types and the statements related to these types. Describes the ARRAY, RECORD, SET, and FILE types, and the procedures used to manipulate these types. Describes the WITH statement and the POINTER type. Also discusses type compatibility and type transfer. |
| VII | Jump Statements — Describes the ESCAPE statement and the GOTO statement, the two jump statements of the language. |
| VIII | The Program and Its Routines — Describes the TI Pascal program structure in detail, and the declarations required. Describes the LABEL, CONSTANT, TYPE, VARIABLE, COMMON, ACCCESS, PROCEDURE, and FUNCTION declarations. Describes the methods of parameter reference provided, and describes the scope and extent of variables and routines. |
| IX | Compiler Options — Describes the options available in the compiler and the option comment that specifies an option. |
| X | Formatting Source Code — Describes source code preparation and the source formatter utility, NESTER. |

| | |
|---|---|
| XI | The TIP Compiler — Describes SILT1, SILT2, and CODEGEN, the three phases of the TIP compiler, and describes the handling of errors in source code by the compiler. |
| XII | Separate Compilation — Describes the problems of separately compiling one or more modules of a program, and the configuration processor, a utility that assists in separate compilation. Also describes the split program utility and the split object utility, which also relate to separate compilation. |
| XIII | Reverse Assembler — Describes the reverse assembler RASS and its output. |
| XIV | Linking and Executing Pascal Tasks — Describes the options for linking Pascal tasks. Describes the debug commands provided for Pascal tasks. Also describes the types of runtime errors and the abnormal termination dumps provided. |
| XV | Computer Dependent Source Code Preparation Techniques — Describes routines that a user program may call for direct CRU I/O and operating system interface. Also describes guidelines for preparing source code for each linking option. |
| A | TIP Standard Routines — Describes the standard procedures and functions included with TIP software. |
| B | Example Programs — Describes four example programs. |
| C | System Dependent Information for DX10 Release 3 — Describes the implementation of Pascal files and device I/O provided by DX10. |
| D | TIP Syntax — Summarizes the BNF productions and syntax diagrams from the entire manual. |
| E | Error Messages — Lists the compiler and runtime error messages, the error codes, and the error messages provided by the Pascal debug commands. |
| F | Assembly Language Routines — Describes the interface required for routines written in assembly language, including task termination routines. |
| G | ASCII Character Set — Lists the characters of the ASCII character set with the hexadecimal and decimal values corresponding to each character. |
| H | Overlays in TIP Programs — Describes the overlay structure available to the user for TIP programs, and how to use it. |
| I | Type Conversions — Describes the implied conversions between types and the methods of performing type conversions, implied and explicit. |

The following documents contain additional information related to TI Pascal:

| Title | Part Number |
|---|---|
| *Model 990 Computer TMS 9900 Microprocèssor Assembly Language Programmer's Guide* | 943441-9701 |
| *Model 990 Computer DX10 Operating System Release 3 Reference Manual, Vols. 1-6* | 946250-9701, -9702, -9703, -9704, -9705, -9706 |
| *Model 990 Computer Link Editor Reference Manual* | 949617-9701 |
| *Model 990 Computer TX990 Operating System Programmer's Guide* (*Release 2*) | 946259-9701 |
| *Model 990/10 Computer RX990 Operating System Programmer's Guide* | 2250065-9701 |

The following book describes the Pascal language and is referenced in this manual:

*Pascal User Manual and Report,* K. Jensen and N. Wirth, Springer-Verlag, 1975

## TABLE OF CONTENTS

| Paragraph | Title | Page |
|---|---|---|
| | SECTION I. INTRODUCTION | |
| 1.1 | Texas Instruments Pascal | 1-1 |
| 1.2 | Extension and Modifications to Pascal | 1-1 |
| 1.3 | Comparisons with FORTRAN | 1-3 |
| 1.4 | Comparisons with PL/I | 1-5 |
| 1.5 | Organization of the Manual | 1-5 |
| | SECTION II. OVERVIEW | |
| 2.1 | Program Structure | 2-1 |
| 2.2 | Data Types | 2-1 |
| 2.2.1 | Simple Types | 2-2 |
| 2.2.2 | Structured Types | 2-3 |
| 2.2.3 | Storage Requirements | 2-3 |
| 2.3 | Declarations | 2-3 |
| 2.4 | Statements | 2-3 |
| 2.5 | Example Program | 2-4 |
| | SECTION III. NOTATION AND VOCABULARY | |
| 3.1 | Syntax Notation | 3-1 |
| 3.2 | Basic Syntax Elements | 3-2 |
| 3.2.1 | Identifiers | 3-3 |
| 3.2.2 | Numbers | 3-4 |
| 3.2.3 | Strings | 3-5 |
| 3.3 | Separators | 3-5 |
| | SECTION IV. THE ASSIGNMENT STATEMENT | |
| 4.1 | Assignment Statement | 4-1 |
| 4.2 | Operators and Expressions | 4-2 |
| | SECTION V. ENUMERATION TYPES AND RELATED STATEMENTS | |
| 5.1 | Enumeration Types | 5-1 |
| 5.2 | INTEGER and LONGINT Types | 5-1 |
| 5.3 | BOOLEAN Type | 5-2 |
| 5.4 | The ASSERT Statement | 5-2 |
| 5.5 | The Compound Statement | 5-3 |
| 5.6 | Conditional and Repetitive Statements | 5-4 |
| 5.6.1 | The IF Statement | 5-4 |
| 5.6.2 | The WHILE Statement | 5-6 |
| 5.6.3 | The REPEAT Statement | 5-6 |
| 5.7 | Machine Dependent and User-Defined Enumeration Types | 5-7 |

## TABLE OF CONTENTS (Continued)

| Paragraph | Title | Page |
|---|---|---|
| 5.7.1 | CHAR Type | 5-7 |
| 5.7.2 | SCALAR Type | 5-8 |
| 5.7.3 | SUBRANGE Type | 5-9 |
| 5.8 | CASE and FOR Statements | 5-10 |
| 5.8.1 | The CASE Statement | 5-10 |
| 5.8.2 | The FOR Statement | 5-11 |
| 5.9 | Expressions | 5-13 |
| 5.10 | REAL Type | 5-15 |
| 5.11 | FIXED Type | 5-19 |
| 5.12 | DECIMAL Type | 5-21 |

SECTION VI. STRUCTURED DATA TYPES

| | | |
|---|---|---|
| 6.1 | Array Type | 6-1 |
| 6.1.1 | Array Procedures | 6-3 |
| 6.1.2 | Static and Dynamic Arrays | 6-4 |
| 6.2 | Record Type | 6-4 |
| 6.2.1 | Records | 6-4 |
| 6.2.2 | Variants | 6-7 |
| 6.3 | WITH Statement | 6-8 |
| 6.4 | SET Type | 6-10 |
| 6.5 | FILE Type | 6-12 |
| 6.5.1 | Sequential Files | 6-14 |
| 6.5.2 | Textfiles | 6-16 |
| 6.5.3 | Formatted I/O with Textfiles | 6-20 |
| 6.5.4 | ENCODE and DECODE | 6-23 |
| 6.5.5 | Random Files | 6-23 |
| 6.5.6 | Alternate Forms | 6-24 |
| 6.6 | Pointer Type | 6-25 |
| 6.6.1 | Static and Dynamically Allocated Variables | 6-26 |
| 6.7 | PACKED Data Types | 6-29 |
| 6.7.1 | Packed Arrays | 6-29 |
| 6.7.2 | Packed Records | 6-30 |
| 6.7.3 | Internal Representation of Types | 6-30 |
| 6.7.4 | The SIZE Function | 6-33 |
| 6.8 | Type Compatibility | 6-33 |
| 6.9 | Type Transfer | 6-34 |

SECTION VII. JUMP STATEMENTS

| | | |
|---|---|---|
| 7.1 | The ESCAPE Statement | 7-1 |
| 7.2 | The GOTO Statement | 7-2 |

SECTION VIII. THE PROGRAM AND ITS ROUTINES

| | | |
|---|---|---|
| 8.1 | The TIP Program | 8-1 |
| 8.2 | Declarations | 8-2 |
| 8.2.1 | Label Declaration | 8-2 |
| 8.2.2 | Constant Declaration | 8-2 |

## TABLE OF CONTENTS (Continued)

| Paragraph | Title | Page |
|---|---|---|
| 8.2.3 | Type Declaration | 8-2 |
| 8.2.4 | Variable Declaration | 8-4 |
| 8.2.5 | Common Declaration | 8-4 |
| 8.2.6 | Access Declaration | 8-4 |
| 8.2.7 | Routine Declarations | 8-4 |
| 8.3 | Statements | 8-7 |
| 8.4 | Procedure and Function Calls | 8-8 |
| 8.5 | Parameters | 8-10 |
| 8.5.1 | Rules for Parameters | 8-13 |
| 8.5.2 | Dynamic Array and Set Parameters | 8-14 |
| 8.5.3 | Procedure and Function Parameters | 8-15 |
| 8.5.4 | Procedures and Functions without Parameters | 8-15 |
| 8.6 | Block Structure | 8-16 |
| 8.7 | Scope and Extent | 8-17 |
| 8.8 | Side Effects | 8-21 |
| 8.9 | Common and Access Declarations | 8-22 |
| | SECTION IX. COMPILER OPTIONS | |
| 9.1 | General | 9-1 |
| 9.2 | The Option Comment | 9-1 |
| 9.3 | Listing Control Options | 9-3 |
| 9.3.1 | List Option | 9-3 |
| 9.3.2 | Widelist Option | 9-3 |
| 9.3.3 | Map Option | 9-3 |
| 9.3.4 | Page Option | 9-4 |
| 9.3.5 | Warnings Option | 9-4 |
| 9.4 | Object Code Options | 9-4 |
| 9.4.1 | Object Option | 9-4 |
| 9.4.2 | Nullbody Option | 9-4 |
| 9.4.3 | Traceback Option | 9-6 |
| 9.4.4 | Asserts Option | 9-6 |
| 9.5 | Runtime Check Options | 9-6 |
| 9.5.1 | CKINDEX Option | 9-6 |
| 9.5.2 | CKOVER Option | 9-6 |
| 9.5.3 | CKPREC Option | 9-6 |
| 9.5.4 | CKPTR Option | 9-6 |
| 9.5.5 | CKSET Option | 9-6 |
| 9.5.6 | CKSUB Option | 9-6 |
| 9.5.7 | CKTAG Option | 9-7 |
| 9.5.8 | PROBER Option | 9-7 |
| 9.5.9 | PROBES Option | 9-7 |
| 9.6 | Miscellaneous Options | 9-9 |
| 9.6.1 | 72COL Option | 9-9 |
| 9.6.2 | FORINDEX Option | 9-9 |
| 9.6.3 | GLOBALS Option | 9-9 |
| 9.6.4 | ROUND Option | 9-9 |
| 9.6.5 | UNSAFEOPT Option | 9-9 |

## TABLE OF CONTENTS (Continued)

| Paragraph | Title | Page |
|---|---|---|
| | SECTION X. FORMATTING SOURCE CODE | |
| 10.1 | General | 10-1 |
| 10.2 | Coding | 10-1 |
| 10.3 | Source Formatter | 10-1 |
| 10.3.1 | Nester Functions | 10-1 |
| 10.3.2 | Nester Option Comment | 10-6 |
| 10.3.3 | Interactive Mode | 10-8 |
| 10.3.4 | Nester Commands | 10-8 |
| 10.3.5 | Executing Nester | 10-9 |
| 10.3.6 | Nester Error Messages | 10-10 |
| 10.3.7 | Nester Example | 10-10 |
| | SECTION XI. THE PASCAL COMPILER | |
| 11.1 | General | 11-1 |
| 11.2 | SILT1 | 11-1 |
| 11.3 | SILT2 | 11-1 |
| 11.4 | CODEGEN | 11-1 |
| 11.5 | Executing the Compiler | 11-1 |
| 11.6 | Error Handling | 11-2 |
| | SECTION XII. SEPARATE COMPILATION | |
| 12.1 | General | 12-1 |
| 12.2 | Requirements for Separate Compilation | 12-1 |
| 12.3 | The Configuration Processor | 12-3 |
| 12.3.1 | Functional Description of CONFIG | 12-3 |
| 12.3.2 | Format of Source Modules | 12-5 |
| 12.3.3 | Process Configuration | 12-5 |
| 12.3.4 | Compilation | 12-9 |
| 12.3.5 | Source Listing | 12-16 |
| 12.3.6 | Flags | 12-19 |
| 12.3.7 | Modifying a Process Configuration | 12-28 |
| 12.3.8 | Libraries | 12-31 |
| 12.3.9 | Text Editing | 12-36 |
| 12.3.10 | Required Files | 12-38 |
| 12.3.11 | Executing CONFIG | 12-38 |
| 12.4 | Split Program Utility | 12-42 |
| 12.4.1 | Split Program Command | 12-42 |
| 12.4.2 | Input Example | 12-43 |
| 12.4.3 | Library and Files | 12-43 |
| 12.4.4 | Execution | 12-43 |
| 12.5 | Split Object Utility | 12-44 |

## TABLE OF CONTENTS (Continued)

| Paragraph | Title | Page |
|---|---|---|
| | SECTION XIII. REVERSE ASSEMBLER | |
| 13.1 | General | 13-1 |
| 13.2 | Required Files | 13-1 |
| 13.3 | Executing the Reverse Assembler | 13-1 |
| 13.4 | Example Listing | 13-1 |
| | SECTION XIV. LINKING AND EXECUTING PASCAL TASKS | |
| 14.1 | General | 14-1 |
| 14.2 | TIP Runtime Options | 14-1 |
| 14.2.1 | Execution Under DX10 | 14-2 |
| 14.2.2 | LUNO I/O | 14-2 |
| 14.2.3 | Multitask Capability Under DX10 | 14-2 |
| 14.2.4 | Execution Under TX990 | 14-3 |
| 14.2.5 | Multitask Capability Under TX990 | 14-3 |
| 14.2.6 | Execution Under RX990 | 14-3 |
| 14.2.7 | Minimal Runtime Capability | 14-3 |
| 14.2.8 | Stand-Alone Execution | 14-4 |
| 14.2.9 | Shared Procedure Capability | 14-4 |
| 14.3 | Linking and Executing | 14-4 |
| 14.3.1 | Linking for DX10 Execution Using SCI Synonyms | 14-4 |
| 14.3.2 | Executing Under DX10 Using SCI Synonyms | 14-6 |
| 14.3.3 | Linking for DX10 Execution Using LUNOs | 14-7 |
| 14.3.4 | Executing Under DX10 Using LUNOs | 14-8 |
| 14.3.5 | Linking for TX990 Execution | 14-9 |
| 14.3.6 | Executing Under TX990 | 14-9 |
| 14.3.7 | Linking Multiple Tasks for TX990 Execution | 14-10 |
| 14.3.8 | Executing Tasks Linked with TX990 | 14-12 |
| 14.3.9 | Linking for RX990 Execution | 14-13 |
| 14.3.10 | Executing Under RX990 | 14-13 |
| 14.3.11 | Linking Minimal Runtime | 14-14 |
| 14.3.12 | Executing Minimal Runtime | 14-15 |
| 14.3.13 | Linking for Stand-Alone Execution | 14-15 |
| 14.3.14 | Stand-Alone Execution | 14-15 |
| 14.3.15 | Linking for Shared Procedures | 14-16 |
| 14.4 | Pascal Task Execution Considerations | 14-18 |
| 14.4.1 | Synonym Table Overflow | 14-19 |
| 14.4.2 | Condition Code | 14-19 |
| 14.5 | Compiling, Linking, and Executing with a Batch Stream | 14-19 |
| 14.6 | Debug Mode | 14-19 |
| 14.6.1 | Pascal Debug Commands | 14-21 |
| 14.6.2 | Debugging Under TX990 | 14-30 |
| 14.7 | Runtime Errors | 14-31 |
| 14.7.1 | Runtime Checks | 14-31 |
| 14.7.2 | I/O Errors | 14-31 |
| 14.7.3 | Memory Space Errors | 14-32 |

## TABLE OF CONTENTS (Continued)

| Paragraph | Title | Page |
|---|---|---|
| 14.7.4 | Error Termination | 14-32 |
| 14.7.5 | Use of the Abnormal Termination Dump | 14-37 |
| 14.7.6 | Unformatted Abnormal Termination Dump | 14-41 |
| 14.7.7 | TX990 Abnormal Termination Dump | 14-41 |

SECTION XV. COMPUTER-DEPENDENT SOURCE CODE PREPARATION TECHNIQUES

| Paragraph | Title | Page |
|---|---|---|
| 15.1 | General | 15-1 |
| 15.2 | Direct CRU I/O Routines | 15-1 |
| 15.2.1 | Procedure $LDCR | 15-2 |
| 15.2.2 | Procedure $SBO | 15-2 |
| 15.2.3 | Procedure $SBZ | 15-2 |
| 15.2.4 | Procedure $STCR | 15-2 |
| 15.2.5 | Function $TB | 15-3 |
| 15.3 | Operating System Interface Routines | 15-3 |
| 15.3.1 | I/O Routines | 15-3 |
| 15.3.2 | Identification Functions | 15-8 |
| 15.3.3 | Time and Date Procedures | 15-9 |
| 15.3.4 | Task Control Procedures | 15-10 |
| 15.3.5 | Message Handling Procedures | 15-12 |
| 15.3.6 | System Common Access Procedures | 15-13 |
| 15.3.7 | Semaphore Procedures | 15-14 |
| 15.3.8 | Procedure SVC$ | 15-16 |
| 15.4 | Dummy Main Routine | 15-16 |
| 15.5 | Program Considerations for LUNO I/O | 15-18 |
| 15.6 | Program Considerations for Multiple Tasks | 15-19 |
| 15.7 | Program Considerations for Stand-Alone Tasks | 15-20 |
| 15.8 | Program Considerations for Minimal Runtime Code | 15-21 |

APPENDIXES

| Appendix | Title | Page |
|---|---|---|
| A | TIP Standard Routines | A-1 |
| B | Example Programs | B-1 |
| C | System Dependent Information for DX10 Release 3 | C-1 |
| D | TIP Syntax | D-1 |
| E | Error Messages | E-1 |
| F | Assembly Language Routines | F-1 |
| G | ASCII Character Set | G-1 |
| H | Overlays in TIP Programs | H-1 |
| I | Type Conversions | I-1 |

## LIST OF ILLUSTRATIONS

| Figure | Title | Page |
|---|---|---|
| 2-1 | TIP Example Program | 2-4 |
| 3-1 | Syntax Diagram Symbols | 3-2 |
| 5-1 | Syntax Diagrams for Expressions | 5-16 |
| 8-1 | Syntax Diagrams of Block | 8-3 |
| 8-2 | Syntax Diagram for Routine Declaration | 8-5 |
| 8-3 | Syntax Diagram for Statement | 8-9 |
| 8-4 | Access to Variables in Program Structure | 8-19 |
| 8-5 | Access to Routines in Program Structure | 8-20 |
| 9-1 | A Source Listing with WIDELIST and MAP Options Enabled | 9-5 |
| 9-2 | Execution Time Displays for PROBER and PROBES Options | 9-7 |
| 9-3 | Compiler Source Listing – PROBER and PROBES Options Enabled | 9-8 |
| 10-1 | NESTER Statement Example | 10-6 |
| 10-2 | NESTER Example | 10-11 |
| 11-1 | Source Listing Example, Default Options | 11-3 |
| 11-2 | TIP Compiler SYSMSG File Display | 11-4 |
| 11-3 | Source Listing Containing Errors | 11-6 |
| 12-1 | Flow of Separate Compilation Using CONFIG | 12-4 |
| 12-2 | Contents of OUTPUT File, Initial CONFIG Run, Full Compilation | 12-11 |
| 12-3 | Source Listing, Full Compilation Example | 12-12 |
| 12-4 | Contents of OUTPUT File, Deferred Processing, Full Compilation | 12-13 |
| 12-5 | Contents of OUTPUT File, Initial Run, Partial Compilation | 12-14 |
| 12-6 | Source Listing, Partial Compilation Example | 12-15 |
| 12-7 | Contents of OUTPUT File, Deferred Processing, Partial Compilation | 12-17 |
| 12-8 | Contents of OUTPUT File for LIST Operation | 12-20 |
| 12-9 | Contents of OUTPUT File for LISTDOC Operation | 12-24 |
| 12-10 | Batch Stream for Separate Compilation | 12-41 |
| 12-11 | Example of Input to SPLITPGM | 12-45 |
| 13-1 | RASS Listing Example | 13-3 |
| 14-1 | Batch Stream for Compiling, Linking, and Executing a TIP Program | 14-20 |
| 14-2 | Abnormal Termination Dump Example | 14-33 |
| 14-3 | Linking Map of Example Program | 14-39 |
| 14-4 | Unformatted Abnormal Termination Dump Listing | 14-42 |
| 14-5 | TX990 Abnormal Termination Dump Listing | 14-44 |
| 15-1 | Semaphore Example | 15-15 |

## LIST OF TABLES

| Table | Title | Page |
|---|---|---|
| 9-1 | TIP Compiler Options | 9-2 |
| 10-1 | NESTER Options | 10-4 |
| 10-2 | NESTER Errors | 10-10 |
| 11-1 | Files Required by the Compiler Tasks | 11-5 |
| 12-1 | System Flags | 12-25 |
| 12-2 | Files Required for CONFIG | 12-38 |
| 14-1 | TIP Runtime Libraries | 14-2 |
| 14-2 | Nonsharable Runtime Library Routines | 14-18 |
| 15-1 | Device Type Codes | 15-8 |

# SECTION I

# INTRODUCTION

## 1.1 TEXAS INSTRUMENTS PASCAL

The programming language TI Pascal (TIP) has been designed to facilitate the construction of reliable, transportable systems and scientific programs. It is a relatively easy language to learn and to use. However, a basic assumption in the design of the language is that readability is as important as writeability. Many programs undergo modifications during their lifetime and the ease of modification is dependent on the ability of a programmer (frequently not the original author) to read and understand the program. Other considerations in the design of the language look forward to the development of more sophisticated techniques for verifying the correctness of programs. In some instances these considerations have resulted in restrictions or omission of features which programmers might otherwise find useful. For this reason it is possible that some other languages which have "special-purpose" features may be better suited for some applications (such as complex string-processing or pattern-matching).

TI Pascal does have excellent bit-manipulation capabilities, and is ideally suited for a wide variety of applications ranging from areas of system programming to the scientific and engineering programs traditionally written in FORTRAN. The TI Pascal compiler is itself written in TI Pascal. At present, other successful projects at TI using the lanugage include a text editor, a library management system, and numerous utility programs such as an object code compress utility, and a track mapper. In addition, favorable benchmarks have been obtained against FORTRAN-coded routines for a realtime navigation system. This manual describes the language as it is implemented on the Model 990 Computer. It is intended for new users of TIP.

## 1.2 EXTENSIONS AND MODIFICATIONS TO PASCAL

TI Pascal offers a number of enhancements to and modifications of the Pascal language as described in *Pascal User Manual and Report,* K. Jensen and N. Wirth, Springer-Verlag, 1975.

The following are the enhancements provided by TIP:

- Random access files (paragraph 6.5.5).
- Common variables having global extent and scope as specified by ACCESS declarations (paragraph 8.9).
- Dynamic bounds for arrays and sets (paragraph 8.5.2).
- Multiprecision integer variables (paragraph 5.2).
- Multiprecision real variables (paragraph 5.10).
- FIXED data type (paragraph 5.11).
- DECIMAL data type (paragraph 5.12).
- ESCAPE statement for exit from structured statements (paragraph 7.1).
- Explicit type override operator (paragraph 6.9).
- ASSERT statement (paragraph 5.4).

- External procedures and functions using FORTRAN linkage (paragraph 8.2.7.2).
- Standard Model 990 Computer dependent procedures and functions.
- Additional type-checking for procedure and function parameters.
- Underscore (_) and dollar sign ($) in identifiers (paragraph 3.2.1).
- Constant expressions (paragraph 5.9).
- FOR statement with IN generator (paragraph 5.8.2).
- Function LOCATION (paragraph 6.6).
- Dynamic array and set parameters (paragraph 8.5.2).
- Empty parameter lists for procedures and functions (paragraph 8.5.4).
- CLOSE procedure (paragraph 6.5).
- SIZE function (paragraph 6.7.4).
- HALT procedure (paragraph A.3).
- MESSAGE procedure (paragraph A.3).
- Hexadecimal constants (paragraph 3.2.2).
- ENCODE and DECODE procedures (paragraph 6.5.4).
- More reliable form of WITH statement (paragraph 6.3).
- OTHERWISE clause and subrange case labels with CASE statement (paragraph 5.8.1).
- Formatted READ operation (paragraph 6.5.3).
- DATE procedure (paragraph A.3).
- TIME procedure (paragraph A.3).

The modifications to Pascal are:

- Restriction of side effects of user-defined functions (paragraph 8.8).
- Restricted use of GOTO statement (paragraph 7.2).
- Local scope of FOR statement control variable (paragraph 5.8.2).
- Altered precedence for Boolean operators (paragraph 4.2).
- More flexible I/O functions and procedures (paragraph 6.5).
- Pre-defined symbol MAXINT is not supported.
- RESET procedure required for textfile INPUT (paragraph 6.5.2).

- REWRITE procedure permitted for textfile OUTPUT (paragraph 6.5.2).
- WRITE procedure replaces PUT (paragraph 6.5).
- READ procedure replaces GET (paragraph 6.5).
- Types of parameters of routines that are passed as parameters must be declared (paragraph 8.5.3).

The Pascal programmer may refer to the paragraphs that describe the extensions and modifications without studying the entire manual in detail. The user who is not familiar with Pascal may study the entire document to familiarize himself with the language.

## 1.3 COMPARISONS WITH FORTRAN

TI Pascal has a well-thought-out collection of control structures and allows statements to be composed from other statements in very general ways. As a result, TIP provides many improved capabilities for organizing the flow of control of a program. In contrast, although it has forms such as

```
IF(X.GT.MAX) MAX = X,
```

FORTRAN allows only very limited composition of statements. The component of the IF statement cannot be another IF statement, or a DO statement, for example. TIP allows the component of a structured statement such as an IF statement to be any statement, for example a compound statement (for performing a sequence of operations), a FOR statement (for looping), or another IF statement. In addition, other statements such as the WHILE, REPEAT, and CASE statements (which have no close counterparts in FORTRAN) can be used very effectively to produce well-structured, readable programs. The need for GOTO statements is greatly minimized in TIP.

Certain statements which are generally considered to be unreliable, such as the arithmetic IF, computed GOTO, and EQUIVALENCE statement, have been omitted from TIP.

Many implementations of FORTRAN do not have a strong type checking feature. The same variable may often be used in completely different contexts, for example, as a character and then as an integer, with no error messages or warnings given by the compiler.

On the other hand, a type is associated with each TIP variable, and when the source program is compiled, all assignments, expressions, and parameters are checked for type compatibility. In addition to real and integer, TIP includes character and Boolean types and also has more general structured types such as records and sets as well as arrays.

There is no conversion from real to integer in TIP except by explicit use of procedures TRUNC and ROUND. In particular, it is not legal to assign a real value to an integer variable. In some instances, there are implicit conversions in TIP, such as from integer or longint type to real, fixed, or decimal type.

Modular decomposition of a program is accomplished by routines in TIP. A TIP routine is either a procedure or a function which corresponds to a subroutine or function respectively in FORTRAN. In FORTRAN, variables are shared among subprograms by parameter passing or by means of COMMON declarations. TIP has call-by-value and call-by-reference for passing parameters, while most versions of FORTRAN have only call-by-reference. COMMON declarations in TIP differ from those in FORTRAN. In addition, routines may be nested, that is, a routine may itself contain declarations of other routines. The scope rules of TIP specify how variables and other objects declared at a higher level (such as a program) may be directly accessible at lower levels (such as routines declared within the main program).

Storage for most variables is allocated from the stack at runtime in TIP. At any moment, only the routines which are currently invoked have stack space allocated. When control returns from a routine, the stack space or stack frame used by that routine is made available for use by other routines. The result is that very efficient use is made of the available memory.

Since each invocation of a routine is allocated a separate area in the stack for its variables, TIP code is naturally reentrant. When a TIP routine is interrupted and reexecuted, a new stack frame is allocated and new copies of the local (noncommon) variables are created. If the original execution is resumed, all local variables are restored to the same values as when the routine was interrupted by restoring a pointer to the original stack frame.

Because a TIP routine is able to directly access data structures and routines declared at a higher level (as long as the routine is within the scope of these declarations), separate compilation of routines requires that the appropriate local declarations be provided for the separately compiled routine. The configuration processor simplifies the process of separate compilation.

Access to externally compiled FORTRAN routines is straightforward. A TIP program can call a FORTRAN routine with a simple external declaration.

Other differences between TIP and FORTRAN are summarized below:

- TIP functions may not have side effects, so that when a function is invoked, it may not change the values of any variables except those local to the function.

- FORTRAN imposes restrictions on the allowable forms which may be used for array subscripts. In TIP a subscript may be a general expression.

- There is nothing corresponding to a Statement Function in TIP.

- There are no FORMAT statements (although formatted I/O is supported) and there is nothing corresponding to the FORTRAN IMPLIED DO in TIP.

- The control variable in a TIP FOR statement may be incremented (or decremented in the case of DOWNTO) only by one. The loop will not be executed at all if the initial value is greater than the final value (or less than the final value in the case of DOWNTO). The control variable may not be altered within the FOR statement.

- There is no exponentiation operator (**) in TIP.

- Some versions of FORTRAN have a multiple assignment capability (X = Y = Z), but this does not exist in TIP.

- In FORTRAN, explicit declarations of variables are optional. In TIP they are mandatory, that is, each variable must be explicitly declared before it is used (except FOR control variables, synonyms in WITH statements, and escape labels).

- FORTRAN compilers ignore most blanks in the source program. Blanks are significant in TIP, however, and are required in some contexts. For example, keywords and identifiers must be separated, so that

  ```
  X = YANDZ > 5
  ```

  is incorrect if the intended expression is

  ```
  X = Y AND Z > 5
  ```

- Statements may be placed anywhere in columns 1-72 of the source records (or optionally records of any length may be used). There is no continuation column, comment column, or label field as in FORTRAN. Statements may be extended across record boundaries, but the end of each record is a separator, so that strings (written within quotes), identifiers, or keywords may not extend from one record to the next.
- In FORTRAN, it is permissible to end several DO loops on the same statement. In Pascal a compound statement (surrounded by a BEGIN - END pair) forms the unit to be repeated in a FOR statement. In this case, as well as in all other uses of BEGIN - END, each BEGIN must have its own END. There is no CONTINUE statement in TIP.
- The COMPLEX type is not standard in TIP, although it can easily be simulated by means of a user defined type.
- Column one of output files is not used for carriage control as it is in FORTRAN. WRITELN and PAGE procedures are used to control spacing.
- In FORTRAN, arrays are stored in column-major order while in TIP they are stored in row-major order.
- In FORTRAN, a read executed after the last record in the file has been read causes end-of-file to become true. In TI Pascal, end-of-file becomes true when the last element in the file is read.
- FORTRAN logical variables correspond to Pascal Boolean variables.

## 1.4 COMPARISONS WITH PL/I

In comparison with PL/I, Pascal offers the following advantages:

- The full language can be processed by implementations on small computer systems.
- Pascal more easily produces well-optimized code.
- Pascal provides better control structures.
- Pascal has fewer default conditions and fewer implicit conversions.
- Data types must be explicit in Pascal and conversion and transfer of types must also be explicit.
- Pascal is simple and avoids unnecessary alternative functions.
- There are efficient implementations of Pascal.

## 1.5 ORGANIZATION OF THE MANUAL

Following an overview of the language, the manual describes the language in detail. Programmers who are familiar with Pascal may only concern themselves primarily with the enhancements and modifications to the language. Others will need to study the language portion of the manual thoroughly.

Section IX describes the compiler options, and should be studied by all programmers. The remaining sections describe the Model 990 Computer implementation of Pascal and the utilities available for program development. These utilities are:

| | |
|---|---|
| Source code formatter | NESTER |
| Configuration processor (for separate compilation of program modules) | CONFIG |
| Source code splitter | SPLITPGM |
| Object file splitter | SPLITOBJ |
| Reverse assembler | RASS |

# SECTION II

# OVERVIEW

## 2.1 PROGRAM STRUCTURE

Pascal is a programming language that promotes structured programming using a top down approach. One of the most distinctive features of the language and the resulting program is the structure. The overall structure is the program.

A program in TI Pascal has a heading which gives the program a name and lists its parameters, if any, followed by the declarations and statements of the program, which are called a block. If the heading includes program parameters, the values for these parameters are passed to the program when it is executed. The block consists essentially of two parts: declarations, which serve to define the structure of the data upon which the program operates, and statements, which specify the structure of the operations which the program performs upon its data.

The secondary structure within the program is the routine, which may be either a procedure or a function. Like the program, each routine consists of a heading and a block. The block contains the declarations of the routine, followed by the statements of the routine. The headings of all routines called in the statements of the main program must appear at the end of the declarations of the program.

The declarations of each routine may include the headings of other routines. All routines called in a statement of a routine must be declared.

Normally a routine is declared within the routine that calls it. When a routine is called by more than one routine, or by the main program and one or more routines, it must be declared at a point that makes it available to all routines that call it.

The structure of a Pascal program implies a scope for data and routines declared in any block of the structure. The scope of a routine or of a data declaration consists of the blocks within which the routine may be called or the data may be accessed. In general, the scope of a routine or of data includes the block in which the routine or data is declared, all routines declared in the same block, and all routines declared in these routines. The scope of data is related to the extent of the data. The extent is the time during program execution during which the data may be accessed, and is, in general, the duration of execution of the scope of the data. The scope and extent are described fully in paragraph 8.7.

Routines in TI Pascal may be overlays, and the program may be structured as a root phase and a series of overlays. Refer to Appendix H for further information.

## 2.2 DATA TYPES

Another distinctive feature of Pascal is the structuring of data. All data items must be declared to be of a specific type. Pascal supports standard types and a means of declaring user defined types.

The basic concept underlying the data structures is that of a data type. Every variable occurring in a TIP program is associated with one and only one type. This type specifies how the value of the variable is represented at the machine level as a sequence of binary digits. For example, there is a representation for the data type REAL, the type INTEGER, the type CHAR (character), etc.

One advantage of the type concept in TIP is that it allows complex programs to be written more easily and with greater reliability since the details of data representation are handled by the compiler. The programmer rarely needs to know these details, unless packed data structures are being used. The effects of the packing algorithm are described in terms of the size of the individual data types in paragraph 6.7. Another significant reason for having types is that they divide the data objects into categories so that the compiler is able to check that consistent use of the data is being made in the program. For example, it is considered inconsistent to perform arithmetic operations on character data, so an attempt to add 1 to the letter 'A' is illegal and will result in a compile-time error.

**2.2.1 SIMPLE TYPES.** Simple types form the fundamental elements of data for a Pascal program. They are the basic units from which the more complex data structures (discussed briefly in the next paragraph) are composed. A simple type is either one of the standard types INTEGER, LONGINT, BOOLEAN, CHAR, REAL, FIXED, or DECIMAL, or is defined by the programmer (scalar or subrange).

**2.2.1.1 Enumeration Types.** An enumeration type is characterized by the set of its distinct values, upon which a linear ordering is defined. For example, the type INTEGER consists of the set of integer values . . . , -2, -1, 0, 1, 2, . . . On the Model 990 computer, integer values are stored in two bytes, and the range of values is -32,768 through +32,767. The type CHAR consists of the character set available on the machine and includes 'A', 'B', . . ., 'Z', '0', . . ., '9', as well as various other symbols and nonprinting control characters. The enumeration types consist of INTEGER, LONGINT, BOOLEAN, and CHAR, which are standard types, and scalar and subrange types, which are user defined. All enumeration types have a first value and a last value. They are ordered so that any value other than the first has a predecessor and any value other than the last has a successor. Also, comparing two enumeration values by means of the relational operators to determine their relative ordering is valid.

Enumeration types are used for counting purposes, for example, to index into an array or to control the number of iterations of a FOR statement.

The basic operators for variables of enumeration types are assignment (:= -- see paragraph 4.2 for the assignment statement) and the relational operators described below:

| | |
|---|---|
| < | less than |
| = | equal |
| > | greater than |
| <= | less than or equal |
| <> | not equal |
| >= | greater than or equal |

The standard functions applying to enumeration types are:

| | |
|---|---|
| SUCC (X) | the successor of X |
| PRED (X) | the predecessor of X |
| ORD (X) | the ordinal value of X (applies to all enumeration types except INTEGER and LONGINT) |

An attempt to take the successor of the largest value or the predecessor of the smallest value of an enumeration type results in a runtime error, if the subrange check (a compiler option) is turned on. Otherwise, the value which is returned is an undefined element of the enumeration.

**2.2.2 STRUCTURED TYPES.** A structured type is composed from other types which are called components. The declaration of a structured type specifies the type(s) of its components and the structuring method by which they are composed. The structuring methods which are available in TIP are the ARRAY, the FILE, the RECORD and the SET. In addition, there is the structured type POINTER, which is used to reference elements of a given type indirectly.

**2.2.3 STORAGE REQUIREMENTS.** When using a high-level language such as TIP, the programmer is usually not concerned with the amount of storage (in terms of bits or words) which is used by the simple types. In general, each simple type uses the least amount of storage that is conveniently accessible on the machine. This means that a type such as Boolean uses an entire word in the Model 990 Computer. However, for those applications for which it is necessary to manipulate bits or conserve storage, TIP allows some data structures to be packed so that several components of the structure may occupy one word. Referencing a component of the packed stucture then allows that portion of the word to be accessed directly. Packing may be used to economize storage requirements, but this may result in a loss of efficiency of access. The structured types which may be packed are ARRAY, RECORD, and SET.

## 2.3 DECLARATIONS

The declarations of the program consist of the parts listed below. Each part is optional, but any declarations which are used must appear in the order listed. The exact syntax for each declaration is given in paragraph 8.2, but a brief description of its purpose is given after each declaration here. Examples appear throughout the manual.

| | |
|---|---|
| Label declaration | (for integer labels referenced by GOTOs) |
| Constant declaration | (synonyms for number, character, Boolean, and string constants) |
| Type declaration | (associates an identifier with a type specification) |
| Variable declaration | (declares variables and specifies their type) |
| Common declaration | (declares variables which do not follow the normal scope and extent rules) |
| Access declaration | (declares access to common and optionally, nonlocal variables) |
| Routine declaration | (defines procedures or functions) |

## 2.4 STATEMENTS

Statements describe the actions which a computer program performs on its data. Statements may contain other statements as components, in which case they are structured; otherwise they are simple. Simple statements include the assignment statement, PROCEDURE statement, ESCAPE statement, ASSERT statement, and GOTO statement. Most of the structured statements are used to control the sequence of execution of other statements, that is, they are used to form loops and conditional branches. The different forms which the loops, branches and transfers can take are collectively called the control structures of the language. The control structures of TI Pascal produce programs which are reliable and very readable. The structured statements are the compound statement, the conditional statements IF and CASE, the repetitive statements WHILE, REPEAT, and FOR, and the WITH statement, which is used with records.

## 2.5 EXAMPLE PROGRAM

Figure 2-1 shows an example TIP program. Notice that the lines of the program contain both upper and lower case letters. Users who write the source code for their program with a 911 VDT (or other device having both upper and lower case letters) may use upper and lower case letters interchangeably. This sometimes promotes the readability of the program.

The program consists of the program heading, the declarations, and the statements. The declarations of the example program consist of a constant declaration and two variable declarations. The program heading must be first, followed by those declarations that are applicable, in the proper order. Note that no label, type, common, access, or routine declarations are included in this example. Each declaration is separated from the following declaration by a semicolon; the last declaration is separated from the first statement by a semicolon.

The statements of the program are structured into a single compound statement. Within the compound statement (from the first appearance of the reserved word BEGIN to the last appearance of the reserved word END) the component statements are separated by semicolons. One of the component statements is a structured statement, specifically a FOR statement, one of the repetitive statements. Within the FOR statement is another compound statement, the component statements of which are separated by semicolons. The reserved word END that terminates the top level compound statement is followed by a period (.), which terminates the program.

```
Program INFLATION;
(* Find the factor by which the dollar is devalued after
   N years for N = 1, 2, . . ., 10.  Use annual inflation
   rates of 7, 8, and 10 percent *)
Const N = 10;
Var YEAR     :  Integer;
   R1, R2, R3:  Real;
Begin
   Writeln('    YEAR    7 PERCENT    8 PERCENT     10 PERCENT');
   R1  := 1.0; R2  := 1.0; R3  := 1.0;
   For YEAR  := 1 To N Do
   Begin
      R1  := R1*1.07;
      R2  := R2*1.08;
      R3  := R3*1.10;
      Writeln(YEAR,  R1,  R2,  R3)
   End
End.
```

**Figure 2-1. TIP Example Program**

# SECTION III

# NOTATION AND VOCABULARY

## 3.1 SYNTAX NOTATION

The syntax of a programming language describes the form which a legal program in that language may take. In a language such as TI Pascal, the syntax may be expressed very concisely by either syntax diagrams or by the more traditional Backus-Naur Form or BNF (sometimes called Backus-Normal form).

In BNF, each element of the language is defined by means of an equation-like rule called a production. The entity being defined is written to the left of the symbol ::= and the definition is written to the right of that symbol. The definition may be expressed in terms of language elements which are defined by additional productions. The following symbols are used in writing definitions:

| | |
|---|---|
| ::= | for writing productions, means "is defined to be", |
| < > | for enclosing nonterminal symbols, i.e., entities which are defined by a production, |
| [ ] | for enclosing entities which are optional, |
| { } | for enclosing entities which may be repeated zero or more times, |
| \| | for representing alternation, e.g., A \| B \| C means A or B or C. |

**NOTE**

Both brackets ([ ]) and braces ({ }) are used in TIP as terminal symbols. When used in BNF productions to specify terminal symbols, the brackets and braces are enclosed in quotation marks.

For example, an identifier may be defined as:

```
<identifier> ::= <letter> {<id character>}
<id character> ::= <letter> | <digit> | _
<letter> ::= A | B | C | D | . . . | Z | $
<digit> ::= 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 0 |
```

In this manual the language syntax is specified by BNF productions. The BNF productions are supplemented by syntax diagrams to illustrate the syntax of TIP declarations and statements.

A syntax diagram is a directed graph with a single input edge and a single output edge. The graph represents a syntax rule. Any possible path from the input edge to the output edge corresponds to an application of the syntax rule. A syntax diagram, in contrast with a BNF production, does not reflect the precedence of operators. The symbols that appear in syntax diagrams are shown in figure 3-1. An example of a syntax diagram representing the syntax of an identifier also is shown in figure 3-1.

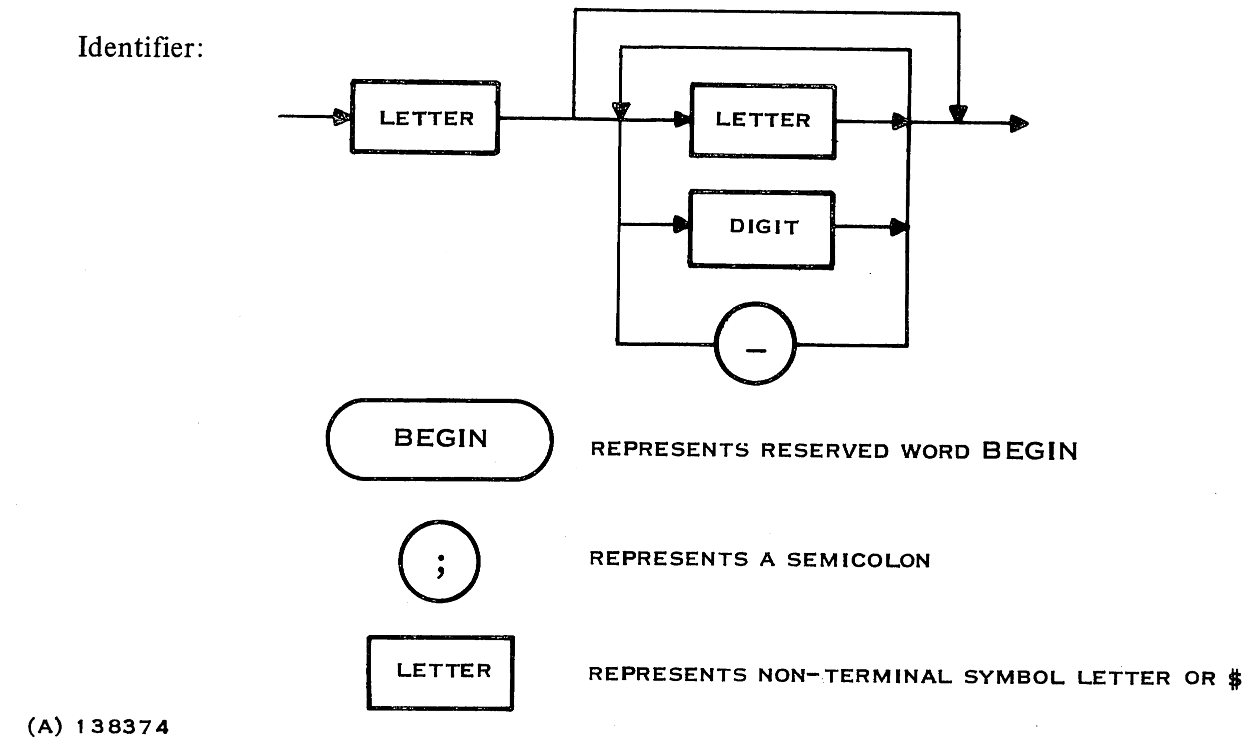


**Figure 3-1. Syntax Diagram Symbols**

## 3.2 BASIC SYNTAX ELEMENTS

The TIP character set consists of the letters A-Z, the digits 0-9, and the special characters

+ - * / " . , ; : = ' < > ( ) [ ] { } # ↑ @ ? _ $

Lowercase letters may be used with the 911 VDT and other devices that have both upper and lowercase letters; however, the TIP compiler translates these letters to uppercase. This means that the reserved word BEGIN can also be entered as Begin or as begin. It also means that identifiers MYPROG, Myprog, and myprog are not unique; the compiler processes any one of them as if it were MYPROG.

Using these characters, certain special symbols are formed which have a fixed meaning in the language. Some of these special symbols are used for operators and delimiters:

```
+  -  *  /  :=  =  <>  <  <=  >=  >  ::
(  )  (. .)  (* *)  . .  . ,  ;  :  ' "  #  @  ?
```

**NOTE**

To delimit array indices and sets, (. .) may be substituted for [ ]; to delimit comments, (* *) may be substituted for { }; and to identify pointers, @ may be substituted for ↑.

Other special symbols are reserved words.

| | | | |
|---|---|---|---|
| ACCESS | ELSE | LABEL | REAL |
| AND | END | LONGINT | RECORD |
| ARRAY | ESCAPE | MOD | REPEAT |
| ASSERT | FALSE | NIL | SET |
| BEGIN | FILE | NOT | TEXT |
| BOOLEAN | FIXED | OF | THEN |
| CASE | FOR | OR | TO |
| CHAR | FUNCTION | OTHERWISE | TRUE |
| COMMON | GOTO | OUTPUT | TYPE |
| CONST | IF | PACKED | UNTIL |
| DECIMAL | IN | PROCEDURE | VAR |
| DIV | INPUT | PROGRAM | WHILE |
| DO | INTEGER | RANDOM | WITH |
| DOWNTO | | | |

Reserved words have a fixed meaning and may not be used as identifiers since they may not be redefined in a TIP program.

**3.2.1 IDENTIFIERS.** Identifiers are used as names denoting constants, variables, types, procedures, functions, programs, and escape labels. In TIP identifiers consist of a letter or $, followed by any combination of letters, digits, $, or _. A maximum length is imposed by the restriction that identifiers may not cross record boundaries and hence may not be more than 72 characters long. All characters in an identifier are significant, except in the case of program, routine, and common names which are limited to 6 characters by the link editor on the TI 990. These names should be unique in the first 6 characters. The syntax for identifiers is:

<identifier> ::= <letter>{<letter>|<digit>| _}

Examples:

| Legal identifiers | Illegal identifiers | |
|---|---|---|
| X | VAR | (RESERVED WORD) |
| $ARRAY | _MAXVAL | (CAN'T START WITH _) |
| VERY_LONG_IDENTIFIER | ARRAY-BOUND | (CAN'T CONTAIN -) |
| VALUE_3 | 3RDVAL | (CAN'T START WITH 3) |
| SQRT | FIRST NUM | (CAN'T CONTAIN BLANK) |

**NOTE**

Some identifiers are standard in TIP, that is, they have a predefined meaning, but they may still be redefined by the programmer in which case the standard meaning is no longer available. For example, if SQRT is declared as a variable, then the standard SQRT function is no longer available. A list of standard routines is given in an appendix. Also, since routines from the runtime library for TIP always include a $ in their name, conflicts with these names may be avoided by not using $ in identifiers.

**3.2.2 NUMBERS.** Unsigned integer numbers are written as a string of decimal digits. The string may be preceded by a # sign to indicate hexadecimal notation and/or followed by L to indicate LONGINT. If the # sign is used to indicate hexadecimal then the number contains hexadecimal digits. The default precision for integer numbers is the precision that can be obtained with 16 bits (two bytes). The range of values is -32,768 through +32,767. Type LONGINT is available for extended precision. Values are stored in four bytes, providing a range of values from -2,147,483,648 through +2,147,483,647.

The following table shows the values represented by the 16 hexadecimal digits:

| **Hexadecimal digit** | **Decimal equivalent** |
|---|---|
| 0-9 | same |
| A | 10 |
| B | 11 |
| C | 12 |
| D | 13 |
| E | 14 |
| F | 15 |

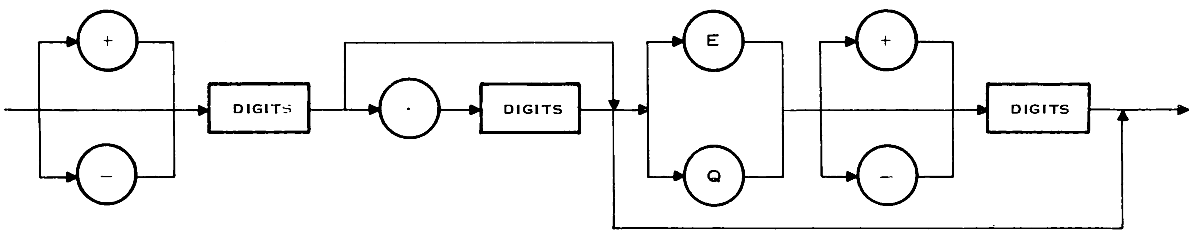
Hexadecimal integer constants are processed by the compiler as positive integers. They may consist of up to eight hexadecimal digits; values greater than $7FFF_{16}$ are converted to type LONGINT. In this conversion, the positive sign is extended; e.g., $FACE_{16}$ becomes $0000FACE_{16}$. If this value is placed in an INTEGER type variable by means of an assignment statement, the LONGINT constant is converted to INTEGER type by truncation; e.g., $0000FACE_{16}$ becomes $FACE_{16}$. If the integer variable is again converted to LONGINT (implicitly or explicitly), the most significant bit is considered to be the sign bit and is extended. For example, $FACE_{16}$ becomes $FFFFFACE_{16}$. The positive hexadecimal integer constant is no longer positive.

Real constants are written according to the following syntax:

```
<real constant> ::= [<sign>] <digits> . <digits>
     | [<sign>] <digits> [. <digits>] E <scale factor>
     | [<sign>] <digits> [. <digits>] Q <scale factor>

<scale factor> ::= <digits> | <sign> <digits>

<sign> ::= + | —

<digits> ::= <digit>{<digit>}
```

The syntax diagram is as follows:

Real constant:



Note that a decimal point must be surrounded on both sides by digits. A decimal number written as

nnn.nnnEmm or nnn.nnnQmm

represents the number nnn.nnn times 10 to the power mm. E represents the default precision while Q specifies that the constant is to be of the maximum precision available within 32 bits.

The precision of a real constant is determined by the number of decimal digits specified. When seven or fewer digits are entered, the constant has the default precision. When eight or more digits are entered, the constant has the maximum precision. When the constant is supplied in the E format and eight or more digits are entered, the constant has the maximum precision.

Examples:

| **Integer numbers** | **Real numbers** |
|---|---|
| 126 | 12.69 |
| #25A | 713E6 |
| #AFL | 5.2E-4 |
| 0006 | 999.2E+3 |
| 237605L | 3.14159268Q0 |

**Illegal numbers**

| | |
|---|---|
| .00159 | — Decimal point not surrounded by digits. |
| 75.E2 | — Decimal point not surrounded by digits. |
| 2.0E10.0 | — Real exponent not allowed. |
| #56A6.3 | — Hexadecimal notation not allowed with decimal point. |

In addition to real and integer types, FIXED and DECIMAL representations are also available. They are described in paragraphs 5.11 and 5.12.

**3.2.3 STRINGS.** A sequence of characters enclosed by apostrophes is called a string constant. String constants are represented internally as packed arrays of characters, as described in paragraph 6.7.1. A string constant cannot be longer than 70 characters. Any character may be represented in a string by a # followed by its 2-digit hexadecimal character code. This enables unprintable control characters to be included in strings. Within a string, ' is represented by '' and # is represented by ##.

Examples:

```
'THIS IS A STRING'
'CARRIAGE RETURN #0D'
'ISN''T THIS RIGHT?'
'STEP ##10'
```

## 3.3 SEPARATORS

At least one separator must occur between any two constants, identifiers, reserved words, or special symbols. No separator may occur within these elements, except that spaces may occur within strings. Separators are spaces, ends of lines, comments, or remarks. For example, in

```
WHILE X <10
```

a space separates WHILE and X. It is not equivalent to write:

```
WHILEX<10
```

A comment is any sequence of characters beginning with { or (* and ending with *) or }, except that { or (* does not begin a comment within a string. A remark is any sequence of characters beginning with a " and extending to the end of the logical source record, except that " within a string does not begin a remark. Comments and remarks serve to document a program and may be replaced by blanks without affecting execution of the program (with the exception of compiler options, which also take the form of comments. See Section IX for compiler options).

# SECTION IV

# THE ASSIGNMENT STATEMENT

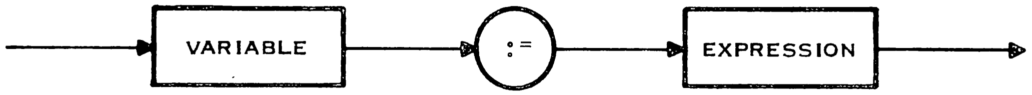
## 4.1 ASSIGNMENT STATEMENT

An assignment statement specifies that the value of an expression is to be calculated and the result assigned to a variable. The form of an assignment statement is:

<variable> := <expression>

The syntax diagram is as follows:

Assignment statement:



The type of <expression> must be compatible with that of <variable>, which means that their types must be the same, with several exceptions. One exception is that it is permissible for <variable> tc be of type REAL and <expression> to be of type INTEGER or LONGINT. Any conversions which are desired may be specified explicitly by a call to the appropriate function. Complete rules on type compatibility are given in paragraph 6.8. There is also the restriction that <variable> must not be of type FILE.

Another use of the assignment statement is within the body of a function where the value computed is assigned to the identifier which denotes the function. In this case the type of <expression> must be compatible with the result type of the function. Functions are described in Section VIII.

There is a compiler option to check that values assigned to subrange variables (see paragraph 5.7.3) are within the proper bounds.

Examples:

```
X := A + 2
MAXVALUE := SQRT(I+J) - A*SIN(X)
P1@.NAME := 'PASCAL' (* POINTER, RECORD, AND STRING*)
HUE := [ RED, SUCC(YELLOW) ] (* HUE IS OF TYPE SET *)
FLAG := A > MAX (* FLAG IS OF TYPE BOOLEAN *)
```

The expression on the righthand-side of the assignment statement specifies a computation which yields a value for the expression. Expressions consist of variables, constants, function designators, and operators such as +, *, <=, etc.

## 4.2 OPERATORS AND EXPRESSIONS

The two operands operated upon by an operator must have data types that are compatible (paragraph 6.8) and appropriate to the operator. In order to evaluate an expression, it is necessary to know the meaning of each operator and its precedence, which specifies the order in which the operators will be applied. The operators are:

Group 1: Multiplying operators:

| | |
|---|---|
| * | Multiplication; set intersection |
| / | real division |
| DIV | integer division (divide and truncate) |
| MOD | modulus, A MOD X = A — ((A DIV X) * X) |

Group 2: Adding operators:

| | |
|---|---|
| + | addition; unary plus; set union |
| - | subtraction; unary minus; set difference |

Group 3: Relational operators:

| | |
|---|---|
| = | equal |
| <> | not equal |
| < | less than, proper set inclusion |
| > | greater than, proper set inclusion |
| <= | less than or equal, set inclusion |
| >= | greater than or equal, set inclusion |
| IN | set membership |

Logical operators:

| | |
|---|---|
| Group 4: NOT | Negation |
| Group 5: AND | Conjunction |
| Group 6: OR | Disjunction |

When used on strings, the relational operators denote alphabetical ordering according to the collating sequence of the ASCII set of characters. The ASCII character set is listed in Appendix G.

The list of operators is in order of precedence, with groups of higher precedence listed first. In an expression, operators of highest precedence are evaluated first, and within each group, the operators have equal precedence and are evaluated from left to right within the expression. Parentheses may be used to explicitly specify the order of evaluation.

Examples:

| **Expression** | **Value** |
|---|---|
| 2 + 3 * 5 | 17 |
| 15 DIV 4 * 4 | 12 |
| NOT (5 + 5 >= 20) | TRUE |
| 6 + 6 DIV 3 | 8 |
| 3 < 5 OR 2 >= 6 AND 1 > 2 | TRUE |

The syntax for expressions is given in detail in paragraph 5.9.

# SECTION V

# ENUMERATION TYPES AND RELATED STATEMENTS

## 5.1 ENUMERATION TYPES

Enumeration types were mentioned briefly in the overview of Section II. Enumeration types are characterized by an ordered sequence of values. The functions which apply to enumeration types are SUCC (successor), PRED (predecessor), and ORD (ordinal value). ORD applies to all enumeration types except INTEGER and LONGINT. Also, the relational operators <, >, =, <=, >=, and <> apply to operands of enumeration type. Runtime checks are available to detect the error of taking the successor of the largest value of an enumeration or the predecessor of the smallest value of an enumeration.

## 5.2 INTEGER AND LONGINT TYPES

A value of type INTEGER is an element of a finite set of whole numbers. The range of integer values is determined by the word size of the machine. On the Model 990 Computer the range is -32,768 through +32,767. The type LONGINT provides for extended precision integers; the range of values is -2,147,483,648 through +2,147,483,647.

A nonsuffixed integer constant is of type INTEGER if its value lies within the *subrange defined* by the predefined type INTEGER, or LONGINT if its value lies outside the subrange defined by INTEGER but within the subrange defined by LONGINT. If an integer constant is suffixed with an L, it is of type LONGINT.

The following operators are defined for INTEGER or LONGINT operands and yield an INTEGER or LONGINT result:

| | |
|---|---|
| + | unary plus or add |
| - | negate or subtract |
| * | multiply |
| DIV | integer divide (divide and truncate) |
| MOD | modulus (A MOD X = A - ((A DIV X) * X)) |

The standard functions applying to arguments of type INTEGER and LONGINT are:

| Function | Value | Result Type |
|---|---|---|
| ABS(X) | Absolute value of X. | INTEGER |
| SQR(X) | X squared. | INTEGER |
| CHR(X) | The character with the ordinal value of X. | CHAR |
| ODD(X) | TRUE if X is odd, FALSE Otherwise. | BOOLEAN |
| LOCATION | See paragraph 6.6 on pointers. | INTEGER |

The standard functions which permit conversion of arguments of type INTEGER or LONGINT are:

| Function | Value | Result Type |
|---|---|---|
| FLOAT(X,P) | Real value with precision P. | REAL |
| FIX(P,Q,X) | Fixed point value with precision (P,Q) | FIXED-POINT |
| DEC(P,Q,X) | Decimal value with precision (P,Q) | DECIMAL |

The standard functions which permit conversion between INTEGER and LONGINT are:

| Function | Value | Result Type |
|---|---|---|
| LINT(X) | X of type INTEGER converted to LONGINT | LONGINT |
| TRUNC(X) | X of type LONGINT converted to INTEGER | INTEGER |

The arithmetic relational operators apply to INTEGER or LONGINT operands and yield a Boolean result.

## 5.3 BOOLEAN TYPE

A value of type BOOLEAN is one of the logical truth values denoted by the reserved words TRUE and FALSE.

Operators defined for Boolean operands which yield Boolean values are:

| | |
|---|---|
| NOT | logical negation |
| AND | logical conjunction |
| OR | logical disjunction |

Since the type BOOLEAN is defined so that FALSE < TRUE, each of the possible Boolean operations can be defined using the Boolean operators listed above and the relational operators. For example, if P and Q are Boolean values:

```
P <= Q expresses implication (P implies Q)
P = Q  expresses equivalence
P <> Q expresses exclusive OR
```

Because of the precedence rules, expressions involving Boolean and relational operators may have to be parenthesized in order to obtain the desired result. For example, to express the relationship between A and the logical conjunction of B and C:

```
A = (B AND C)
```

## 5.4 THE ASSERT STATEMENT

ASSERT statements allow the user to specify by a Boolean expression a condition which should exist at a given point in a program. The form is:

```
ASSERT <expression>
```

The <expression> must be of type Boolean and must not contain the call of an external function. Each time flow of control passes through the ASSERT statement, the expression is evaluated, if ASSERTs are enabled (by a compiler option). If the expression is true, processing continues normally. If the expression is false, a runtime error occurs.

Examples:

```
ASSERT X = 0
ASSERT MAX >= L
ASSERT FNC(X)
```

Assert statements are an aid to program testing. They may be included at any point in a program where it is desired to test at runtime a condition or relation which should be true. When the ASSERT statement is disabled by a compiler option, the ASSERT may still serve as a useful comment.

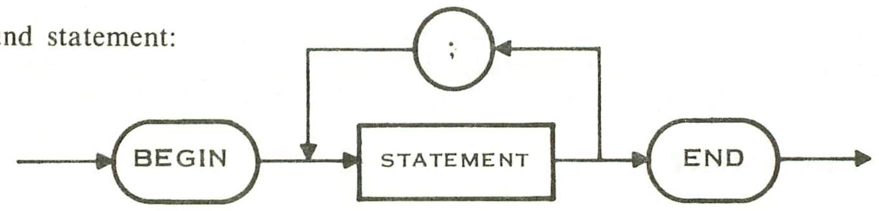
## 5.5 THE COMPOUND STATEMENT

A compound statement is written by enclosing a sequence of statements between the keywords BEGIN and END. The compound statement specifies that the component statements are to be executed one by one in the order in which they are written, but that the entire sequence is to be treated as a single statement. The syntax is:

BEGIN <statement> {; <statement>} END

The syntax diagram is as follows:

Compound statement:



The semicolon (;) is used as a statement separator, that is, it appears between statements, and need not appear after the end of the last statement.

Example:

```
BEGIN
     SWAP := X;
     X := Y;
     Y := SWAP
END
```

Note that no semicolon is needed after the statement Y := SWAP. The presence of a semicolon here would imply the existence of a statement between the semicolon and the END. This is an example of the empty statement which occurs wherever the syntax of TIP requires a statement but no statement appears. In this case, the empty statement does no harm since its presence has no effect. Other examples of the empty statement (paragraph 5.6.1) may alter the intended meaning of the program.

As an illustration of the empty statement, the compound statement

```
BEGIN
    <statement1>;
    <statement2>;
    ...
    <statementn>;
END
```

is interpreted as

```
BEGIN
    <statement1>;
    <statement2>;
    ...
    <statementn>;
    <empty>
END
```

## 5.6 CONDITIONAL AND REPETITIVE STATEMENTS

A conditional statement contains an expression and one or more component statements. At run time, the value of the expression determines which, if any, of the component statements is executed. The two types of conditional statement are the IF statement and the CASE statement. (See paragraph 5.8.1 for the CASE statement.)

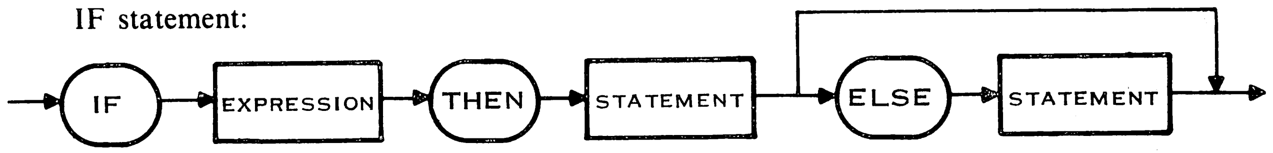
A repetitive statement specifies the repeated execution of its component statements. If the number of repetitions is known before the loop is entered, a FOR statement may be used, and if the number of repetitions is determined after the loop is entered, a WHILE or REPEAT statement should be used.

**5.6.1 THE IF STATEMENT.** The syntax of the IF statement is:

```
IF <expression> THEN <statement1>
       [ ELSE <statement2> ]
```

The syntax diagram is as follows:

IF statement:



The <expression> must be of type Boolean. If the value of <expression> is true, then <statement1> is executed. If <expression> is false and the optional ELSE clause is included, then <statement2> is executed.

Examples:

```
IF A >= 0 AND A <= 1 THEN X := SIN(A)
IF X < Y THEN MAX := Y ELSE MAX := X
```

Since there is no keyword that terminates the IF statement, the preceding syntax is ambiguous relative to ELSE clauses in nested IF statements. For example, in the statement

```
IF <condition1> THEN
     IF <condition2> THEN <statement1>
     ELSE <statement2>
```

the "ELSE <statement2>" can be interpreted as belonging to either IF statement. This ambiguity is resolved by always associating an ELSE clause with the most recent unmatched THEN preceding it. The preceding example then is interpreted as:

```
IF <condition1> THEN
     BEGIN
          IF <condition2> THEN <statement1>
               ELSE <statement2>
     END
```

Note that there is never a semicolon preceding the ELSE. The statement

```
IF X > 0 THEN N := N + 1;
     ELSE N := 0
```

contains a syntax error, since the semicolon would separate the initial part of the IF statement from a statement beginning with the reserved word ELSE, which is not possible in TIP.

Misplaced semicolons in some positions may not result in a syntax error, as the following example shows:

```
IF CONDITION THEN;
     N := + 1
```

The IF statement is terminated by the semicolon, so that an empty statement is implied between THEN and N := N + 1. The assignment statement N := N + 1 is executed regardless of the value of CONDITION.

It is important to realize that the component statements of IF statements may be any statements. Frequently an entire sequence of statements is to be executed if the condition is not TRUE, in which case the compound statement is used to group the sequence into a single statement, as the following example shows:

```
IF X >  0 THEN C2 := SQRT(X)
     ELSE BEGIN
          C1 := 0;
          C2 := SQRT(-X)
     END
```

A compound statement may also be used following the keyword THEN.

The following examples illustrates a common misuse of the IF statement:

```
IF X>0 THEN POS := TRUE ELSE POS := FALSE
```

This statement is needlessly complex. The same result is more clearly accomplished by the simple assignment:

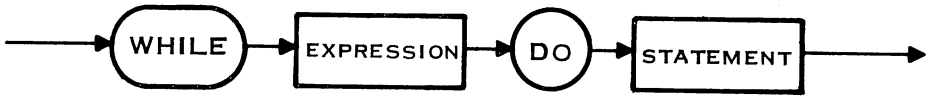
```
POS := X>0
```

**5.6.2 THE WHILE STATEMENT.** The WHILE statement consists of a Boolean expression and a component statement. The component statement is executed repeatedly as long as the Boolean expression is TRUE. The syntax is:

WHILE <expression> DO <statement>

The syntax diagram is as follows:

WHILE statement:



The <expression> must be of type Boolean. It is evaluated before <statement> is executed, so if <expression> is initially FALSE, then <statement> is not executed at all.

Examples:

```
WHILE A > 0 DO A := A - FACTOR

WHILE ABS(TERM) > LIM DO
    BEGIN
        SINH := SINH + TERM;
        N := N + 2;
        TERM := TERM * SQR(X) / (N * (N-1))
    END
```

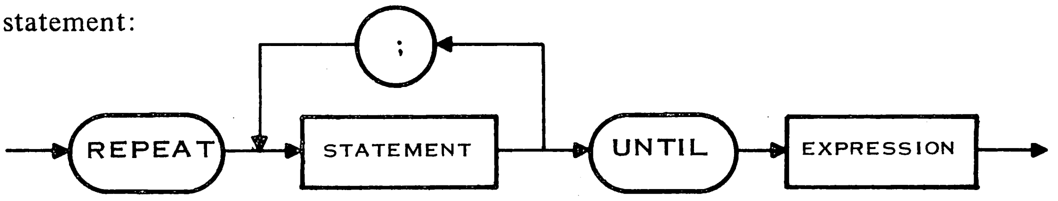
The BEGIN - END pair delimits a sequence of statements to form a single compound statement which the syntax requires.

**5.6.3 THE REPEAT STATEMENT.** The REPEAT statement provides another control structure for looping. It should be used when at least one execution of the component statements is always to be performed. (The WHILE statement allows for the possibility of no execution of the component statement.) The syntax is:

REPEAT <statement> { ; <statement> } UNTIL <expression>

The syntax diagram is as follows:

REPEAT statement:



The <expression> must be of type Boolean, and there may be a list of statements between the REPEAT and UNTIL. At runtime, the component statements are executed and then <expression> is evaluated. If it is FALSE, these actions are repeated. When the value of <expression> becomes TRUE, the looping is terminated.

Example:

```
I := 0; SUM := 0;
REPEAT
    I := I + 1;
    SUM := SUM + SQR(I)
UNTIL I := 10
```

## 5.7 MACHINE DEPENDENT AND USER-DEFINED ENUMERATION TYPES

The enumeration types CHAR, scalar, and subrange are discussed in this section. The values of type CHAR are machine dependent, and scalar and subrange types are user-defined.

**5.7.1 CHAR TYPE.** A value of type CHAR is an element of the character set used on the machine on which the program is executing. Each character is represented by its ordinal value which is the internal representation of the character on the machine. The standard function ORD may be used to determine the ordinal value of the character constant which represents a given character. For example, ORD('A') is 65 in ASCII (Appendix G).

The following are properties of the character set:

- The upper case Roman alphabet 'A' . . 'Z' is in the character set and is in order, i.e., ORD('A') < ORD('B') < . . .< ORD('Z').
- The digits '0' . . '9' are in the character set, are in order, and are contiguous, i.e., SUCC('0') = '1', SUCC('1') = '2', . . ., SUCC('8') = '9'.
- The blank character is in the character set.

Character values are written as a single character surrounded by apostrophes. (Paragraphs 6.1 and 6.7 discuss packed arrays of characters, which are used to represent strings of characters.)

There are two characters which require special treatment to be represented in strings: the apostrophe and the number sign (#). Each of these characters is represented by two consecutive characters. Also, recall from paragraph 3.2.3 that a character may be represented by a # sign, followed by its two-digit hexadecimal character code.

'A' '3' '''' '##' '+' '#0A'

Since ORD(CH) is the number which is the internal representation for the character CH and since CHR(I) is the character whose internal representation is the number I, it is easy to see that within the proper subrange, ORD and CHR are inverse functions. Suppose CH is of type CHAR and I is of type INTEGER. Then:

CHR(ORD(CH)) = CH
ORD(CHR(I)) = I

Also, the following relations are true:

```
SUCC(CH)  = CHR(SUCC(ORD(CH)))
          = CHR(ORD(CH)+1)

PRED(CH)  = CHR(PRED(ORD(CH)))
          = CHR(ORD(CH)-1)
```

**5.7.2 SCALAR TYPE.** A scalar type is a programmer-defined data type. The values of a scalar type are elements of a set of identifiers specified by the programmer. Each identifier defines a value of the type, and the order in which they are written defines the order of the type.

<scalar type>::= (<identifier>{,<identifier>})

Examples:

```
TYPE PRIMARY = (RED, YELLOW, BLUE);
     DAY = (MON, TUE, WED, THUR, FRI, SAT, SUN);
     IODEVICE = (ST01, ST02, LP01, CR01, DS01, DS02, DS03);
```

The standard function ORD returns the ordinal number of a scalar value. The ordinal number of the first identifier is zero.

Scalar values are used primarily to improve the readability of a program. For example, instead of using digits 0-6 to represent the days of the week, the definition of DAY above allows the identifiers MON, TUE, . . ., SUN to be used as values.

Examples:

Using the scalar type definitions given above,

```
ORD(MON) = 0
ORD(YELLOW) = 1
ORD(LP01) = 2
```

Also, the standard type BOOLEAN is predefined by

```
TYPE BOOLEAN = (FALSE, TRUE)
```

This defines the standard identifiers FALSE and TRUE and specifies that FALSE < TRUE.

Each identifier that appears as a value in a scalar type declaration may not be used for any other purpose within that scope. (See Section VIII for a discussion of scope.) For example, the type declaration

```
TYPE COLOR = (RED, YELLOW, BLUE);
     SHADE = (VIOLET, RED, PURPLE);
```

is illegal because the type of RED is ambiguous.

**5.7.3 SUBRANGE TYPE.** A type may be defined as a subrange of any previously defined enumeration type by specifying the least and largest values in the subrange. Thus a subrange type defines an interval over an existing enumeration type, which is called its associated enumeration type or base type. The syntax for declaring a subrange type is:

```
<type identifier> = <manifest constant> . . <manifest constant>
```

A manifest constant is a value which can be computed at compile time and is either an enumeration constant or an integer constant expression.

```
<manifest constant> ::= <enumeration constant>
    | <integer constant expression>

< enumeration constant> ::= <scalar identifier> | <character constant>
    | <Boolean constant> | [<sign>] <integer constant>
```

Integer constant expressions are described in detail in paragraph 5.9.

Examples:

```
TYPE        DEC = 0 . . N-1;
            WORKDAY = MON . . FRI;
            DISK = DS01 . . DS03;
            DIGIT = '0' . .'9';
```

The first constant (the lower bound) must be less than or equal to the second (the upper bound).

A subrange type may be used wherever the associated base type is allowed. Thus an operator defined for operands of a certain type may also be used with operands of subranges of that type. The result type of the operator will remain unchanged. For example, given that N is of the subrange type 1 . . 10, N + 25 is a legal expression whose value is of type INTEGER. In the same manner, associated base types determine the validity of assignment statements. For example, given the declaration

```
VAR  X : 1 . . 10;
     Y : 0 . . 30;
     Z : 20 . .30
```

The assignments

```
X := Z;
Z := Y;
Y := Z;
```

are all legal statements because the associated base types are all INTEGER. Of course some of the above assignment statements imply an "out of range" assignment for certain variable values. The compiler has an option available which may be turned on to check at runtime for illegal assignment to a variable of subrange type. If the value to be assigned is outside the subrange and the option is enabled, a runtime error occurs.

## 5.8 CASE AND FOR STATEMENTS

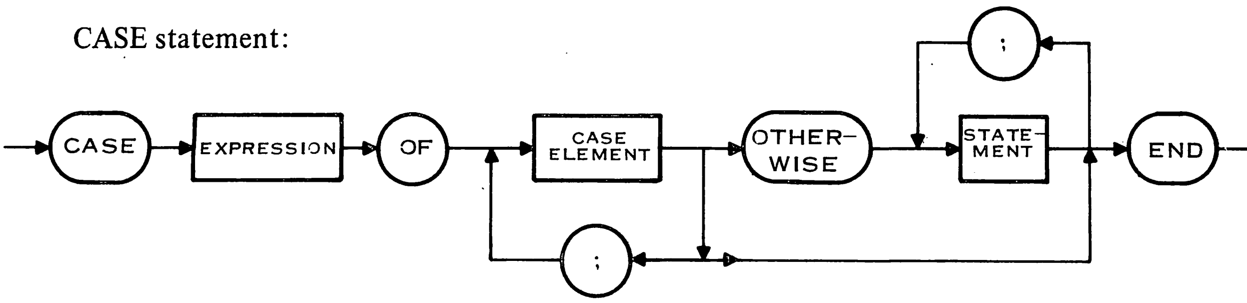
The enumeration types which have been discussed in the previous sections have important applications in the CASE and FOR statements, which are discussed next. While the IF statement uses a Boolean expression to select between two statements, the CASE statement uses an expression of any enumeration type to select from an arbitrarily large group of alternative statements. The FOR statement uses a control variable of any enumeration type to determine the number of iterations or repetitions of a loop. Using scalar types in these statements is very helpful in documentation, since descriptive identifiers help explain the purpose of the program.

**5.8.1 THE CASE STATEMENT.** The CASE statement contains an expression and a list of statements, each one being preceded by a list of constants. The expression is called the selector and its value at runtime determines which if any of the component statements is executed. The syntax of the CASE statement is:

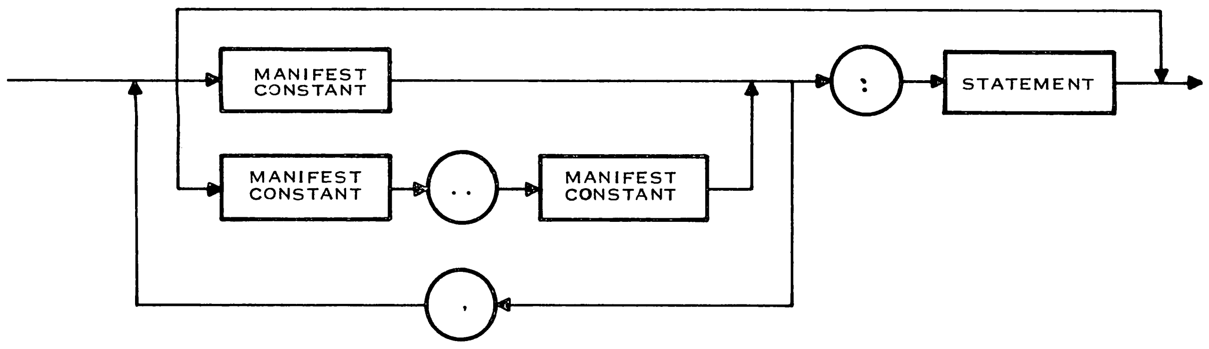
```
<case statement> ::=

     CASE <expression> OF
          <case element> {; <case element>}
       [OTHERWISE <statement> {;<statement>}]
     END

<case element> ::= <case label list> : <statement> | <empty>

<case label list> ::= <case label> {, <case label>}

<case label> ::= <manifest constant>
     | <manifest constant> . .<manifest constant>
```

The syntax diagrams are as follows:

CASE statement:



CASE element:



The <expression> is called the selector and must be of enumeration type. At runtime, the selector expression is evaluated, and the component statement labeled with the value of the selector is executed. If no such label appears within the CASE statement, the component statement or statements after the keyword OTHERWISE are executed, and if no OTHERWISE clause exists, a runtime error occurs.

The subrange form

<manifest constant> . . <manifest constant>

may be used as an abbreviation to specify all of the values greater than or equal to the first enumeration constant and less than or equal to the second enumeration constant. The first enumeration constant must be less than or equal to the second. Any value may be in at most one CASE label.

Examples:

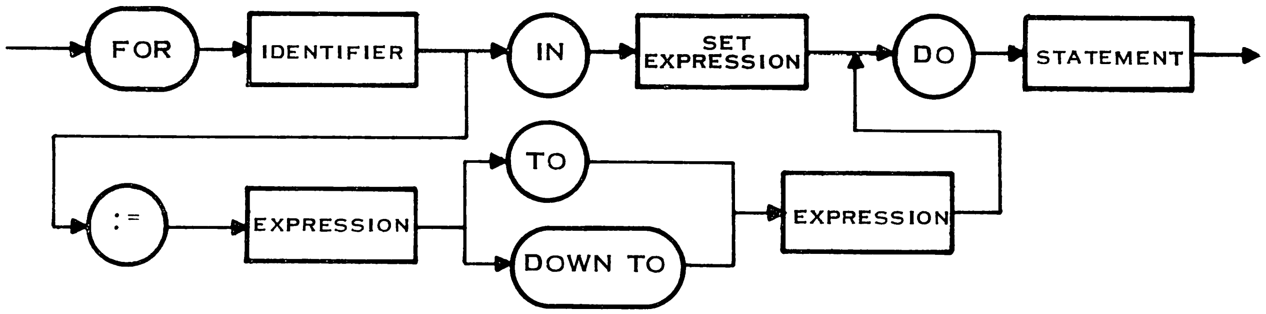
```
CASE NUM OF
     1,3,5     : X := X + 1;
     6..10,20  : X := X * 10;
     15        : X := X DIV Y
END

CASE CH OF
     '0' . . '9': N := ORD(CH) - ORD('0');
     'A' . .'F': N := ORD(CH)-ORD('A') + 10
     OTHERWISE WRITELN('ILLEGAL HEX DIGIT'); N := -1
END
```

**5.8.2 THE FOR STATEMENT.** The FOR statement specifies that its component statement is to be executed repeatedly for a sequence of values which are assigned to the control variable of the FOR statement. The syntax is:

```
<for statement> ::= FOR <control variable> <generator> DO <statement>

<generator> ::= IN <set expression>
              | : = <initial value> TO < final value>
              | := <initial value> DOWNTO <final value>
```

The syntax diagram is as follows:

FOR statement:



The <initial value> and <final value> may be any expression of an enumeration type.

Sets are discussed in paragraph 6.4.

Examples:

```
FOR J := -5 TO Z MOD 10 DO
    S:= S + SQR(J)

(*ASSUME TYPE D = (MON, TUE, WED, THUR, FRI, SAT, SUN) *)
FOR DAY := MON TO FRI DO
    BEGIN READ(HRS); PAY := PAY + HRS*RATE END

FOR K IN ERRORSET DO
    WRITELN(M[K])
```

The <generator> expression is evaluated once, when the loop is entered. Changing any of the values in the generator (either the set expression or the initial and final values) within the FOR statement does not affect the number of times the component statement is executed. The control variable is assigned the <initial value> when the FOR statement is entered, and is incremented (decremented in the case of DOWNTO) after each repetition of the component statement. The last repetition of the component statement is when the control variable is equal to the <final value>. If the initial value is greater than the final value in the case of TO (or less than, in the case of DOWNTO), then the component statement is not executed at all. In the case of a set generator, the values are taken in the order of the underlying base type of the set.

Example:

```
N := 10;
FOR J := 1 TO N DO N := N + 1
```

This loop is repeated 10 times and the value of N is 20 when the loop terminates.

Example:

```
VAR X: SET OF A..Z;
        .
        .
        .
BEGIN
        .
        .
        .
    FOR M IN X DO
      A [M] := 0
```

The FOR statement is equivalent to the following FOR statement:

```
FOR M:= A TO Z DO
  IF M IN X THEN
    A [M]:= 0
```

The control variable is an implicitly declared variable, that is, it is declared by its appearance in the FOR statement. If the control variable does appear in the declaration part of the program, the two identifiers are distinct and a compiler warning is given if the proper option is enabled. (This is an enhancement to standard Pascal.)

Example:

```
PROGRAM FOREXAMPLE;
VAR I, N, X : INTEGER;
BEGIN
    READ(N);
    . . .
    FOR I := 1 TO N DO
        BEGIN READ(X);
            WRITE(X)
        END:
    WRITELN
END.
```

The control variable I of the FOR statement in this example is distinct from the variable I from the VAR declaration. Any occurrence of I within the FOR statement is a reference to the control variable. The control variable I is not accessible outside the body of the FOR statement.

Furthermore, it is not legal to change the value of the control variable within a FOR statement, either by an assignment statement or by passing it as a variable parameter to a procedure (see Section VIII). An attempt to do so will result in an error.

## 5.9 EXPRESSIONS

```
<factor> ::= <unsigned constant> | <variable>
    | <function identifier> [([<expression > {, <expression>}])]
    | ( <expression>) | <set>

<unsigned constant> ::= <constant identifier> | <real constant>
    | <integer constant> | <string constant> | <character constant>
    | <Boolean constant> | <scalar identifier> | NIL
    | <fixed point constant> | <decimal constant>

<set> ::= "["<element list>"]"

<element list> ::= <element> {, <element>} | <empty>

<element> ::= <expression> | <expression>. .<expression>

<term> ::= <factor> | <term> <multiplying operator> <factor>

<multiplying operator> ::= * | / | DIV | MOD

<simple expression> ::= <term> | <adding operator> <term>
    | <simple expression> <adding operator> <term>

<adding operator> ::= + | -

<Boolean primary> ::= <simple expression>
    | <Boolean primary> <relational operator> <simple expression>
```

<relational operator> ::= = | <> | < | <= | > | >= | IN

<Boolean factor> ::= <Boolean primary> | NOT <Boolean primary>

<Boolean term> ::= <Boolean factor>
    | <Boolean term> AND <Boolean factor>

<expression> ::= <Boolean term> | <expression> OR <Boolean term>

Figure 5-1 contains syntax diagrams for these expressions.

Examples:

| | |
|---|---|
| Factors: | 5<br>I<br>SIN(X)<br>[RED, GREEN] |
| Terms: | X * Y<br>I DIV J<br>X/Y<br>VAL MOD 6 |
| Simple Expressions: | X + Y<br>-Z<br>B + SQRT(B*B-4*A*C) |
| Boolean Term: | NOT BOOL<br>X = 3 AND Y >= 2<br>1<br>7.2 |
| Expressions: | X = 3 OR Y <> 2<br>P >= Q<br>RED IN HUES<br>(X + Y) DIV M |

An expression may contain both real and integer values. In such mixed mode expressions, the rule is that if either operand of a binary operator is real, then the result of the operation is real. (The operator / never produces an integer result.) The operators MOD and DIV require that both operands be INTEGER.

Mixed Mode Expressions:

```
(* assume VAR X : REAL; J : INTEGER *)
          J*15.8
          X<=J
```

| | | | |
|---|---|---|---|
| Illegal expressions: | A*-5 | Should be | A*(-5) |
| | --T | | -(-T) |

Several features of expressions should be noted: first, the syntax of TI Pascal, by itself, permits some forms of expressions which have no reasonable interpretation and hence are semantically illegal. For example,

NOT 5 * 2

is a syntactically correct Boolean factor and therefore an expression, but NOT makes sense only when applied to a Boolean value, so this will result in an error at compile time.

Another point has to do with the way in which Boolean expressions are evaluated. There is a frequent need for Boolean expressions such as

I <= N AND A[I] <> X

which is intended to be TRUE if I is less than or equal to N and the Ith value of array A is not equal to X, and FALSE otherwise. The problem is that if I is greater than N and A has upper bound N, the above expression is illegal because A[I] is undefined for I > N. This problem is solved in TIP by means of what is called short-circuit evaluation of Boolean expressions. In a Boolean expression of the form

X AND Y

if X is FALSE then Y is not evaluated and the value of the entire expression is FALSE. In a Boolean expression of the form

X OR Y

if X is TRUE, then Y is not evaluated and the value of the entire expression is TRUE. (This is an enhancement to standard Pascal.)

Constant expressions may be evaluated at compile time and may, except where statement labels and more restricted integer constant expressions (see next paragraph) are required, occur anywhere that a constant may appear. Constant expressions are the same as expressions except that they may not contain variables or function designators.

Integer constant expressions are defined in the same way with the exception that they do not have the operator '/'. They may be used in the specification of array, set, and subrange bounds. All operands of an integer constant expression must be of type INTEGER or LONGINT or a subrange thereof.

## 5.10 REAL TYPE

The type REAL may be used to represent real values with 6-7 decimal digits of precision. The range of absolute values which can be represented is about 1.0E-78 to 1.0E75. The default precision may be overridden by specifying the desired precision in decimal digits. This is done by enclosing the precision in parentheses after the symbol REAL. The maximum precision supported is 16 decimal digits. For example,

REAL(12)

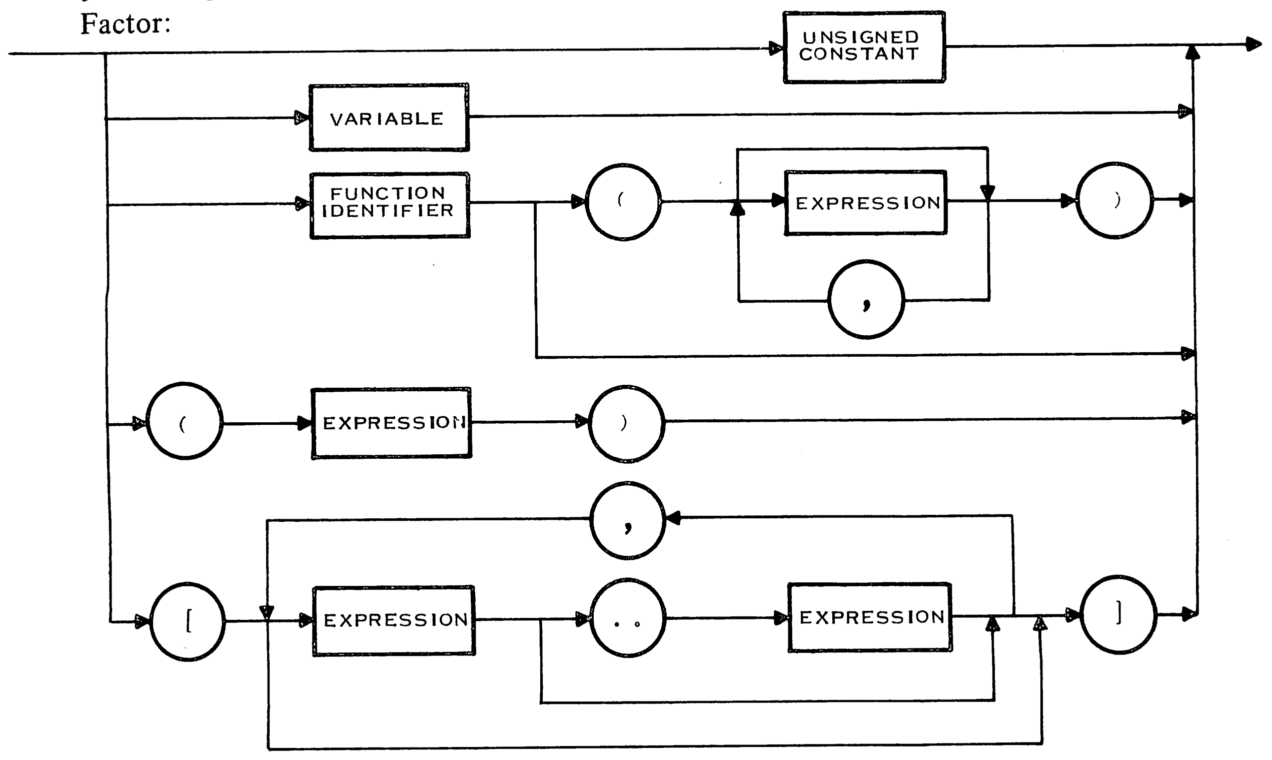
will cause the compiler to allocate enough space for values of type REAL(12) so that 12 decimal digits of precision may be obtained.

The following operators accept operands of type REAL (of any precision) and yield a real value (with precision equal to the maximum precision of the operands):

| | |
|---|---|
| * | multiply |
| / | divide |
| + | unary plus or add |
| - | negate or subtract |

The syntax diagrams are as follows:

Factor:



Unsigned constant:

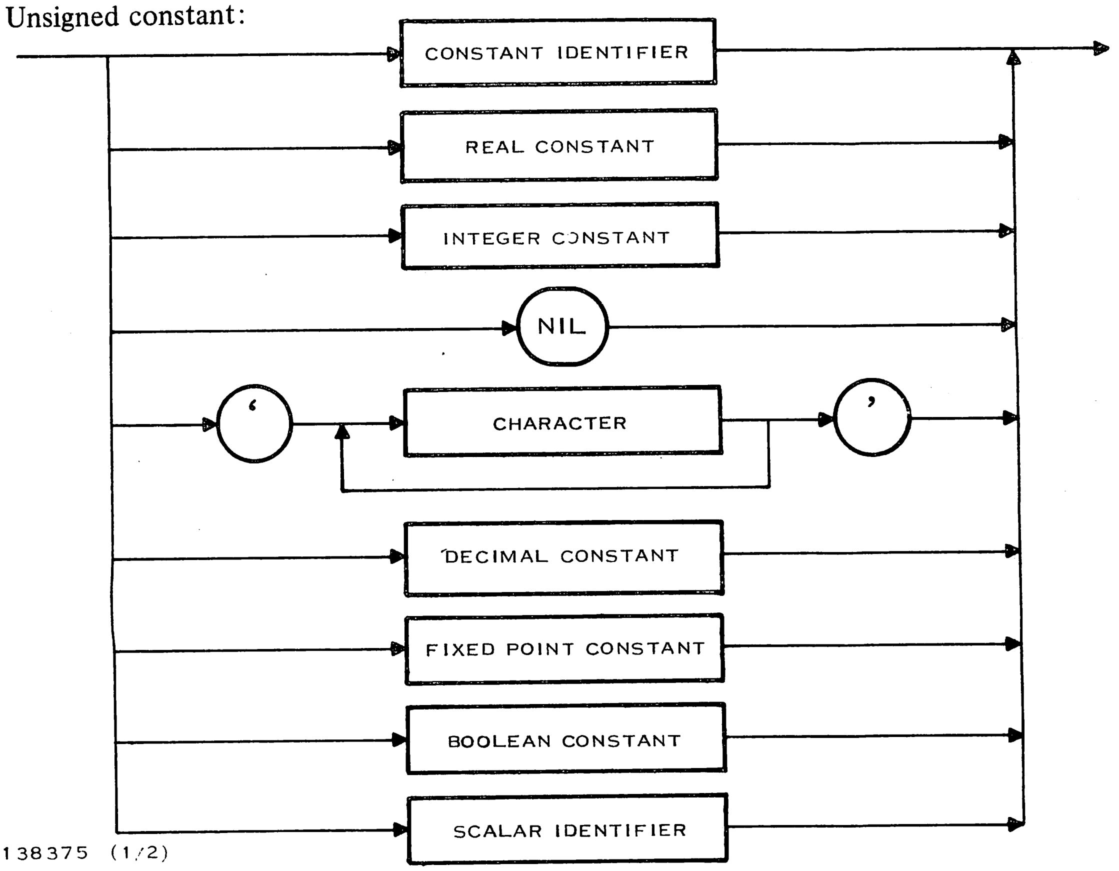
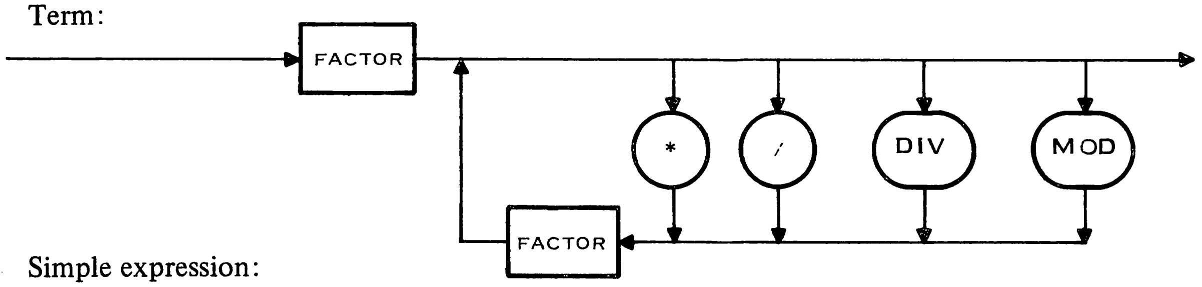
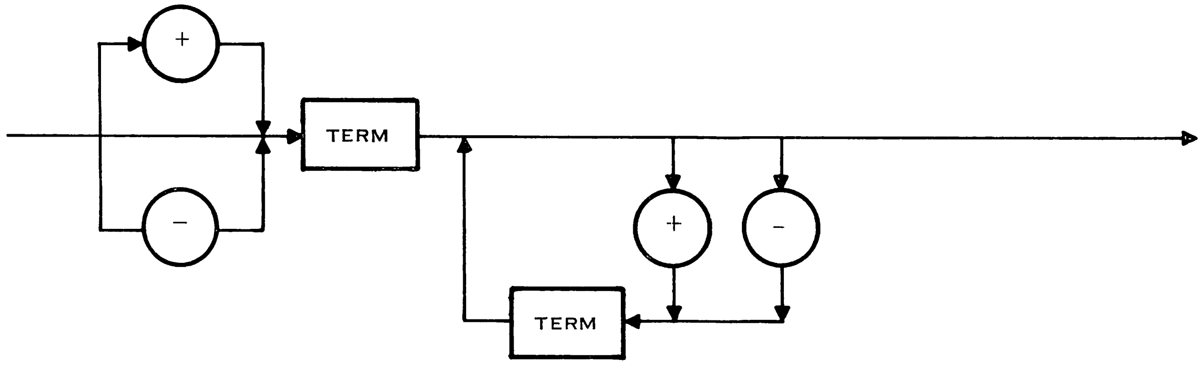


**Figure 5-1. Syntax Diagrams for Expressions (Sheet 1 of 2)**

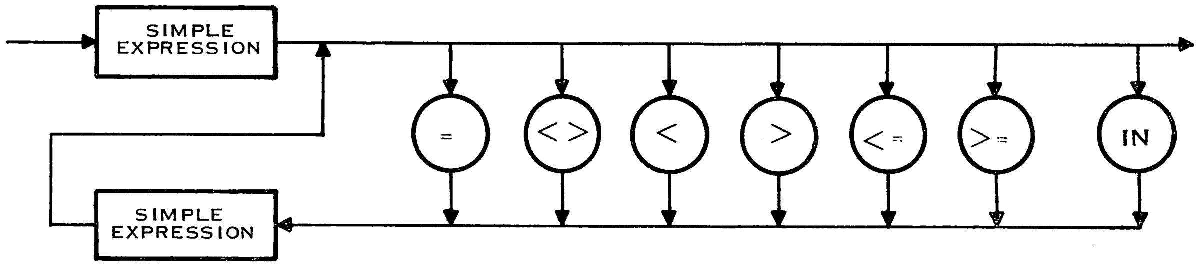
Term:



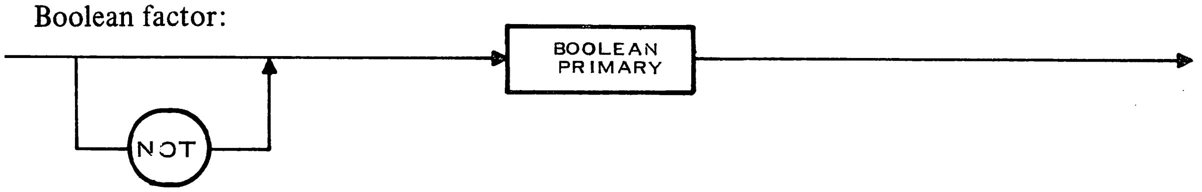
Simple expression:



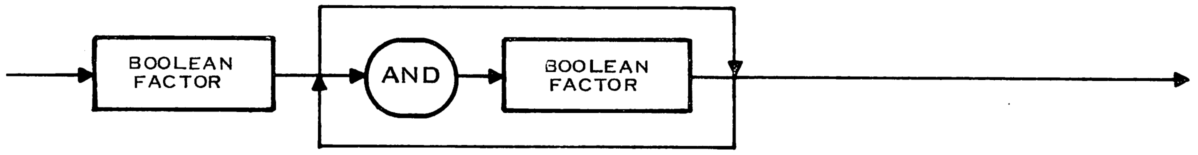
Boolean primary:



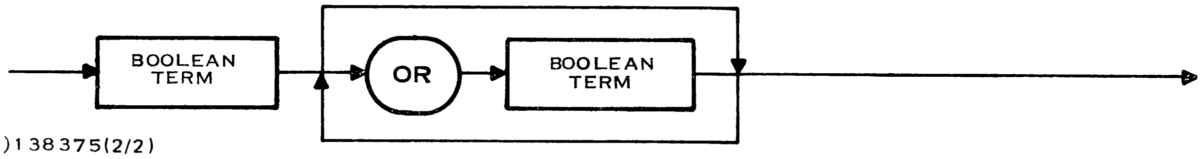
Boolean factor:



Boolean term:



Expression:



(A)138375(2/2)

**Figure 5-1. Syntax Diagrams for Expressions (Sheet 2 of 2)**

The assignment operator may be used to assign a REAL value to a REAL variable. Also, the relational operators are defined for REAL operands just as for enumeration operands (see paragraph 2.2.1.1).

Standard functions accepting a REAL argument and yielding a REAL result with precision the same as that of the argument:

| | |
|---|---|
| ABS(X) | absolute value |
| SQR(X) | X squared |

Standard functions with REAL or INTEGER argument in which the REAL result has the same precision as that of the argument:

| | |
|---|---|
| SIN(X) | Trignometric sine of X in radians |
| COS(X) | Trigonometric cosine of X in radians |
| ARCTAN(X) | Trigonometric inverse tangent in radians |
| LN(X) | Natural logarithm if X > 0; otherwise a runtime error occurs. |
| EXP(X) | Exponential function -- 2.718281828 to the power X. |
| SQRT(X) | Square root if X >= 0; otherwise a runtime error occurs. |

Standard functions with a REAL argument yielding INTEGER results are:

| | |
|---|---|
| TRUNC(X) | The result is the whole part of X, i.e., the fractional part of X is discarded. |
| ROUND(X) | The result is X rounded to the nearest integer, i.e., TRUNC(X + 0.5) if X >= 0 or TRUNC(X - 0.5) if X < 0. |

Similar standard functions, LTRUNC and LROUND, yield a result of type LONGINT.

The standard functions which permit conversion of arguments of type REAL are:

| | |
|---|---|
| DEC(P,Q,X) | The result is the decimal value with precision (P,Q) (paragraph 5.12) corresponding to the REAL X. |
| FLOAT(X,P) | The result is the real value with precision P corresponding to the REAL X. |
| FIX(P,Q,X) | The result is the fixed-point value with precision (P,Q) (paragraph 5.11) corresponding to the REAL X. |

The syntax for real numbers is given in paragraph 3.2.2. Scientific notation is allowed with real constants in which case the letter E or Q is used to specify a power of 10 scale factor. The letter E denotes the predefined default precision (32 bits) while the letter Q specifies the maximum precision available (64 bits).

Examples:

```
5E-3
3.14159268Q0
```

## 5.11 FIXED TYPE

FIXED types allow fractional values to be represented without using the generality of the real type representation. A value of type FIXED is a scaled binary number. Its precision consists of two parts: P and Q. P specifies the total number of binary digits which includes the fractional digits (but does not include the sign bit); Q is the scale factor which specifies the binary point alignment, i.e., the position of the binary point relative to the rightmost binary digit of the binary number. Precision is stated by two decimal integers P and Q, separated by a comma and enclosed in parentheses. P must be unsigned but Q may be signed. A positive value for Q indicates the number of binary digits following the binary point; a negative value indicates the number of imaginary zero binary digits following the number and preceding the binary point.

The maximum precision supported on the Model 990 Computer is 31 bits.

Fixed point constants have the following syntax:

```
<fixed-point constant> ::= <digits>. <digits> F |
    <digits> F | <binary digits>. <binary digits> B |
    <binary digits> B
```

The following operators are defined for fixed point operands of (possibly) different precisions and yield a fixed point value whose precision depends on the operator.

| | |
|---|---|
| + | unary plus or add |
| - | negate or subtract |
| * | multiply |
| / | divide |

The assignment operator may be used to assign a fixed point value to a fixed point variable, and a range check option is available to determine loss of most significant digits at runtime.

Whenever an expression of type FIXED is assigned to a fixed point variable, the declared precision of the variable is maintained. The assigned item is aligned on the binary point. Leading zeros are inserted if the assigned item contains fewer integer digits than declared; trailing zeros are inserted if it contains fewer fractional digits. An error occurs (if the range check option is enabled) if the assigned item contains too many integer digits; truncation on the right occurs, with rounding, if it contains too many fractional digits.

The relational operators (<, =, >, <=, <>, >=) are defined for fixed point operands and yield a Boolean value.

Standard functions accepting a fixed point argument and yielding a real result are:

ABS(X)

SQR(X)

SIN(X)

COS(X)

ARCTAN(X)

LN(X)

EXP(X)

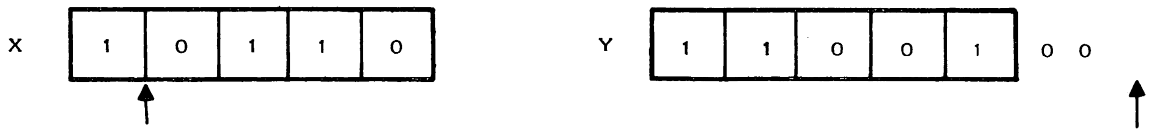
SQRT(X)

The standard functions which permit conversion of arguments of type FIXED are:

| | |
|---|---|
| TRUNC(X) | The result is X truncated to the nearest INTEGER. |
| ROUND(X) | The result is X rounded to the nearest INTEGER. |
| FLOAT(X,P) | The result is the real value with precision P corresponding to the fixed point X. |
| DEC(P,Q,X) | The result is the decimal value with precision (P,Q) corresponding to the fixed point X. |
| FIX(P,Q,X) | The result is the fixed point value with precision (P,Q) corresponding to the fixed point X. |

Examples:

```
VAR X: FIXED(5, 4);
    Y: FIXED(5, -2)

X := 1.011B;      Y := 1100100B
```



## 5.12 DECIMAL TYPE

DECIMAL types are suitable for applications requiring that operations be performed using decimal arithmetic. A decimal type represents a value of the standard type DECIMAL. The syntax is:

```
<decimal constant> ::= <digits>. <digits> D | <digits> D
```

DECIMAL types allow fractional decimal values to be represented without converting them to real type representation. A value of type DECIMAL is a scaled decimal number. Its precision consists of two parts; P and Q. P specifies the total number of decimal digits which includes the fractional digits; Q is the scale factor which specifies the decimal point alignment, i.e., the position of the decimal point relative to the rightmost digit of the decimal number. Precision is stated by two decimal integers P and Q, separated by a comma and enclosed in parentheses. P must be unsigned but Q may be signed.

The maximum precision is 15 decimal digits on the Model 990 Computer.

The following operators are defined for decimal operands of (possibly) different precisions and yield a decimal value whose precision depends on the operator.

| | |
|---|---|
| + | unary plus or add |
| - | negate or subtract |
| * | multiply |
| / | divide |

The assignment operator may be used to assign a decimal value to a decimal variable, and a range check option is available to determine loss of most significant digits at runtime.

Whenever an expression of type DECIMAL is assigned to a decimal variable, the declared precision of the variable is maintained. The assigned item is aligned on the decimal point. Leading zeros are inserted if the assigned item contains fewer integer digits than declared; trailing zeros are inserted if it contains fewer fractional digits. An error occurs if the assigned item contains too many digits and the range check option is enabled; truncation on the right occurs if it contains too many fractional digits. An option may be enabled to cause rounding in addition to truncation.

The relational operators (<, =, >, <=, <>, >=) are defined for decimal operands and yield a Boolean value.

Standard functions accepting a decimal argument and yielding a real result are:

ABS(X)

SQR(X)

SIN(X)

COS(X)

ARCTAN(X)

LN(X)

EXP(X)

SQRT(X)

The standard functions which permit conversion of arguments of type DECIMAL are:

| | |
|---|---|
| TRUNC(X) | The result is X truncated to the nearest INTEGER. |
| ROUND(X) | The result is X rounded to the nearest INTEGER. |
| FLOAT(X,P) | The result is the real value with precision P corresponding to the decimal X. |
| DEC(P,Q,X) | The result is the decimal value with precision (P,Q) corresponding to the decimal X. |
| FIX(P,Q,X) | The result is the fixed-point value with precision (P,Q) corresponding to the decimal X. |

The following examples show the equivalent COBOL representation for several DECIMAL types:

| TIP | COBOL | MAXIMUM VALUE | SMALLEST INCREMENT |
|---|---|---|---|
| DECIMAL (4,0) | PICTURE S9(4) | 9,999. | 1. |
| DECIMAL (8,2) | PICTURE S9(6)V99 | 999,999.99 | .01 |
| DECIMAL (5,-3) | PICTURE S9(5)P(3) | 99,999,000. | 1000 |
| DECIMAL (4,-6) | PICTURE SVPP9999 | 0.009999 | .000001 |

## SECTION VI

## STRUCTURED DATA TYPES

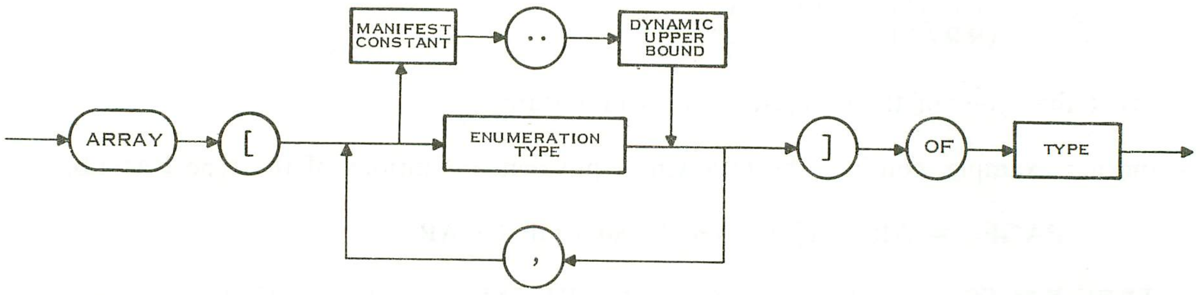
### 6.1 ARRAY TYPE

An array consists of a fixed number of components which are all of the same type, called the component type. Components of the array are designated by specifying their relative positions in the array, using expressions of the index type. An ARRAY type has the following syntax:

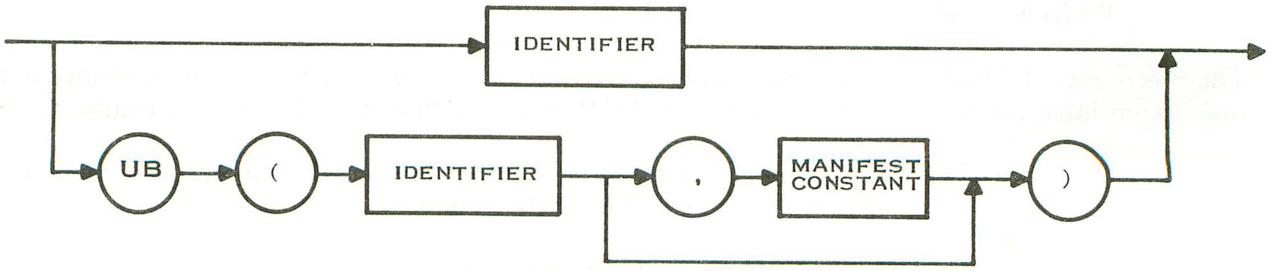
```
<array type> ::=
     ARRAY "[" <index type> {, <index type>}"]" OF <component type>

<index type> ::= <static index type> | <dynamic index type>

<static index type> ::= <enumeration type> | <type identifier>

<dynamic index type> ::= <manifest constant> . . <dynamic upper bound>

<dynamic upper bound> ::= <entire variable>
     | UB( <dynamic array variable>[, <manifest constant>])

<entire variable> ::= <identifier>
```

The syntax diagrams are as follows:

Array type:



Dynamic upper bound:



The <component type> may be any type, simple or structured, except that it may not be FILE. In particular, <component type> may itself be an ARRAY type. The number of <index types> in the declaration determines the dimension of the array. There is no limit to the number of dimensions which an array may have. Each <index type> must be an enumeration type: INTEGER, LONGINT, BOOLEAN, CHAR, subrange, or scalar.

Examples of one-dimensional arrays:

```
    (* RECALL TYPE WORKDAY = MON. . FRI *)
    LATEDAYS = ARRAY[ WORKDAY ] OF BOOLEAN; (*LENGTH 5*)
    LIST = ARRAY[ '0' . . '9' ] OF LATEDAYS; (*LENGTH 10*)
    TRANSLATE = ARRAY[ CHAR ] OF CHAR (*LENGTH 256*)
```

TWO DIMENSIONAL ARRAY:

```
    TYPE TABLE = ARRAY[ 0. . 10, -50. . 50 ] OF INTEGER;
```

Arrays with two or more dimensions are called multidimensional arrays. These may be described in terms of one-dimensional arrays because the type

```
                 A1 = ARRAY[ T1, T2, . . ., Tn ] OF <type>
```

is equivalent to

```
        A1 = ARRAY[T1] OF ARRAY[T2] OF . . . OF ARRAY[Tn] OF <type>
```

Another way of expressing exactly the same type is:

```
    TYPE An = ARRAY[ Tn ] OF <type>;
        . . .
        A2 = ARRAY[ T2 ] OF A3;
        A1 = ARRAY[ T1 ] OF A2
```

(Note that the order of these declarations is important.)

For another example, consider the following equivalent definitions of the type PAGES:

```
    TYPE PAGES = ARRAY[ 1. . 66, 1. .80 ] OF CHAR

    TYPE PAGES = ARRAY[ 1. . 66 ] OF ARRAY[ 1. .80 ] OF CHAR

    TYPE LINE = ARRAY[ 1. . 80 ] OF CHAR;
        PAGES = ARRAY[ 1. . 66 ] OF LINE
```

Therefore, even if PAGES is declared as a two-dimensional array of characters, it can be treated as a one-dimensional array of LINEs, where each LINE is a one-dimensional array of characters.

A particular element of a one-dimensional array is denoted by writing the array identifier followed by a bracketed expression of a type compatible with the index:

```
                    <identifier> [ <expression> ]
```

The expression is called an index or a subscript. If the index type is a subrange, the value of the expression must fall within this subrange. A compiler option is available which may be used to generate runtime checks for array indices out of bounds.

Just as there are several equivalent ways to declare a multidimensional array, there are several equivalent ways to access components of a multidimensional array. Given the array PAGES declared above:

| | |
|---|---|
| PAGES[ 14 ] | denotes the 14th line of the array, i.e., the one-dimensional array of characters which is the 14th component of PAGES. |
| PAGES[ 5, 21 ] | denotes the 21st character of the 5th line. |
| PAGES[ 5 ][ 21 ] | also denotes the 21st character of the 5th line. |

Operators:

The basic operator between array operands of compatible type is assignment (:=). (See paragraph 6.7 for packed arrays and additional operators which apply to packed arrays of characters.) For example, given any of the declaration of PAGES above and the declaration

```
VAR CARD : ARRAY[ 1. . 80 ] OF CHAR
```

the following are legal:

```
PAGES[ 12, 8 ] := 'M';
PAGES[ 36 ] := CARD;
```

The standard function which applies to an array argument is:

| | |
|---|---|
| UB(A, D) | the result is the upper bound of the Dth dimension of the array A. |
| UB(A) | the same as UB(A, 1). |

D must be an integer constant.

For example, UB(PAGES, 2) = 80, and UB(PAGES) = 66. The type of the result is the same as the Dth index type. Dimensions are numbered left to right, starting with 1. As another example, UB(LATEDAYS) = FRI (recall the first example in this section).

**6.1.1 ARRAY PROCEDURES.** Standard procedures for arrays (see paragraph 6.7 for a discussion of packed arrays):

```
PACK(A, I, Z)        means FOR J := U TO V DO
                           Z[ J ] := A[ J-U+I ]

UNPACK(Z, A, I)      means FOR J := U TO V DO
                           A[ J-U+I ] := Z[ J ]
```

where

A is a variable of type ARRAY[ M. .N ] OF T1,
Z is a variable of type PACKED ARRAY [ U. .V ] OF T2,
T1 and T2 are compatible types, and (N-M) >= (V-U)

UNPACK allows a packed array to be unpacked so its components may be efficiently accessed. PACK allows an unpacked array to be packed.

**6.1.2 STATIC AND DYNAMIC ARRAYS.** The number of components in an array is fixed, and in each of the previous examples this number has been fixed at compile time. Arrays of this sort are said to be static. It is also possible to fix the number of components at runtime in TIP, and arrays of this sort are called dynamic. Arrays are specified to be dynamic by means of dynamic index types. The number of components of a dynamic array is determined upon entry to the block (program or routine) containing the array type declaration. (See Section VIII for a discussion of routines.) For multidimensional arrays, if at least one index type is a dynamic index type, then the array is dynamic. Only the upper bound of a dynamic index type may be specified at runtime. The lower bound must be a *manifest* constant, which is fixed at compile-time. The upper bound may be an entire variable (not a component of an array or record and not a referenced variable), or the upper bound may be specified by means of the UB function applied to an argument which is an array variable. An entire variable appearing as dynamic upper bound must be either a nonlocal variable or a formal procedure parameter. (Also, see paragraph 8.5.2 for an example of dynamic array parameters.)

Examples of Dynamic Arrays:

```
TYPE VEC = ARRAY[ 1. . N ] OF INTEGER;
    TABLE = ARRAY[ 0. . UB(VEC) ] OF REAL
    BOOK = ARRAY[ 1. . N, 1. . 54, 1. .60 ] OF CHAR
```

## 6.2 RECORD TYPE

The record type declaration defines a record. In TI Pascal, a record may have a fixed part, a fixed part and a variant part, or a variant part alone.

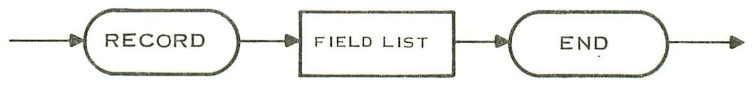
**6.2.1 RECORDS.** A record consists of a number of components of possibly different type called fields. Each field must be given a distinct name, called the field identifier, which is used to reference the individual component, and a type must be specified for each field. A field may be in the fixed part of the record variable, or it may be in the variant part. The variant part of a record is a list of alternative forms which the field may take. A record may have a fixed part, a variant part, or both, but the variant part must be last if it appears.

Syntax:

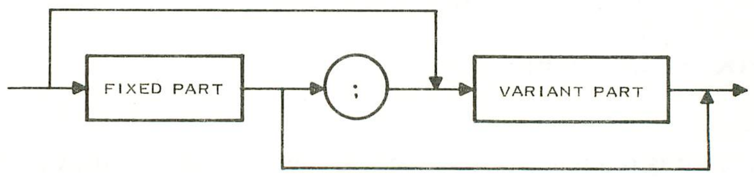
```
<record type> ::= RECORD <field list> END

<field list> ::= <fixed part> | <fixed part> ; <variant part> | <variant part>

<fixed part> ::= <record section> {; <record section>}

<record section> ::= [<field identifier> {, <field identifier> } : <type>]

<variant part> ::= CASE <tag field> <type identifier> OF <variant> {; <variant>}

<variant> ::= [<case label list> : (<field list>)]

<case label list> ::= <case label> {, <case label>}

<case label> ::= <manifest constant> | <manifest constant>. .<manifest constant>

<tag field> ::= [<identifier> :]
```

The syntax diagrams are as follows:

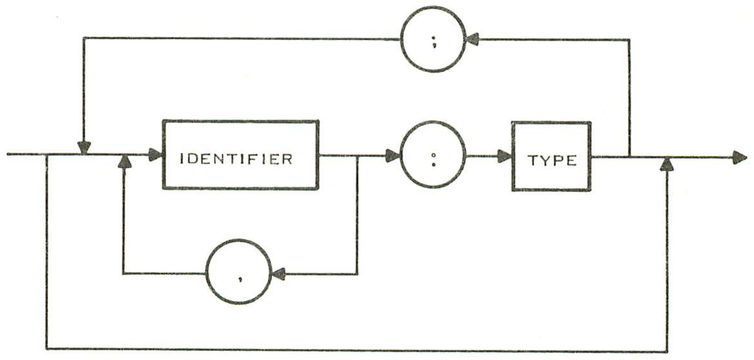
Record type:



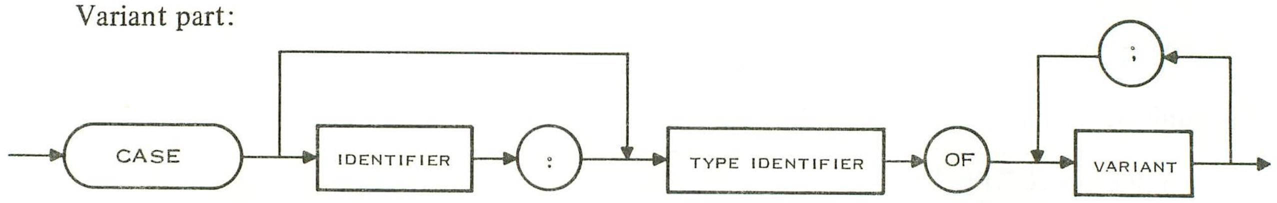
Field list:



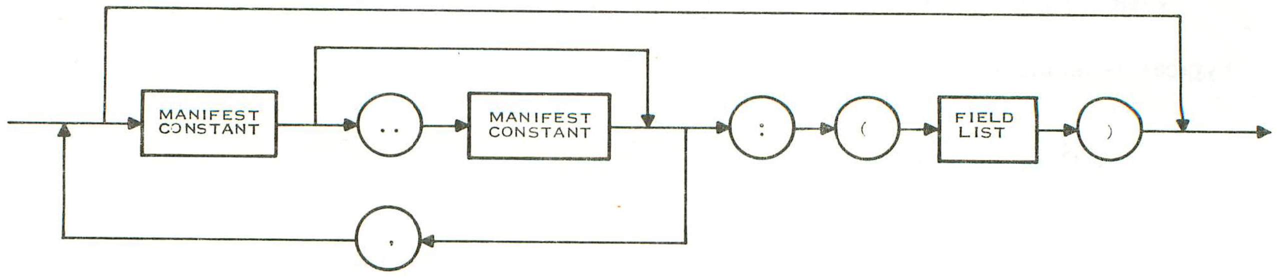
Fixed part:



Variant part:



Variant:



The type of a field may not be a dynamic array, dynamic set, or file. The assignment operator (:=) applies to operands which are compatible records. (Type compatibility is discussed in paragraph 6.8.) No other operator applies to records.

Examples:

```
TYPE COMPLEX = RECORD RE, IM : REAL END;
    DATA = RECORD
        MONTH : (JAN, FEB, MAR, APR, MAY, JUN,
                 JUL, AUG, SEP, OCT, NOV, DEC);
        DAY   : 1. . 31;
        YEAR  : INTEGER
    END
```

A component of a record is referenced by the record identifier followed by a period followed by the appropriate field identifier:

<record variable>.<field identifier>

Examples:

```
VAR X, Y, Z : COMPLEX:
    START, FINISH : DATE;

START.DAY := 11;
START.MONTH := APR;
```

Multiplication of complex numbers:

```
Z.RE := X.RE * Y.RE - X.IM * Y.IM;
Z.IM := X.RE * Y.IM + X.IM * Y.RE;
```

The assignment statement

```
Y := X
```

is equivalent to

```
Y.RE := X.RE;
Y.IM := X.IM
```

An array of records could be defined as:

```
VAR CPLX : ARRAY[ 1. . N ] OF COMPLEX
```

Typical assignments might be:

```
CPLX[I] := Z;
CPLX[I].RE := Z.RE
```

A record containing an array could be defined as:

```
VAR REC : RECORD
    KEY : 1. . 3;
    CODE : ARRAY[1. . 3] OF CHAR
  END:
```

Usage:

```
REC.KEY := N;
REC.CODE[N] := 'X';
REC.CODE[REC.KEY] := 'T';
```

**6.2.2 VARIANTS.** In some situations it is convenient to have a record type which allows individual records to have some differences in their structure. For example, an employee record for college graduates might contain certain information which is not needed in records for noncollege graduates. The variant part of a TI Pascal record provides this capability with a form similar to a CASE statement in which a selector or tag field can be used to indicate the variant which is currently used.

Example:

```
TYPE ED = (HS, COLLEGE);
VAR EMPLOYEE : RECORD
    NAME   : STRING;
    ID     : INTEGER;
    STATUS : (EXEMPT, NONEXEMPT);
    CASE EDUCATION : ED OF
        HS : (SPECIALTY  : ARRAY[1. . 20] OF CHAR;
                     GPA : REAL);
        COLLEGE : (CODE : INTEGER;
                 DEGREE : (BS, MS, PHD, NONE);
                    AVE : REAL)
    END
```

This record contains four fixed fields: NAME, ID, STATUS, and EDUCATION, and a variant part of either the two fields SPECIALTY and GPA or the three fields CODE, DEGREE, and AVE. The EDUCATION field is the tag field. Note that both variants have a real field representing the grade point average (GPA and AVE). Different names must be used for these, since all field identifiers must be distinct at a given level. Actually, since this field is common to all variants, it should be moved to the fixed part of the record.

The tag field need not be included, so that the field identifier EDUCATION and the colon that follows could be omitted in the previous example. However, the type of the tag must be specified. Also, each case label must be a unique nonnegative enumeration constant or integer constant expression of the same type as the tag field, and the case labels must be disjoint.

The tag type specification merely determines the type which is used to label the variants. Which variant is selected is actually determined by the field identifier which is used, which explains why the tag field may be omitted and why all field identifiers at a given level, even in the variant part, must be distinct.

A compiler option is available, however, to check at runtime that the value of the tag field, if present, corresponds with the variant which is selected.

Example:

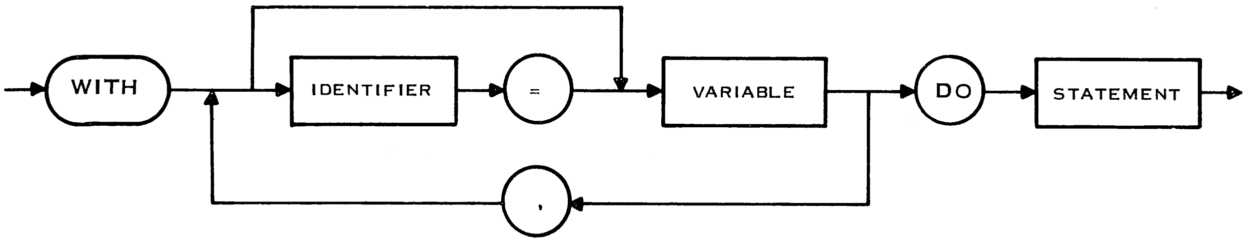
```
EMPLOYEE.ID
EMPLOYEE.EDUCATION
EMPLOYEE.GPA
EMPLOYEE.DEGREE
```

## 6.3 WITH STATEMENT

Repeated references to components of the same record can be considerably simplified by using the WITH statement. The record is specified at the beginning of the WITH statement, and within the scope of the WITH statement the record's component can be denoted by the field identifiers alone, with the record identifier omitted. The syntax is:

```
<with statement> ::= WITH <with variable list> DO <statement>

<with variable list> ::= <with variable> {, <with variable>}

<with variable> ::= <record variable> | <identifier> = <record variable>
```

WITH statement:



Example:

```
(* ASSUME VAR STARTDATE : DATE *)

WITH STARTDATE DO
    BEGIN MONTH := MAY;
        DAY := 16;
        YEAR := 1977
    END
```

This has the same effect as:

```
STARTDATE.MONTH := MAY;
STARTDATE.DAY := 16;
STARTDATE.YEAR := 1977
```

Another example is:

```
(* ASSUME VAR MEMBER : ARRAY[1. . 80] OF EMPLOYEE *)
(* EMPLOYEE DEFINED AS BEFORE *)
WITH MEMBER [I] DO . . .
```

Nested WITH statements may be abbreviated as

```
    WITH R1, R2, . . ., Rn DO <statement>
```

which is equivalent to

```
    WITH R1 DO
        WITH R2 DO
        . . .
            WITH Rn DO <statement>
```

The normal scope rules (described in detail in paragraph 8.7) apply, so that a WITH variable may be redefined in nested WITH statements. The innermost definition is the one which applies.

Example:

```
    (* ASSUME STARTDATE, ENDATE : DATE *)
    WITH STARTDATE, ENDATE DO
        MONTH := MAY
```

assigns MAY to ENDATE.MONTH.

Another form of the WITH statement allows synonyms to be defined for the record variables, as illustrated by the following example:

```
    (* ASSUME VAR A : ARRAY[1. . 5] OF DATE, I = 1, J = 2, K = 3*)

        WITH X = A[1], C = A[J], A[K] DO
            BEGIN
                C := X;                (* COPY RECORD A[1] TO A[2] *)
                I := 2;
                K := 5;
                X.YEAR := 1975;        (* ASSIGN 1975 TO A[1]. YEAR *)
                MONTH := JUN           (* ASSIGN JUN TO A[3]. MONTH *)
            END
```

Record variables are bound prior to execution of the qualified statement, so that in the above example, X always denotes record A[1] within the WITH statement, since I had the value 1 when the WITH statement was executed. Assigning 2 to I within the WITH statement does not affect the denotation of X.

The identifiers X and C are implicitly declared and their scope is the WITH statement in which they appear. This means that any identifier X or C which exists outside the WITH statement is not accessible inside the WITH statement. An exception to this is identifiers which denote fields of record types. For example, if the record A[I] above had a field denoted by X, then within the WITH statement X. X would denote this field.

Besides being a convenient shorthand notation, WITH statements allow the compiler to do a certain amount of optimization when several components of a record are accessed.

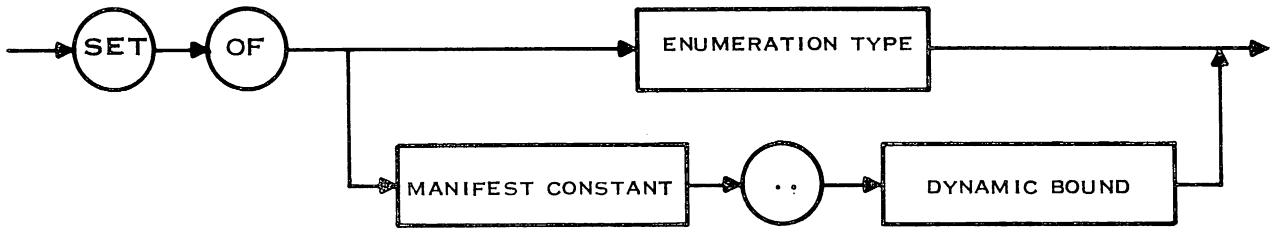
## 6.4 SET TYPE

A SET type is used to define variables whose values are sets. A SET type specifies a base type, and a value of the SET type is then any subset of values from the base type. The syntax for the SET type is:

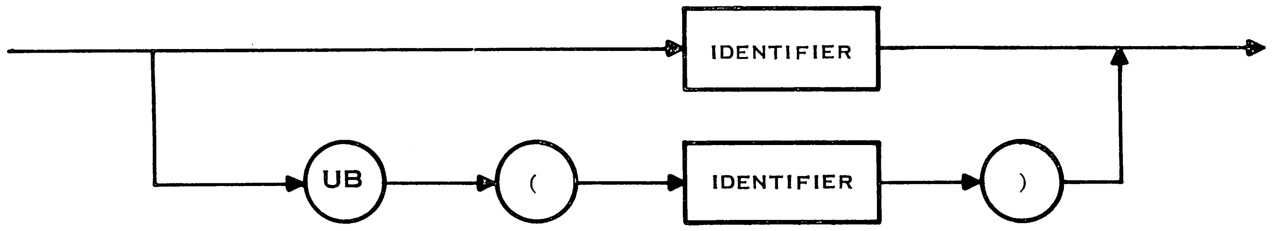
```
<set type> ::= SET OF <base type>

<base type> ::= <static base type> | <dynamic base type>

< static base type> ::= <enumeration type> | <type identifier>

<dynamic base type> ::= <manifest constant>. .<dynamic bound>

<dynamic bound> ::= <entire variable>
          | UB(<dynamic set variable>)

<entire variable> ::= <identifier>
```

The syntax diagrams are as follows:

Set type:



Dynamic Bound:



The base type is any enumeration type. Each SET type includes the set of no elements, called the empty set. The lower bound X of the base type must have an ordinal greater than or equal to 0, and the largest element Y must have an ordinal less than 1023. For sets of integers, only values from 0 to 1023 are allowed.

Examples:

```
TYPE BYTE = SET OF 0. . 7;
    CHARSET = SET OF CHAR;
    COLOR = SET OF PRIMARY;
    ERRORSET = SET OF (OVERFLOW, BUSY, EOFILE)
```

Set values are represented by set constructors, which are a list of set elements (i.e., of expressions of some base type) separated by commas and enclosed by the set brackets [ and ]. The empty set can be denoted by [ ]. Subranges may be used in constructing set values. For example, [M. . N] denotes the set of values from M to N, and if N > M, this is the empty set.

The standard function accepting an argument of type SET is UB(S), which gives the largest value in the base type of the set S. The type of the result is the same as the base type of S.

The following operators apply between operands that are compatible sets (or set and set member in the case of IN):

| | | |
|---|---|---|
| + | set union (inclusive OR) | set of elements present in either A or B |
| - | set difference | set of elements in A and not in B |
| * | set intersection (AND) | set of elements in both A and B |
| <= | set inclusion (contained in) | all elements of A are present in B |
| >= | set inclusion (contains) | all elements of B are present in A |
| < | proper set inclusion | all elements of A are present in B but not all elements of B are present in A |
| > | proper set inclusion | all elements of B are present in A but not all elements of A are present in B |
| = | set equality | A and B contain the same elements |
| <> | set inequality | A and B contain different elements |
| IN | set membership | |
| := | assignment | |

The relationships shown in the right column apply when A is the operand that precedes the operator and B is the operand that follows the operator.

A set is represented internally by a bit string or characteristic vector in which each element of the base type is associated with one bit. Elements that are in the set correspond to ones in the bit string.

Examples:

```
[1, 3, 2] + [2, 6, 3] = [1, 2, 3, 6]
[1, 3, 2] * [2, 6, 3] = [2, 3]
[1, 3, 2] - [2, 6, 3] = [1]
[1, 2, 3] <= [1, 2, 3] — TRUE
[1, 2, 3] < [1, 2, 3] — FALSE
[1, 2]    < [1, 2, 3] — TRUE
TYPE DAYS = (MON, TUE, WED, THUR, FRI, SAT, SUN);
VAR DAY : DAYS;
    WEEK, WORK, WEEKEND : SET OF DAYS;

WORK := [ MON. .FRI ];
WEEKEND := [ SAT, SUN ];
WEEK : = WORK + [ SAT, SUN ];
WEEKEND := WEEK - WORK;
IF WORK * [ TUE ] = [] THEN. . . .
IF DAY IN WEEKEND THEN . . . .
IF WORK <= WEEK THEN . . . .
```

Given appropriate declarations for ERR, CURRENTERRS, FATALERRS:

```
    IF ERR IN FATALERRS THEN WRITELN(' FATAL ERROR ');
    CURRENTERRS := CURRENTERRS + [ERR];
    IF CURRENTERRS <= FATALERRS THEN
        WRITELN(' ALL ACTIVE ERROR CONDITIONS ARE FATAL');
    IF CURRENTERRS >= FATALERRS THEN
        WRITELN(' EVERY FATAL ERROR CONDITION IS ACTIVE')
```

A set may be defined by a static base type, in which case the number of elements is determined at compile times, or the set may be defined by a dynamic base type, in which case the number of elements is determined upon entry to the block containing the set type declaration. Just as for dynamic arrays, the lower bound of a dynamic set is fixed. It must be a manifest constant, that is, an enumeration constant or an integer constant expression which can be evaluated at compile time. The upper bound may be a variable or it may be the UB function applied to an argument which is a dynamic set variable.

Examples:

```
TYPE SETA = SET OF 0. . N;
     SETB = SET OF 128. . UB(SETA)
```

## 6.5 FILE TYPE

A FILE type specifies a structure consisting of a sequence of components which are all of the same type, and may be either sequential or random. In addition, a textfile is a special kind of sequential file of type CHAR which is divided into lines by end of line markers. (A file of type FILE OF CHAR is not substructured into lines, and hence is not equivalent to a textfile.) The number of components, called the length of the file, is not fixed and may grow to any size, limited only by the storage medium with which the file is associated.

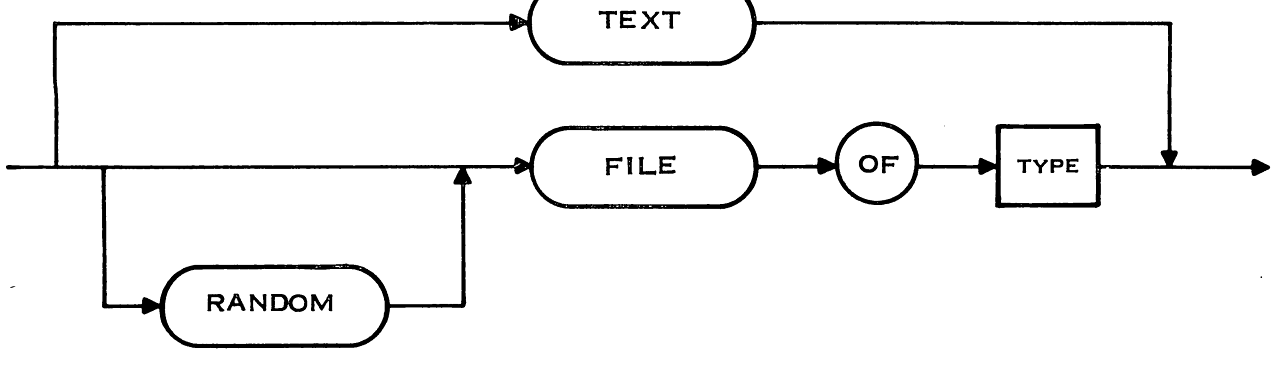
On the Model 990 Computer, files are written to the disk. The I/O procedures and functions create files automatically using default values when they do not already exist. Alternatively, the user may create files using DX10 File Management commands prior to executing a TIP program. When parameters other than the defaults are required, the user must create the files.

The syntax of FILE type is:

<file type> ::= [RANDOM] FILE OF <type> | TEXT

The syntax diagram is as follows:

File type:



The component type may not be a pointer or file type or contain pointers or files. Any other type is legal as a component of a file.

The characteristic feature of a sequential file is that its components are accessible only by progressing sequentially through the file.

The prefix RANDOM specifies a random file in which components are accessible by their component number, which is defined by the natural ordering of the sequence of components. The first component is number zero.

Examples:

```
TYPE  INFILE = FILE OF INTEGER;

      OUTFIL = RANDOM FILE OF LONGINT;

      MSG = TEXT;
```

The predefined textfiles INPUT and OUTPUT represent the standard I/O media of a computer installation (such as the terminal keyboard and line printer).

Files are accessed by means of READ procedure statements, and are written to by means of WRITE procedure statements. In addition, READLN and WRITELN statements apply to textfiles. Before writing to a file, it is necessary to execute a REWRITE statement which erases any previous components of the file and opens it for writing. (A REWRITE is done automatically on the standard textfile OUTPUT.)

Before values may be read from a file, it is necessary to execute a RESET statement which positions to the beginning of the file and opens it for reading. READ returns the next value from the file. For a sequential file, reading may proceed until the last component is read. Then the sequential file is in the end-of-file state which is indicated by the standard Boolean function EOF returning a value TRUE when applied to the file identifier. For RANDOM files, EOF is true when a nonexistent component is read.

Sequential files may be opened for reading or writing but not both simultaneously. A RANDOM file is simultaneously opened for reading or writing by either a REWRITE or EXTEND.

I/O errors may cause program termination (the default) or be handled by the program; which action to take is set by the routine IOTERM. The status of the last I/O operation on a specified file may be determined by the function STATUS.

A file may be associated at execution time with a particular operating system file or device by the procedures SETNAME and SETMEMBER. These routines may only be called when a file is in an inactive (closed) state. A file is placed in the closed state by the routine CLOSE. By default, a file is placed in the closed state when control returns to another block from the block in which the file is declared. Once a file is closed, it may be reopened by the procedure EXTEND, RESET, or REWRITE.

The following procedures and functions may be applied to any file F:

| | |
|---|---|
| EXTEND(F) | For a sequential file or a textfile, open F for output and position it to write the first component. The first component is written as the successor of the last component of the last logical file of the file. For a random file, open F for both input and output. |

| | |
|---|---|
| RESET(F) | Open the file F for input and position to read its first component. For a sequential file, if the file is not empty, EOF(F) becomes FALSE; otherwise it becomes TRUE. |
| REWRITE(F) | For a sequential file, make the file empty and open it for output. EOF(F) becomes true. For a random file, make the file empty and open it for both input and output. |
| CLOSE(F) | Place the file F in a closed state; if F is a sequential file that is open for output, write an end-of-file before closing. |
| SETNAME(F,NAME) | Associate file F with the external file specified by NAME which is of type PACKED ARRAY[1. . 8] OF CHAR. NAME may not be the textfile OUTPUT. |
| SETMEMBER (F,LIBNAME,MEMBER) | Associate the file F with the MEMBER of library LIBNAME, both of which are of type PACKED ARRAY[1. . 8] of CHAR. LIBNAME is a library synonym and may not be the textfile OUTPUT. |
| SETLUNO (F,LUNO) | Associate file F with logical unit number LUNO. |
| STATUS(F) | Returns a value of type INTEGER that indicates the status of the last I/O operation on file F. If the operation was successful, 0 is returned; otherwise the result is an integer value that indicates the reason for the operation's failure. These values are listed in paragraph 14.3. |
| IOTERM(F,OVAL,NVAL) | Save the old value of the I/O error flag associated with the file F in OVAL and set the I/O error flag to the new value, NVAL. OVAL and NVAL are of type BOOLEAN, and OVAL must be a variable. If the I/O error flag is TRUE, the occurrence of an I/O error will cause program termination; otherwise control returns as normal and the function STATUS must be used to determine the kind of error which has been encountered. |
| EOF(F) | For a sequential or textfile F, the result is TRUE if the file is not open for input or is in the end-of-file or end-of-medium state. For a random file F, the result is TRUE if the last read attempted to access a nonexistent record. Otherwise, the result is FALSE. |

**6.5.1 SEQUENTIAL FILES.** In the following, let F be a sequential file, and V denote a variable compatible with the component type of F. The file F may contain components of any type except FILE, POINTER, or ARRAY and RECORD whose component types contain a FILE or POINTER type.

| | |
|---|---|
| READ(F,V) | Assign the next component of the file F to the variable V. (The variables may be components of a PACKED array or record structure with an effect the same as that of an assignment statement.) If F is positioned at the end-of-file mark, nothing is read, the positioning of F is not altered, and an error exception occurs. (To skip past an end-of-file mark, SKIPFILES must be called.) |

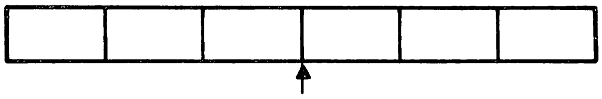
| | |
|---|---|
| WRITE(F,E) | Write the expression E as the next component of the file F. |

**NOTE**

When reading or writing, it is possible to list more than one parameter in the READ or WRITE statement. The forms for reading or writing multiple values in one statement (along with other abbreviations) are explained in paragraph 6.5.6. These forms are easily understood if the basic read and write operations with one parameter are studied first.

| | |
|---|---|
| WRITEEOF(F) | Write an end-of-file mark on the file F which is open for writing. |
| SKIPFILES(F,NFILE) | Skip the number, NFILE, of file marks on the file F which is open for input. If NFILE, of type INTEGER, is negative, the skip is in the "backward" direction; if NFILE is zero, the file is positioned to the beginning of the current logical file; if NFILE is positive, the skip is in the forward direction. The file is positioned at the start of a logical file. An attempt to position to a nonexistent file will cause an error. If EOF is TRUE following a skip, then end-of-medium has been reached. |

A file position is associated with each sequential file which is RESET and divides the file into a part which has already been read and a part which remains to be read. The position is indicated by means of a diagram in which each rectangle represents a component value:



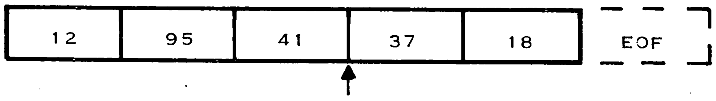
The symbol ↑ is used to mark the current position, in the file. The end of the file is indicated by the symbol eof.

Examples:

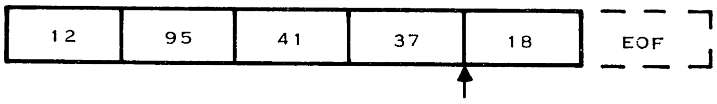
Given the declarations

```
VAR I : INTEGER;
    F : FILE OF INTEGER;
```

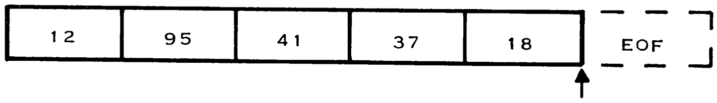
The file F may be at the position:



READ(F, I) results in



At this point, I has the value 37 and EOF(F) is FALSE. Another READ(F, I) yields:



Now I has the value 18 and EOF(F) is TRUE. An attempt to READ F at this point will result in an error.

Example:

```
CONST N = 100;
TYPE REC = RECORD
              NAME : PACKED ARRAY[1. . 10] OF CHAR;
              SSAN : INTEGER
           END;
VAR R : REC;
    LIST : FILE OF REC;

BEGIN
    REWRITE(LIST);
    RESET(INPUT);
    FOR I := 1 TO N DO
        BEGIN
            . . .    (* BUILD RECORD R FROM INPUT DATA *)
            WRITE(LIST, R)
        END;
    RESET(LIST);     (* REWIND LIST *)
    WHILE NOT EOF(LIST) DO
        BEGIN
            READ(LIST, R)
            . . .
        END
END.
```

**6.5.2 TEXTFILES.** Input and output on many devices, including card punches and readers, line printers, CRT terminals, etc., is in the form of characters. The physical properties of these devices naturally divide files of characters into lines. A file of characters which is divided logically into lines by end-of-line markers is called a textfile.

In the Model 990 computer there is no explicit end-of-line character. The last nonblank character (within the logical record length specified when the file was created) is the last character by implication, and the blank following that character is the end-of-line character.

Values are written to a textfile a line at a time. It may be helpful to imagine that a line buffer temporarily stores the character representations of values specified by WRITE statements. A WRITELN statement causes the current line buffer to be added to the textfile and the line buffer to be cleared. On input, a READLN obtains the next line from the file and moves it to the line buffer. The line buffer is intended to help conceptualize I/O on textfiles and does not necessarily reflect the implementation.

An implementation may delete some of the trailing blanks on a line, but never all if the line is blank. When the last character of a line is read the standard function EOLN yields the value TRUE. Reading the next character (i.e., the end-of-line marker) causes EOLN to yield the value FALSE and a blank to be read.

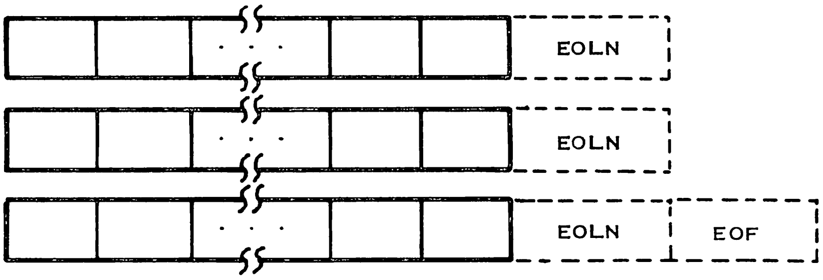
Let F be a textfile and X be of type CHAR, INTEGER, LONGINT, BOOLEAN, REAL, FIXED, DECIMAL, or string. The default field widths for printing values of each of these types is given in an appendix. The procedures for unformatted reading and writing of the textfile F are as follows:

| | |
|---|---|
| WRITE(F,X) | When F is a textfile, WRITE does not add the value of X directly to F. Instead, the value is written to a line buffer. Multiple WRITE statements may add values to the line buffer, and the line buffer is not written to F until a WRITELN is executed. |
| READ(F,X) | Returns the next value in the textfile F. If EOLN(F) is TRUE and X is of type CHAR, a blank is read, which need not correspond to an actual character from the file but rather represents an "end-of-line" value. If a READ is performed and EOLN is TRUE, a READLN is performed to move the next line into the line buffer, and EOLN(F) becomes FALSE. (The variable X may be a component of a packed structure with the read having the same effect as that of an assignment statement.) |
| RESET(F) | A textfile must be reset just as any other file, before it may be read. RESET opens the file, positions to the beginning of the file, and executes a READLN. RESET may not be applied to the standard file OUTPUT. |
| REWRITE(F) | A REWRITE must be executed on textfiles before a WRITE may be performed. The only exception is that by default a REWRITE is performed on the file OUTPUT. REWRITE may not be applied to the standard file INPUT. |
| WRITELN(F) | Output the current contents of the line buffer, followed by an end-of-line, to the textfile F. Then clear the line buffer. |
| READLN(F) | Move the next line of the textfile F into the line buffer. If there are no more lines in F, EOF(F) becomes TRUE. If EOF(F) is already TRUE, an error occurs. |
| EOLN(F) | TRUE when the last character on the current line of textfile F has been read. |
| COLUMN(F) | Provides the column index (based at 1) at which the next character on a textfile will be read or written. |
| PAGE(F) | Causes a skip to the top of a new page when the textfile F is printed. |

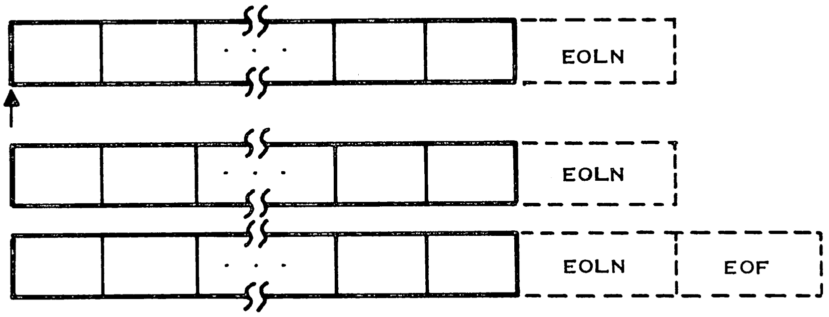
Example:

```
    VAR CH : CHAR;
```

Suppose the INPUT file consists of the three lines

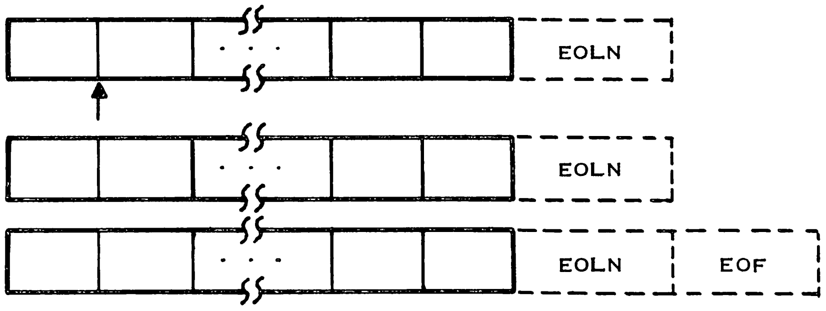


Then RESET (INPUT) yields

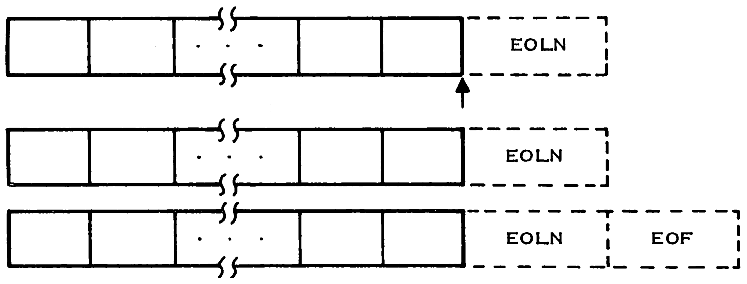


At this point, EOLN(INPUT) and EOF(OUTPUT) are both FALSE.

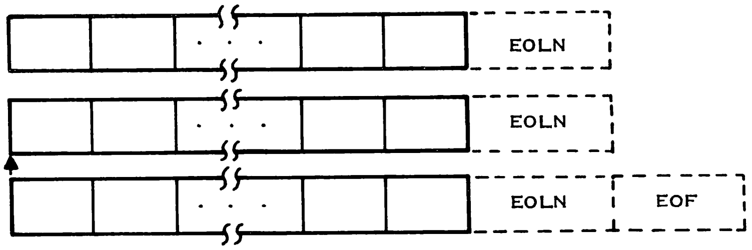
READ(INPUT, CH) yields



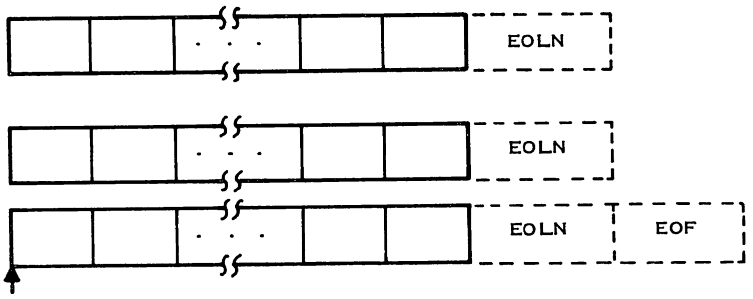
The first character has been assigned to CH. Successive READ statements eventually result in the file position being at the end of the first line:



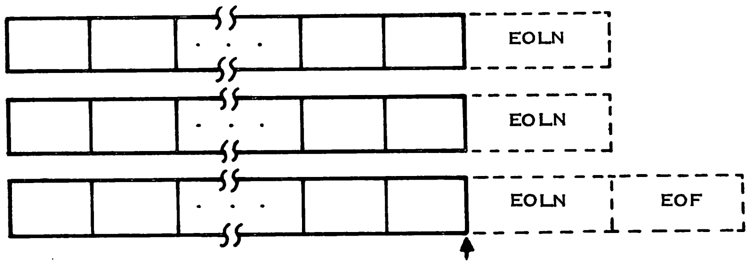
Now EOLN(INPUT) is TRUE and EOF(INPUT) is still FALSE. READ(INPUT, CH) results in CH = ' ', EOLN(INPUT) = FALSE, EOF(INPUT) = FALSE and the file position is:



From this position, if a READLN(INPUT) is executed, the position would be at the beginning of the next line.

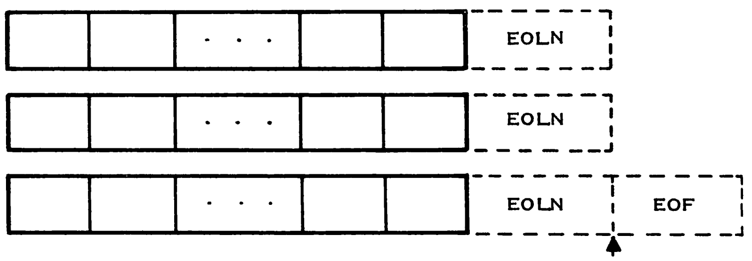


Successive READs will yield:



at which point EOLN(INPUT) is TRUE and EOF(INPUT) is FALSE.

Another READ(INPUT, CH) results in



and now CH = ' ', EOLN(INPUT) = FALSE, and EOF(INPUT) = TRUE. Note that when the end of a textfile is reached, first EOLN is TRUE, and then the next READ makes EOLN FALSE and EOF TRUE. If a textfile originates from a card deck, each end-of-line corresponds to the end of a card, and if it originates from the keyboard of a remote terminal, end of line occurs when a line of text is transmitted, for example, when the RETURN key is depressed. Trailing blanks on a line may be suppressed, so the last character before the end of line is always nonblank.

**NOTE**

> TI PASCAL does not assume that column one of a file is for printer carriage control, as is done in some languages such as Fortran.

**6.5.3 FORMATTED I/O WITH TEXTFILES.** It is frequently convenient to be able to read or write data types other than characters to or from a textfile. For example, numerical values are frequently entered from a device such as a card reader which must be associated with a textfile. The types which may be read from or written to a textfile are CHAR, INTEGER, LONGINT, BOOLEAN, REAL, DECIMAL, FIXED, or string. For each of the types other than CHAR, an implicit data conversion to or from CHAR is performed. In a READ procedure statement of the form

```
READ(F,Q)
```

if the file F is a textfile then the read-parameter Q may have the following forms:

| | |
|---|---|
| V | (unformatted) |
| V:W | (formatted) |

V is a variable to be assigned the value read and must be one of the following allowable types : CHAR, INTEGER, LONGINT, BOOLEAN, REAL, FIXED, DECIMAL, or string. The value to be read may not be split across two logical records.

W is the field width, which must be an integer expression greater than zero. If W is less than zero, the effect is the same as for an unformatted read. V will be read from the next W columns starting with the current file component. The next file component to be read starts with the character immediately following the field. The value to be read may occur anywhere within the specified field. In a formatted read, if EOLN is initially TRUE or the end-of-line mark is reached before W columns have been read, the value accumulated thus far is read. A formatted read never skips past the end-of-line mark. The progression past an end-of-line mark requires an explicit READLN or an unformatted read (which can read an end-of-line mark as a blank).

- If V is of type INTEGER or LONGINT, then the value to be read may be a hexadecimal number. Then either the hexadecimal number is prefixed by a "#" character in the textfile, or the read-parameters have the form:

  ```
  V HEX
  V: W HEX
  ```

- If V is a variable of type CHAR, then V is assigned the next component (a character) if no field width is specified. If a field width is specified, the first nonblank character is read and the remaining characters are ignored. If the entire field is blank spaces, the character read is a blank.

- If V is a variable of type INTEGER (or subrange thereof), LONGINT, or REAL, then READ(H,V) implies reading from H a sequence of characters which form an integer or real constant, respectively, according to the syntax of the language and the assignment of that constant to V. Preceding blanks and end-of-line markers are skipped. In a formatted read, if the field is blank, the value read is zero.

- If V is a variable of type BOOLEAN, then the character T or F is read, or the standard identifier TRUE or FALSE is read.

- If V is a variable of type FIXED or DECIMAL, then READ(H,V) implies reading from H a sequence of characters which form a fixed-point or decimal constant (respectively) according to the syntax of the language except that the sequence need not end in an F or D. If the fixed-point constant is in binary form, then the sequence must end with a B. In a formatted read, if the field is blank, the value read is zero.

- If V is a variable of type string with length L, then the next L characters are read. In a formatted read, if W>L then the rightmost L characters are read; if W<L then the string is padded on the right with blanks.

In a WRITE procedure statement of the form

WRITE(F,P)

where F is a textfile, the write-parameter P may have the following forms:

| | |
|---|---|
| E | (unformatted) |
| E:M | (formatted) |
| E:M:N | (formatted fixed point) |

E is the expression that represents the value to be written, and is of any of the types that may be read from a textfile, that is, CHAR, INTEGER, LONGINT, BOOLEAN, REAL, FIXED, DECIMAL, or string. The value to be written is never split across two logical output records. If the value's length is greater than the output logical record length, an error occurs. The default values for an unformatted write are:

| **Type** | **Field Width** |
|---|---|
| INTEGER | 10 |
| LONGINT | 15 |
| REAL(X) | 15, for 0<X<8 |
| | 25, for 8<=X<N |
| FIXED | 15 |
| DECIMAL | 20 |
| BOOLEAN | 10 |
| CHAR | 1 |
| Hexadecimal | 10 |
| String | Length of string |

M is an integer expression which is the minimum field width. If M is omitted, an implementation defined default value is assumed according to the type of E. In general, the value E is written with M characters. If the value E requires less than M characters for its representation, then an adequate number of preceding blanks is written such that exactly M characters are written. If M is less than the number of characters required to represent E, then necessary additional space is allocated.

- The specification of N is optional. If N is specified, E must be of type REAL, FIXED, or DECIMAL, and the value of E is written in a fixed point representation with N digits after the decimal point. If N is omitted, and E is of type REAL, its value is written in a floating point representation which consists of a coefficient and scale factor. For real, fixed point, and decimal write-parameters of the form E:M, no more precision is ever printed than the value contains.

- If V is of type INTEGER or LONGINT then the value may be written as a string of hexadecimal digits (not preceded by the character "#"). The write parameters then have the form:

  E HEX
  E:M HEX

  If the number of nonzero hexadecimal digits in E is less than or equal to M, E is written with (possible) leading zeros. The maximum number of hexadecimal digits (including leading zeros) that can be written is defined by the implementation of INTEGER and LONGINT.

- If the value of E is of type FIXED, then it may be written as a binary number. In this case, the write-parameters are of the form

  E BIN

  E:M BIN

  E:M:N BIN

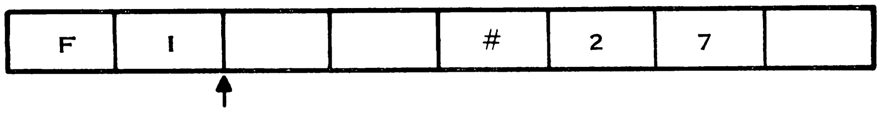
- If the value of E is of type BOOLEAN, then the standard identifier TRUE or FALSE is written, preceded by an appropriate number of blanks as specified by M. If M<5, the character T or F is written instead.

- If the value of E is a packed array of characters, then the string E is written.

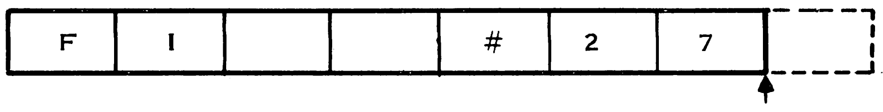
Example:

Given the declarations

```
VAR CH : CHAR;
     X : INTEGER
```



Executing READ(F,X) results in



**6.5.4 ENCODE AND DECODE.** The procedures ENCODE and DECODE are processed just like the textfile procedures WRITE and READ respectively, except a memory array is used instead of a file.

| | |
|---|---|
| ENCODE(S,N,STAT,P) | P is a write-parameter of the form given in paragraph 6.5.3. The character representations of E's value is placed into S, of type string, starting at the Nth component of S. N may be a constant or variable of type INTEGER. If N is a variable, it is automatically incremented by the number of characters transferred. The status of the operation is returned in the variable STAT which is of type INTEGER. The value returned is the same as that for the function STATUS. If (N-1) plus the number of characters to be transferred is not a valid index into the string, an error is indicated in the STAT variable, nothing is transferred and N is not updated. |
| DECODE(S,N,STAT,Q) | Let Q be a read-parameter of the form given in paragraph 6.5.3. The characters starting at the Nth component in the string S are converted to the internal representation of V and this value is assigned to V. V may be a component of a packed structure. N may be a constant or variable of type INTEGER. If N is a variable, it is automatically incremented by the number of characters transferred. The status of the operation is returned in the variable STAT which is of type INTEGER. If Q is formatted and (N-1) plus the number of characters to be transferred is not a valid index into the string, an error is indicated in the STAT variable, nothing is transferred, and N is not updated. If Q is unformatted, the last character of the string terminates the value to be transferred. |

**6.5.5 RANDOM FILES.** The READ and WRITE procedures for random files are the same as for sequential files except they include an argument that specifies the logical position of the file element to be accessed.

The EOF function can be used to determine if a nonexistent record has been referenced. Note that if record N has been written, then records 0. . .N exist, even though values may not have been written to some of them. RESET opens a random file for input, either EXTEND or REWRITE opens a random file for both input and output, and REWRITE causes the file to be erased.

In the following, let F be a random file, and let V denote a variable compatible with the component type of F and of any type except FILE, POINTER, or ARRAY and RECORD whose component types contain a FILE or POINTER type.

| | |
|---|---|
| READ(F,RECNUM,V) | Assign the component with logical position RECNUM to the variable V. RECNUM must be an expression of type INTEGER whose value is greater than or equal to zero. It is not automatically incremented after the READ operation. (The variable V may be a component of a PACKED array or record structure with the read having the same effect as that of an assignment statement.) An error occurs if the component does not exist or the value of RECNUM is less than zero. |

| | |
|---|---|
| WRITE(F,RECNUM,V) | Write the variable V as the component with logical position RECNUM. RECNUM must be an expression of type INTEGER whose value is greater than or equal to zero, and is not automatically incremented after the WRITE operation. An error occurs if the value of RECNUM is less than zero. |

**6.5.6 ALTERNATE FORMS.** Certain nonstandard or abbreviated forms of the procedure statements for file manipulation are also allowed. These forms are shown in the lefthand column and the equivalent standard form is shown on the right.

```
EOF                          EOF(INPUT)
EOLN                         EOLN(INPUT)

READ(F,V1,V2, . . ., Vn)     BEGIN
                               READ(F, V1);
                               READ(F, V2);
                               . . .
                               READ(F, Vn)
                             END

READ(V1, V2, . . ., Vn)      READ(INPUT, V1, . . ., Vn)

READLN(F, V1, V2, . . ., Vn) BEGIN
                               READ(F, V1, . . ., Vn);
                               READLN(F)
                             END

READLN(V1, V2, . . ., Vn)    READLN(INPUT, V1, . . ., Vn)

READLN                       READLN(INPUT)

WRITE(F, P1, P2, . . ., Pn)  BEGIN
                               WRITE(F, P1);
                               WRITE(F, P2);
                               . . .
                               WRITE(F, Pn)
                             END

WRITE(P1, P2, . . ., Pn)     WRITE(OUTPUT, P1, . . ., Pn)

WRITELN(F, P1, . . ., Pn)    BEGIN
                               WRITE(F, P1, . . ., Pn);
                               WRITELN(F)
                             END

WRITELN(P1, . . ., Pn)       WRITELN(OUTPUT, P1, . . ., Pn)

WRITELN                      WRITELN(OUTPUT)

PAGE                         PAGE(OUTPUT)
```

If N is a constant:

```
ENCODE(S,N,STAT,P1, . . ., PN)        BEGIN
                                        ENCODE(S,N,STAT,P1);
                                        ENCODE(S,N+L(E1), STAT, P2);
                                        . . .
                                  ENCODE(S,N+L(E1)+L(E2)+. . .+L(EN-1),STAT,PN)
                                      END
```

where L(Ei) is the length of the string representation of Ei. If N is a variable, replace the second parameter in all of the above procedure calls by N.

If N is a constant:

```
DECODE(S,N,STAT,Q1, . . ., QN)        BEGIN
                                        DECODE(S,N,STAT,Q1);
                                        DECODE(S,N+L(V1),STAT,Q2);
                                        . . .
                                  DECODE(S,N+L(V1)+L(V2)+. . .+L(VN-1),STAT,QN)
                                      END
```

where L(Vi) is the number of characters converted for Vi. If N is a variable, replace the second parameter in all of the above procedure calls by N.

If F is a random file:

```
READ(F,RECNUM,V1,V2, . . . VN)        BEGIN
                                        READ(F,RECNUM,V1);
                                        READ(F,RECNUM+1,V2);
                                        . . .
                                        READ(F,RECNUM+N-1,VN)
                                      END

WRITE(F,RECNUM,P1,P2, . . . PN)       BEGIN
                                        WRITE(F,RECNUM,P1);
                                        WRITE(F,RECNUM+1,P2);
                                        . . .
                                        WRITE(F,RECNUM+N-1,PN)
                                      END
```

After each call, RECNUM has not been incremented.

## 6.6 POINTER TYPE

Variables may be referenced indirectly by means of a pointer, which may be thought of as the address of the variable pointed to by the pointer variable. A pointer type consists of an unbounded set of values pointing to elements of a given type. Pointer variables are most often used in conjunction with records to create data structures such as linked lists or trees. The syntax of a pointer type is:

```
<pointer type> ::= @<type identifier>
```

The type of <type identifier> is said to be bound to the pointer type, and may not be a file. A pointer variable can point only to variables of the type to which it is bound. The predefined constant NIL is an element of every pointer type and points to no element at all.

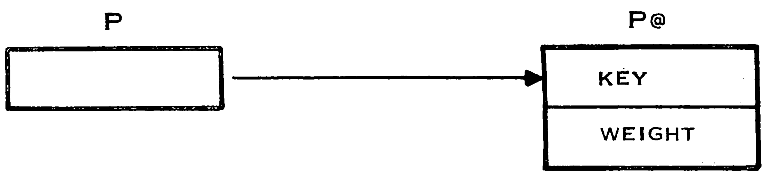
A pointer type is a structured type. The component of a pointer variable is denoted by the pointer variable followed by the symbol '@'. The syntax is:

<referenced variable> ::= <pointer variable>@

If the value of P is NIL, an attempt to reference P@ will use the representation of NIL as if it were a valid pointer. The result of this is an error when the program is executed. A compiler option is available to check at runtime for a reference to a NIL pointer.

Example:

```
TYPE    PTYPE = @REC;
        REC = RECORD
                KEY : INTEGER;
                WEIGHT : REAL
            END;
VAR P : PTYPE;
```



The declaration of PTYPE is an example of a forward type declaration, since it precedes the declaration of REC. Forward type declarations are permitted only with pointer types.

The operators applying to pointer operands with compatible types are:

| | |
|---|---|
| := | assignment |
| = | equal (The result is TRUE if the operands point to the same "address") |
| <> | not equal |

The standard function LOCATION may be used to obtain a result of type INTEGER which is the address of the variable V denoted by its argument. LOCATION may also be used to obtain the entry point of a routine. It may not be used on an argument which is a component of a packed structure or a file variable.

**6.6.1 STATIC AND DYNAMICALLY ALLOCATED VARIABLES.** Declared variables are referenced by the identifier by which they are declared, so pointers to these so-called static variables are of little use. On the other hand, it is possible to create variables without the use of declarations. These dynamically allocated variables are not associated with an identifier, so they must be referenced by means of a pointer. Dynamically allocated variables are created at execution time by the standard procedure NEW, and may be deallocated by the standard procedure DISPOSE.

NEW(P)

Creates a new variable of the same type as the component type of the pointer variable P. The address of this new variable is assigned to P. (If the component type of P is a record type with variants, then enough space is allocated to accommodate the largest variant.)

NEW(P,T1,. . ., Tn)

This form is valid when the component type of P is a record type with variants nested to a depth greater than or equal to n. T1 through Tn are compile time constants which specify the value of the first n tag fields in the order of their declaration. The effect is to allocate storage for a new variable of the record type with tag field values T1, . . ., Tn, and assign the pointer to this new record to the pointer variable denoted by P. The actual values of the tag fields in the new component are not initialized.

When the second form of NEW is used to generate a variant that requires less storage to be allocated than the "largest" variant, errors may arise if an object with larger storage requirements is assigned into that variant. For example, if the record definition is

```
TYPE REC = RECORD
              KEY : INTEGER;
              CASE T : BOOLEAN OF
                  TRUE : (X : ARRAY[1. . 5] OF INTEGER);
                  FALSE : (Y : ARRAY[1. . 10] OF INTEGER)
          END;
     RECPOINT = @REC;
```

If the procedure NEW had been used to create a new component

```
NEW(RECPOINT, TRUE)
```

then RECPOINT would point to a record with enough space allocated for the variant X with array of length 5. In this case, if the tag field T is assigned the value FALSE, it is possible to assign to the variant Y, and the program may then produce incorrect results.

The procedure NEW obtains storage from what is called the heap to allocate space for the variable which it creates. Variables which have been allocated dynamically from this heap may be deallocated at runtime by the standard procedure DISPOSE.

DISPOSE(P)

Makes the space pointed to by P available for reuse. P must point to a dynamic variable, i.e., one allocated by NEW. If the value of P is NIL, an error occurs. After the storage is deallocated, P is set to NIL.

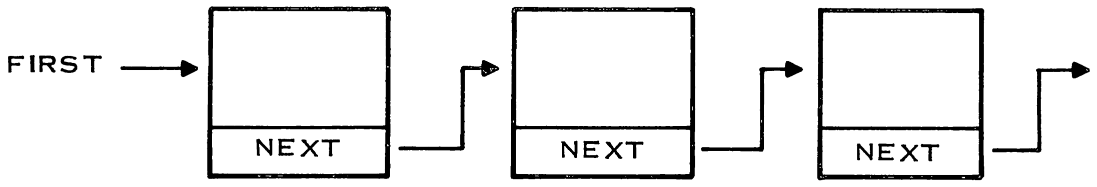
Another form of DISPOSE may be used which corresponds to the second form of NEW.

DISPOSE(P, T1, T2, . . ., Tn)

The same rules apply to T1 . . . Tn as in NEW. These values should agree with the values specified when the component was created by NEW (but don't have to since always exactly as much space is deallocated as was allocated by NEW. In other words, a check is not made to see that the value of T1 . . . Tn agree with the variant actually deallocated.)

Example:

A linked list of records may be created very easily by defining a record which contains one field which is a pointer to the next record. This is illustrated by the following diagram:



The form of the record definition for this linked list is:

```
TYPE PT = @LISTELEMENT;
    LISTELEMENT = RECORD
                        . . .
                        KEY : INTEGER;
                        NEXT : PT
                    END;
VAR FIRST, POINT, PN : PT;
```

FIRST is a pointer to the head of the linked list, and POINT is used to access elements of the list. Note that a "forward declaration" is allowed here.

The list may be created as follows:

```
NEW(POINT);
FIRST := POINT;
WHILE NOT EOF DO
    BEGIN
        (* BUILD COMPONENTS OF THE RECORD *)
        . . .
        NEW(PN);
        POINT@.NEXT := PN;
        POINT := PN;
    END;
POINT@.NEXT := NIL
```

Suppose the list is to be searched for an element with a KEY field value of M. This will be done by letting the variable POINT point in turn to each element of the list until the proper element is found (if it exists), as shown in the following example:

```
POINT := FIRST;
WHILE POINT <> NIL AND POINT@.KEY <> M DO
    POINT := POINT@.NEXT
```

Note that this example would be more complicated if short circuit evaluation were not available.

## 6.7 PACKED DATA TYPES

The symbol PACKED may be prefixed to the structured type definition of arrays, records, or sets. If a data structure is declared to be packed, the compiler utilizes the packing algorithm described here to obtain a representation for the data in which, if possible, several components of the structure are stored in one word. While packing may economize the storage requirements of a data structure, it also may cause a loss in efficiency of access of its components.

The prefix PACKED does not distribute to the components of a type. Note also that a packed type is not compatible with an unpacked but otherwise compatible type. A direct component of a structured type is the component at the first level of decomposition of the structured type. A direct component of a packed structured type may not be passed by reference to a routine (see Section VIII for routines).

**6.7.1 PACKED ARRAYS.** Let T1 . . . Tn be index types and T a type. Then for a static array,

PACKED ARRAY[T1, . . . Tn] OF T

is equivalent to

PACKED ARRAY[T1] OF . . . OF PACKED ARRAY[Tn] OF T

which is not equivalent to

ARRAY[T1] OF PACKED ARRAY[T2] OF . . . OF PACKED ARRAY[Tn] OF T

since it may not occupy the same amount of storage. For a dynamic array, only the last dimension may be packed, i.e.,

PACKED ARRAY[T1. . .Tn] OF T

is equivalent to

ARRAY[T1] OF . . . OF PACKED ARRAY[Tn]

Strings consisting of N>=1 characters are defined via the type

PACKED ARRAY[T1] OF CHAR

where T1 must be of the form "1. . N". If T1 is a static index type, the string's length is fixed at compile time; if T1 is a dynamic index type, the string's length is fixed at runtime. The length of the string can be determined by using the standard array function UB. A string constant is of such a type, with length equal to the number of characters. Routines can readily be defined to extract substrings, do pattern matching, or perform any other desired operation on strings. The basic operators for variables of string type are assignment (:=) and the relational operators, (<, =, >, <=, <>, >=).

**6.7.2 PACKED RECORDS.** A packed record allows the programmer to define the storage allocation of a record type in which the exact position and size of each variable field may be specified subject to the size algorithm (paragraph 6.7.3).

Fields are allocated in the order specified. The size algorithm may not produce the tightest packing if a field's size is an integral number of words plus a fraction of a word. To achieve the tightest packing, it is the programmer's responsibility to handle this special case by splitting the logical field into two or more physical fields and write routines to pack and unpack a value before using and storing it.

**6.7.3 INTERNAL REPRESENTATION OF TYPES.** The size algorithm, given below, specifies in terms of bits and words the internal representation for the value of a type. These specifications are given so that machine dependent records and machine code routines can be sensibly defined, and so that the effect of a type transfer (paragraph 6.9) can be predicted.

If a type occurs in a packed structure, then exactly as much storage as specified by the size algorithm should be allocated to it. The size algorithm allocates either a portion of a word or an integral number of words. That is, if a type requires more than one word, then it always uses an integral number of words and not an integral number of words plus a fraction of a word. Consequently, gaps of unused bits may occur. If the type does not occur in a packed structure, the size becomes a lower bound, the actual size being selected to facilitate efficient access to the type on the underlying machine.

The size associated with each type is defined as follows:

- CHAR: ASCII - 8 bits.
- INTEGER: 16 bits.
- LONGINT: 32 bits
- Boolean: 1 bit
- Scalar: Let N be the ordinal of the largest member of the enumeration, and define NR(N) to be the least value of I such that N < 2**I. Then the scalar type requires NR(N) bits.

  Example:

  TYPE WEEK = (MO, TU, WD, TH, FR, SA, SU)

  Then N = ORD(SU) = 6

  3 is the least value of I such that $6 < 2^I$, so the size of the type WEEK is 3 bits.

- Subrange: Let L and U be the lower and upper bounds of the subrange. Then if L >= 0 the size is the same as for a scalar type which has the ordinal of the largest member of its enumeration equal to U. If L< 0, the size is Max(NR(-L-1), NR(ABS(U))) + 1.

  Examples:

  ```
  TYPE T1 = TU. . FR
  ```

  The size is 3 bits, since NR(FR) = 3.

  ```
  TYPE T2 = -8. . 3
  ```

  The size is Max(NR(7), NR(3)) + 1 = Max(3,2) + 1 = 4.

- Real: The size of a type REAL (n) is 32 bits when $n \leqslant 7$ and 64 bits when $n \geqslant 8$.

- Pointer: The size of a pointer is 16 bits.

- Array: If the array is not packed, each element occupies one or more consecutive words. Let S be the size of an element, that is, the size of the component type. If the array has E elements, then the size of the array is E*S.

  If the array is packed and the minimum size of an element is greater than a word, then the space, S, allocated for each element is the minimum number of words which will contain it. If the array has E elements, the size of the array is E*S.

  If the array is packed and the minimum size, S, of an element is less than a word, as many elements as possible are packed per word (D) with a possible number of bits left unused at the end of the word. The data will occupy a nonintegral number of words W+F, where W is the whole number of words, E/D, and F is the fraction of a word (E MOD D) * S. The remaining bits of the portion of the word are unused. That is, the array occupies an integral number of words; either the exact number of words required for the elements of the array, or the whole number of words required (W) plus the word that contains the fraction of a word (F). An array may occupy a single word or a portion of a word (W = 1, F = 0, or W = 0, F = 1).

  Example:

  ```
  (* assume 16-bit words *)

  A = PACKED ARRAY [1. . 10] OF 0. . 31
  ```

  The subrange type 0. . 31 requires 5 bits. A requires three whole words (with one bit unused in each word) and 5 bits of a fourth word.

  ```
  X = PACKED ARRAY[1. . 5] OF T
  ```

  where T requires 20 bits. Then X requires 10 full words.

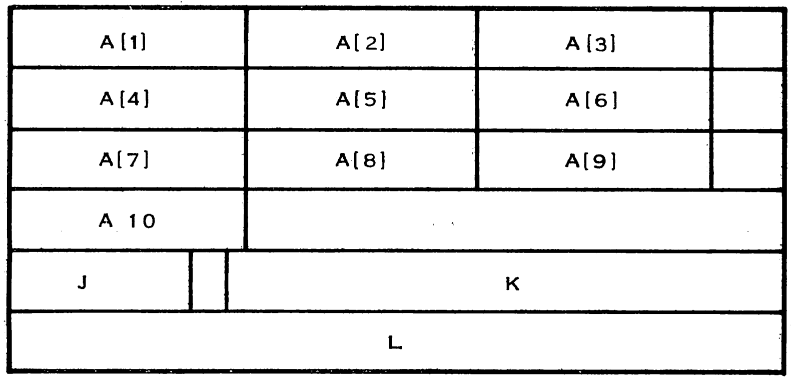
- Record: The size associated with a record type is the number of consecutive words and bits needed to contain the fields in the fixed part plus the largest field list in the variant part. Fields are allocated in the order of declaration.

  If a record is not packed, a field occupies one or more words as required by the size of its associated type. If the record is packed and the preceding field occupies less than a full word, a field is allocated within the remainder of the word allocated to the preceding field provided it fits. If the preceding field occupies more than one word (even though there may be unused bits in the second or subsequent word of the field), or if there are not enough unused bits, the field is left justified at the beginning of the next available word, and the previous field is right justified in the previous word. If the size of the record is greater than a word, then the last field of every variant is right justified.

  Field lists within the variant part are overlaid upon one another.

  Example:

```
    (* ASSUME 16-BIT WORDS *)
    TYPE R = PACKED RECORD
            A : PACKED ARRAY [1. . 10] OF 0. . 31;
                            (* NOTE SIZE 0. . 31 = 5 *)
            J: 0. . 7;
            K : 0. . #FFF;
            L : INTEGER
        END
```



  Note the unused bits: 1 in each of the first three words, 11 in the fourth word, and 1 in the fifth word.

- Set: The size of a set type depends on the size of its base type. If the base type of a set has an upper bound with ordinal N, then a packed set requires at least N+1 bits; otherwise it occupies the least number of bits that is greater than or equal to N+1 which can be efficiently accessed on the machine. The maximum set size is 128 bytes (1024 elements).

- Fixed: A variable of type FIXED occupies one byte per 8 bits of precision.

- Decimal: A variable of type DECIMAL occupies one byte for each two digits. A sign, which counts as a digit, is included.

**6.7.4 THE SIZE FUNCTION.** The standard function SIZE applies to any type. SIZE(T) yields a result of type INTEGER which is the number of bytes required to represent type T.

The following forms of the call to function SIZE are valid:

```
SIZE(T)
SIZE(T,T1,. . .,Tn)
SIZE(V)
SIZE(V,T1,. . .,Tn)
```

The first argument is either a type (T), a type identifier (T), or a variable (V). For all types except REAL(P), DECIMAL (P,Q), and FIXED (P,Q), T is the type. For REAL (P), DECIMAL (P,Q), and FIXED (P,Q), T is the type identifier of an item of any of these types. A variable (V) may be any type. When T is a record type or V is a variable of record type, T1 through Tn are tag fields of variants in the record. The number of tag fields specified must be less than or equal to the number of variants in the record. T1 through Tn must represent a complete initial sequence of tag fields and must be compile time constants.

## 6.8 TYPE COMPATIBILITY

Two types T1 and T2 are distinct if they are explicitly or implicitly declared in different parts of the program. Types T1 and T2 are compatible if T1 may be used in the context of T2 with the exception of VAR parameter transmission (see paragraph 8.5). If the type definition involves a dynamic type, e.g., arrays or sets with expression or "?" bounds, runtime checks may have to be made for type compatibility. The rules that apply are the same as for nondynamic types which can be checked for compatibility at compile time. It should be noted that a "?" bound matches any subrange's upper bound.

One type T1 is compatible with type T2 if:

- Both types are subranges of a single enumeration type within the scope in which the compatibility check occurs, or
- Both are string types of the same length, or
- Both are set types whose base types B1 and B2 are semantically identical (i.e., K is an element of B1 if and only if K is an element of B2). The empty set is compatible with any set. Set expressions will be typed if possible from the context established by set variables; otherwise they are typed as either the full scalar type, type CHAR, or type 0. . 1023, or
- T1 is of type INTEGER or LONGINT, or a subrange thereof, and T2 is of type REAL, FIXED, or DECIMAL, or
- T1 is of type INTEGER and T2 is of type LONGINT, or
- Both are file types of compatible element types, or
- Both are array types of compatible index types with identical bounds and of semantically identical component types with the following exception: the component types may both be subranges of the type INTEGER, both may be subranges of the type LONGINT, or both may be subranges of any other enumerations type, or

- Both are record types with corresponding fields of semantically identical component types with the following exception: the component types may both be subranges of the type INTEGER, both may be subranges of the type LONGINT, or both may be subranges of any other enumeration type, or

- Both are pointer types which either point to nondistinct structured types or to compatible nonstructured types, or

- Both are of type REAL, or both are of type FIXED, or both are of type DECIMAL, but with different precision.

Structured types have the further restriction that both are either packed or unpacked. Even if two structured types are not compatible, their components may be compatible. With reference to the first item of the preceding list, if either subrange occurs in a packed structure, then they must have the same bounds.

The only implicit type conversions are:

- INTEGER to LONGINT
- LONGINT to INTEGER
- INTEGER to REAL
- LONGINT to REAL
- REAL of one precision to REAL of another precision
- FIXED of one precision to FIXED of another precision
- DECIMAL of one precision to DECIMAL of another precision

Type conversion, both implicit and explicit, is described in Appendix I.

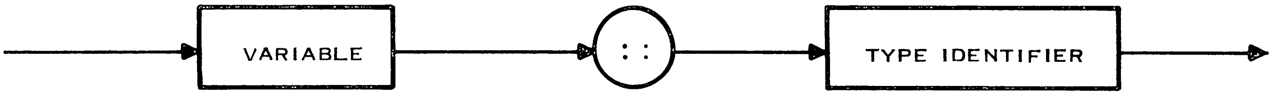
## 6.9 TYPE TRANSFER

Type transfer is a means of temporarily changing the type of an existing variable. The syntax is:

```
<type transferred variable>
            ::= <variable> :: <type identifier>
```

The syntax diagram is as follows:

Type-transferred variable:



A type-transferred variable may be used wherever a variable is allowed. Regardless of its original type, the type-transferred variable is accessed according to the type indicated.

Example:

```
TYPE BYTE = 0. .#FF;
    RECTYPE = PACKED RECORD
                MSBYTE,LSBYTE : BYTE
            END;

VAR V : ARRAY[0. . 9] OF INTEGER;
    R: RECTYPE;
```

Valid type transfers:

```
R.MSBYTE := V[0] ::BYTE;
V[1] ::BYTE := R.LSBYTE;
READ(R::INTEGER)
```

No value conversion is performed; the only effect is to change the apparent type of the variable. The variable must not be declared to be a procedure, function, or constant. In addition, a variable which is a component of a packed structure may only be transferred to a type representable within the boundaries of that component. The type transfer applies only in the variable in which it is stated; other appearances of the variable must use the type transfer format if a different type is required.

Example:

```
TYPE PT = @SCB
VAR FIRST : PT

FIRST::INTEGER := FIRST::INTEGER + 40
```

Example:

The type transfer

```
R.MSBYTE::INTEGER
```

is illegal since the size of the type INTEGER is larger than the 8 bits required to represent the component R.MSBYTE.

The fundamental use of type transfer is to overlay a type template on a data structure so that components of the structure may be treated as if they were of any desired type. It requires a precise understanding of the compiler's representation of the data type on the machine to make use of type transfer, and because of this it should be used with caution and only when necessary.

# SECTION VII

# JUMP STATEMENTS

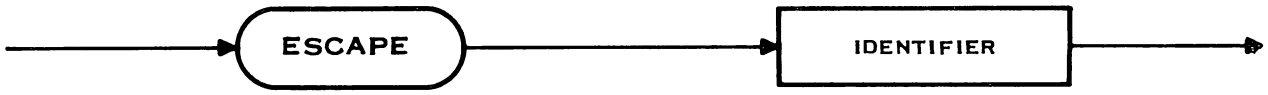
## 7.1 THE ESCAPE STATEMENT

The ESCAPE statement is a structured jump statement. It is used to terminate execution of the current statement, routine or program. The syntax is:

ESCAPE <identifier>

The syntax diagram is as follows:

ESCAPE statement:



The <identifier> must be one of the following:

1. An escape label
2. A routine identifier
3. A program identifier

An escape label is an identifier which is prefixed to a structured statement. It is separated from the statement by a colon. The escape label is implicitly declared by its use within the program. The structured statement is a unit of scope, so the escape label may not be used as a variable, constant, program, or routine identifier within the labeled structured statement. (See paragraph 8.7 for a discussion of scope.)

An ESCAPE statement may appear only within the statement labeled with the escape label or within the scope of the routine named by the routine identifier. When an ESCAPE from a statement is executed, further processing continues at the statement following the structured statement labeled by the escape label. When an ESCAPE from a routine is executed, control returns from the most recently entered activation of the routine. ESCAPE <program identifier> terminates the program.

Example:

```
LOOP : FOR I := 1 TO N DO
    BEGIN
        IF EOF THEN ESCAPE LOOP;
        READ A[I];
        S := S + A[I]
    END
```

A statement prefixed by an escape label may contain any number of ESCAPE statements that reference this label. The escape label and all of its associated ESCAPE statements must appear in the same routine body (it is illegal to escape across routine boundaries). Also, routines cannot escape *from brothers*. That is, if routines C and D are defined at the same level (see paragraph 8.6), it is not possible to have ESCAPE C as a statement in D. Escape labels are implicitly declared, do not have to be unique, and may be reused within a block.

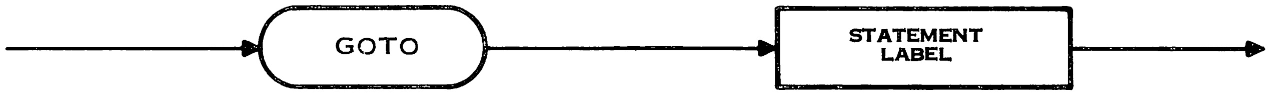
## 7.2 THE GOTO STATEMENT

The GOTO statement transfers execution to the statement having the named label. The syntax is:

GOTO <statement label>

The syntax diagram is as follows:

GOTO statement:



The <statement label> must be an unsigned integer and must be in a LABEL declaration, as discussed in paragraph 8.2. If the label is not declared or does not appear as a statement label in the program, a syntax error occurs.

Example:

```
PROGRAM LAB;
LABEL 100;
VAR X: REAL;
  .
  .
  .
BEGIN
  .
  .
  .
100 : I := I + 1;
          IF A[I] <> X THEN GOTO 100
```

GOTO statements should be used as seldom as possible, since the use of other control structures such as WHILE and CASE can result in clearer code, especially if the program is well designed. In addition, any procedure, function, or program with label declarations will not have its <body> part optimized by the TIP compiler.

It is not legal to jump into or out of a procedure or function. And, it is not legal to jump into a FOR or WITH statement. At most one statement label can mark a given statement. If a statement label and an escape label both are used on a structured statement, the statement label must be first. For example,

```
100 : LOOP : FOR N := 1 TO 64 DO . . .
```

# SECTION VIII

# THE PROGRAM AND ITS ROUTINES

## 8.1 THE TIP PROGRAM

The TIP program consists of the program heading, the declarations, and a compound statement that includes the statements of the program. The syntax of a program is as follows:

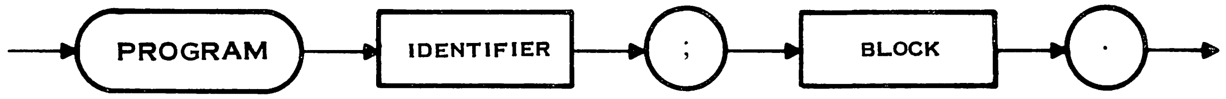
<program> ::= <program heading> <block>.

<program heading> ::= PROGRAM <program identifier>;

<program identifier> ::= <identifier>

The syntax diagram is as follows:

Program:



The declarations and statements of the program are referred to as the block. The syntax of the block is as follows:

<block> ::= <declarations> <compound statement>

<declarations> ::= [<label declaration part>] [<constant declaration part>]

[<type declaration part>] [<variable declaration part>]

[<common variable declaration part>]

[<access declaration part>]

[<procedure and function declaration part>]

The syntax diagrams are shown in figure 8-1. The label, constant, type, variable, common, access, procedure, and function declarations are defined in subsequent paragraphs. A procedure or function declaration either includes or references a block that contains the declarations and compound *statement* for the procedure or function. The syntax of the block is the same whether it constitutes a program or a procedure or function at any level.

## 8.2 DECLARATIONS

The seven types of declarations used in the blocks of programs, functions, and procedures are described in the following paragraphs. The declarations may be omitted, but, when included, must be in the sequence in which they appear in the BNF production for declarations. Each identifier used in the program must be declared (either in the block in which it is used or in a block at a higher level that encloses the block in which it is used) or it must be standard in the language. (Some identifier declarations are implicit: FOR control variables, variables used as abbreviations in WITH statements, and ESCAPE labels. These identifiers do not require explicit declaration.)

**8.2.1 LABEL DECLARATION.** The syntax of the LABEL declaration is as follows:

```
LABEL <statement label> {, <statement label>} ;
```

A label on a statement must be declared in the label declaration part of the program. The label is an unsigned integer, and only one statement in the statement part may be prefixed with a given label. A GOTO statement may then be used to transfer control to the labeled statement.

**8.2.2 CONSTANT DECLARATION.** The syntax of the constant declaration is as follows:

```
CONST <identifier> = <constant expression>
    {; <identifier> = <constant expression>};
```

In the subsequent text of the program, the identifier may be used as a synonym for the constant. The value associated with the constant identifier may not be changed during program execution. Constant expressions may involve only numbers, previously defined constant identifiers, operators, and strings.

Examples:

```
CONST MAX = 128;
    HEAD = 'INITIAL VALUE';
    VAL = MAX*2 - 1;
```

**8.2.3 TYPE DECLARATION.** Identifiers may be used to denote a type by means of type declarations with the syntax:

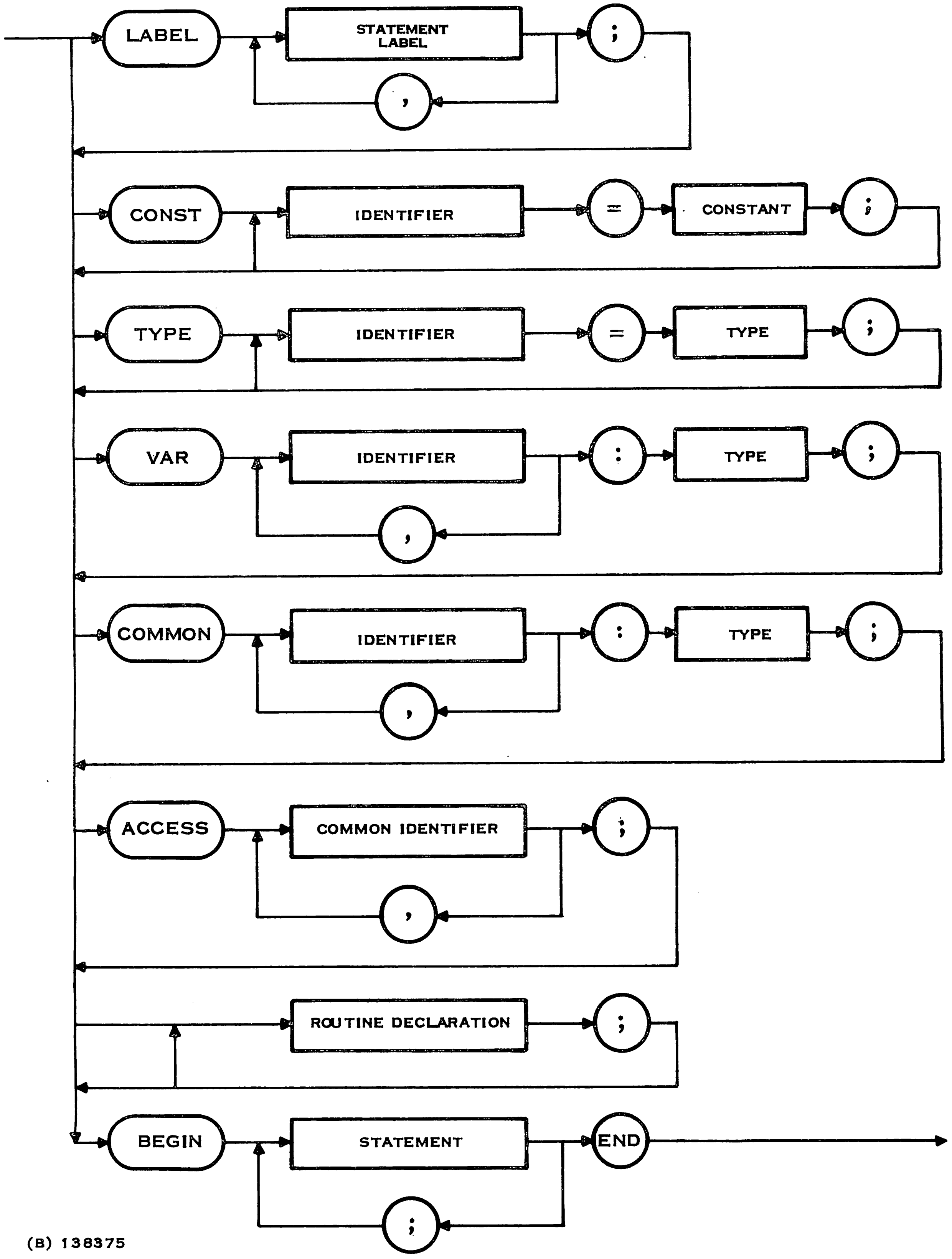
```
TYPE <identifier> = <type> {; <identifier> = <type>};
```

The identifiers may be used to denote the types within the scope of the declaration.

Examples:

```
TYPE REC = RECORD
                 NAME : PACKED ARRAY[1. . 20] OF CHAR;
                 ADDRESS : ADDREC
           END;
     VEC = ARRAY[1. . 10] OF REC;
```

Block:



(B) 138375

Figure 8-1. Syntax Diagrams of Block

Type declarations are a convenient means of declaring a user-defined type. A type declaration alone does not reserve memory for any variables of the specified type; a variable declaration described in the next paragraph is required also. The type may be defined in the variable declaration, as in the third example in that paragraph. Use of the type declaration promotes readability of the program; it saves work where several variables of the same type are declared; and it is required for a static array that is passed to a routine as a parameter.

**8.2.4 VARIABLE DECLARATION.** Variables must be declared before they are used in the program or routine. (Implicitly declared variables such as FOR control variables and WITH variables do not require declaration.) The syntax of a variable declaration is as follows:

```
VAR <identifier> {, <identifier>} : <type>
    { ;<identifier> { , <identifier>} : <type>};
```

The declaration specifies a type which is associated with each variable. The type may be specified by means of a standard type (X, Y, and Z in the example below), a previously declared type identifier (A below), or by defining the type directly in the VAR declaration (X below).

Example:

```
VAR X, Y, X : INTEGER;
    A : VEC;
    X : ARRAY[1. . 64] OF REAL;
```

**8.2.5 COMMON DECLARATION.** The syntax of a COMMON declaration is as follows:

```
COMMON <identifier> {, <identifier>}: <type>
       {;<identifier> {, <identifier>}: <type>};
```

The COMMON declaration declares the variables, specifies their type, and indicates that the variables are to be common as described in paragraph 8.9.

**8.2.6 ACCESS DECLARATION.** The syntax of the ACCESS declaration is as follows:

```
ACCESS <identifier> { , <identifier>};
```

where each identifier must also appear in a COMMON declaration whose scope includes the ACCESS declaration. An option is available which will restrict access to all nonlocal variables to those which have an ACCESS declaration. Access rules are described in paragraph 8.9.

**8.2.7 ROUTINE DECLARATIONS.** Declarations of procedures and functions are called routine declarations. The syntax for routine declarations is as follows:

```
<procedure declaration> ::= <procedure heading> <block>
                          | <procedure heading> FORWARD
                          | <procedure heading> EXTERNAL <linkage>

<function declaration> ::= <function heading> <block>
                         | <function heading> FORWARD
                         | <function heading> EXTERNAL <linkage>
```

<procedure heading> ::= PROCEDURE <identifier>; |
PROCEDURE <identifier> <parameter list>;

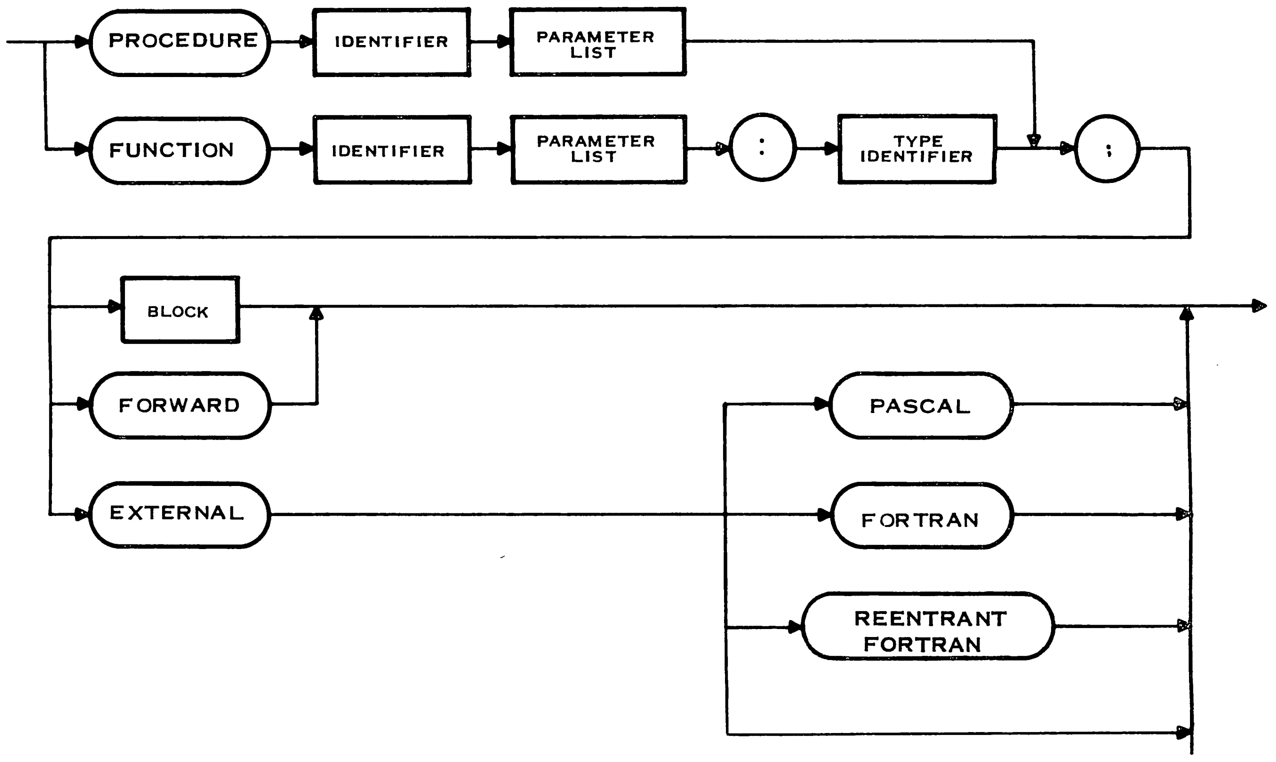
<function heading> ::=
FUNCTION <identifier>[<parameter list>] : <result type>;

<result type> ::= <type identifier>

<linkage> ::= PASCAL | FORTRAN | REENTRANT FORTRAN | <empty>

The syntax diagram for routine declarations is shown in figure 8-2. The parameter list portion of the heading is described in a subsequent paragraph. Each routine declaration contains either a block, a reference to a block included at a subsequent point in the same program (FORWARD), or reference to a block not included in the program (EXTERNAL).

Routine Declaration:



(A) 138376

**Figure 8-2. Syntax Diagram for Routine Declaration**

**8.2.7.1 Forward Declaration.** A routine declaration which refers to a body included elsewhere in the same program is called a forward declaration. This may be done for convenience, to allow the program modules to be managed by utility CONFIG. However, a routine declaration must precede a call to the routine. This requirement makes a forward declaration necessary when procedures A and R call each other. This is called indirect or mutual recursion.

Example:

```
PROGRAM MAIN;
PROCEDURE A(X, Y : REAL) ; FORWARD;

PROCEDURE R (L : INTEGER);
     VAR S, T : REAL;
     BEGIN (* R *)
     . . .
     A(S, T);
     . . .
     END (* R *);

PROCEDURE A;
VAR K : INTEGER:
     BEGIN (* A *)
     . . .
     R(K);
     . . .
     END (* A *);

BEGIN (* MAIN *)
. . .
END (* MAIN *).
```

In the forward declaration the name of the procedure and its parameters are specified. The parameters are omitted from the later declaration that includes the block of the procedure. The heading of this declaration consists of the procedure name only. Similarly, the parameters and result type are included in the forward declaration of a function. The heading of the function declaration that includes the block consists of the function name only.

Example:

```
FUNCTION F(X : REAL) : REAL; FORWARD;

PROCEDURE P(M : REAL);
     . . .
     BEGIN (* P *)
          X := F(A)
     END (* P *);

FUNCTION F;
     . . .
     BEGIN (* F *)
     P(T)
     END (* F *).
```

**8.2.7.2 External Declaration.** EXTERNAL declarations are used when a TI PASCAL program calls a routine which has been externally defined. The routine may be written in Pascal or any other language including assembly language. The <linkage> specifies which linkage is used. If <linkage> is <empty>, 'PASCAL' is assumed.

<linkage> ::= PASCAL | FORTRAN | REENTRANT FORTRAN | <empty>

Example:

```
PROCEDURE SKIPBLANKS(VAR F : TEXT) ; EXTERNAL PASCAL;
```

When parameters are passed to external FORTRAN routines, the following rules apply:

- Arrays (other than strings), records, and dynamic arrays are always passed by reference.
- VAR parameters are passed by reference.
- Value parameters which are strings (packed arrays of type CHAR having a lower bound of 1) and all other types are also passed by reference, but a copy is made and the copy is passed by reference.
- Procedures or functions may not be passed as parameters.

There is no provision in FORTRAN for passing parameters by value; all parameters are passed by reference. The reference is to the location of the parameter in the case of arrays (other than strings) and records. The reference is to the location of a copy in all other cases.

When Pascal linkage is specified or implied, the external routine must have been compiled at the same static nesting level as that of the EXTERNAL declaration. For example, if the EXTERNAL declaration is one of the declarations of the main program (the routine is global), the external routine must have been declared as a global routine within the environment in which it was compiled.

The parameter list of an EXTERNAL definition must be identical to that of the external routine. Furthermore, the declarations of data structures accessed by the external routine must be identical to those in the program in which the external routine was compiled, and the declaration sections in which these data structures are declared must be identical. For example, when a global routine accesses an array declared in the main program, the declaration of the array in the main program that calls the routine as an external routine must be identical to that of the main program in which the routine was compiled, and the other declarations for the main programs must be identical, also.

Assembly language routines may be written using any of these linkages and the routine may be declared EXTERNAL with the appropriate linkage specification. Appendix F contains additional information about assembly language routines.

## 8.3 STATEMENTS

The statement portion of the block consists of a compound statement that includes the statements of the block as component statements. The syntax of a TIP statement is as follows:

<statement> ::= [<statement label>:] <simple statement> |

[<statement label>:] [<escape label>:] <structured statement>

<simple statement> ::= <empty statement> | <assignment statement> |

<procedure statement> | <escape statement> | <goto statement> |

<assert statement>

<structured statement > ::= <compound statement> |

<conditional statement> | <repetitive statement> | <with statement>

<conditional statement> ::= <if statement> | <case statement>

<repetitive statement> ::=<for statement> | <while statement> |

<repeat statement>

<empty statement> ::= <empty>

<escape label> ::= <identifier>

<statement label> ::= <integer constant>

The syntax *diagram* is shown in figure 8-3.

## 8.4 PROCEDURE AND FUNCTION CALLS

A procedure or function may be called within the statement portion of the block within which it is declared, or in the statement portion of a block within the scope of the block within which it is declared. A call to a procedure is in the form of a procedure statement. The procedure statement syntax is as follows:

<procedure statement> ::= <procedure identifier>

[([<actual parameter>{, <actual parameter>}])]

<procedure identifier> ::= <identifier>

<actual parameter> ::= <expression> | <variable> |

<procedure identifier> | <function identifier>

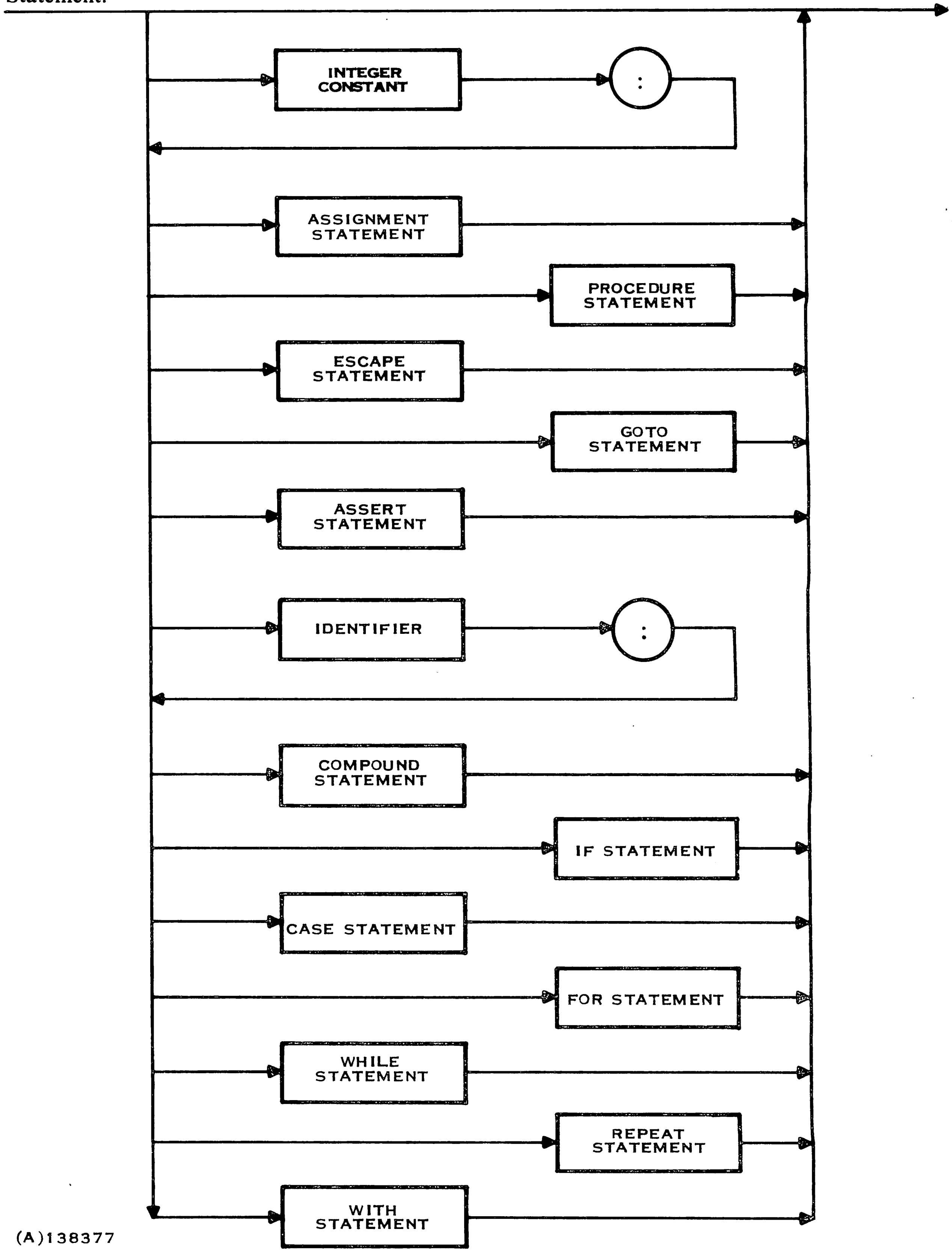
A function call may be used in any statement in which an expression is valid. The syntax is as follows:

<function identifier>[([<actual parameter>{, <actual parameter>}])]

<function identifier> ::= <identifier>

The function returns a result of the type specified in the function heading. This result becomes the value of the function call in the statement in which the call occurs. The actual parameter syntax is the same in the function call as in the procedure call.

Statement:



(A)138377

Figure 8-3. Syntax Diagram for Statement

## 8.5 PARAMETERS

The heading of a routine (paragraph 8.2.7) specifies the formal parameters for the routine. These formal parameters, also called dummy parameters do not denote a value until the routine is called. At that time the actual parameters are substituted for the formal parameters. This mechanism for passing parameters between programs and routines identifies the parameters to be passed, and should be used to the exclusion of other methods of transferring data between programs and routines. Parameters of functions are sometimes called arguments, and may not be used to return a result to the calling program or routine.

Parameters may be substituted by value or by reference as follows:

- Value Substitution — When the routine heading does not specify a type of substitution, value substitution, or call by value, is performed. The actual parameter is evaluated and its value is substituted for the formal parameter. This prevents the called routine from changing the value of the actual parameter in the calling program or routine.

- Variable Substitution — When the routine heading specifies variable substitution (call by reference) for a parameter, the address of the actual parameter is substituted for the formal parameter, and the address is used to access the actual parameter indirectly. Any assignment of a value to an actual parameter by a statement in the routine alters the value of the actual parameter in the calling program or routine. Reserved word VAR precedes the specification of the formal parameter to specify variable substitution.

Examples:

```
PROGRAM SAMPLE;
VAR A, V : INTEGER;

PROCEDURE ABS1(X : INTEGER; VAR Y : INTEGER);
    BEGIN
        IF X < 0 THEN X := -X;
        Y := X
    END;

BEGIN (* SAMPLE *)
    A := -5; V := 0;
    ABS1(A, V);
    WRITELN(A, V)
END (* SAMPLE *).
```

Since X is a value parameter, the value of A is not changed within procedure ABS1. Y, however, is a variable parameter, so the assignment to Y in ABS1 changes the value of V in the main program. The values printed are -5 and 5.

Value parameter transmission offers the security of preventing inadvertent changes to program values by a routine. It also may be an efficient way to pass simple variables as parameters. However, since this method involves copying values, it may be inefficient to pass data structures such as large arrays by value.

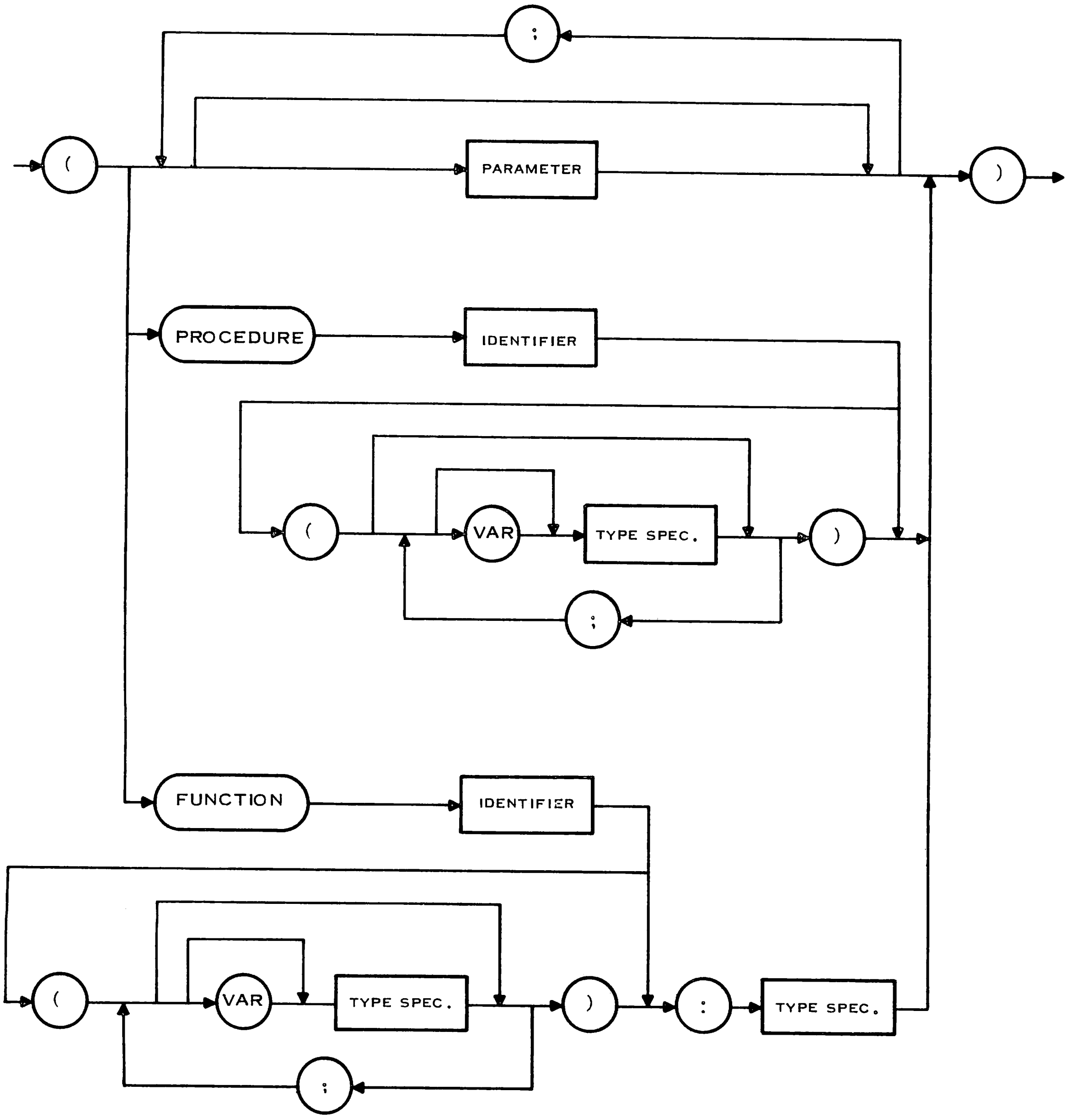
The only way a function can return values is by assigning a value to the identifier which denotes the function. This result type must be an enumeration, REAL, FIXED, DECIMAL, or pointer type.

The syntax for the parameters of a routine declaration are as follows:

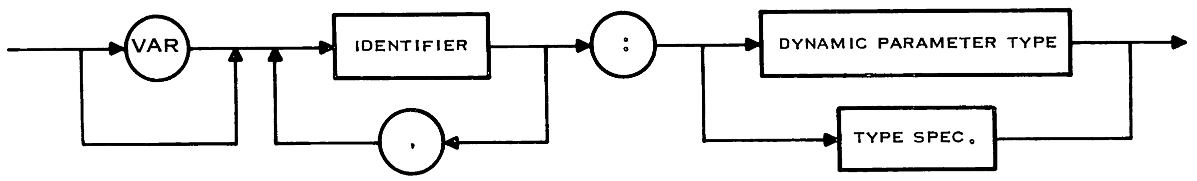
```
<parameter list> ::= ([<anyparameter> { ; <anyparameter>}])

<anyparameter> ::= <parameter> |
    PROCEDURE <identifier> [([[VAR] <type specification>
        {; [VAR] <type specification>}])] |
    FUNCTION <specification> [([[VAR] <type specification>
    {;[VAR] <type specification>}])]:<type specification>

<parameter> ::= [VAR] <identifier> {,<identifier>}: <partype>

<partype> ::= <type specification> | <dynamic parameter type>

<dynamic parameter type> ::=
    [PACKED] ARRAY "["<parameter index> {, <parameter index>}"]"
    OF <type specification> | [PACKED] SET OF
    "["<dynamic parameter index type>"]"

<parameter index> ::= <subrange type> | <type identifier>
    | <dynamic parameter index type>

<dynamic parameter index type> ::= <manifest constant>. . ?

<type specification> ::= INTEGER | LONGINT | CHAR | BOOLEAN
    | REAL[( <integer constant>)] | FIXED( <integer constant>),
        [<sign>] <integer constant>)
    | DECIMAL( <integer constant>, [<sign>] <integer constant>)
    | <type identifier>
```

The syntax diagrams are as follows:

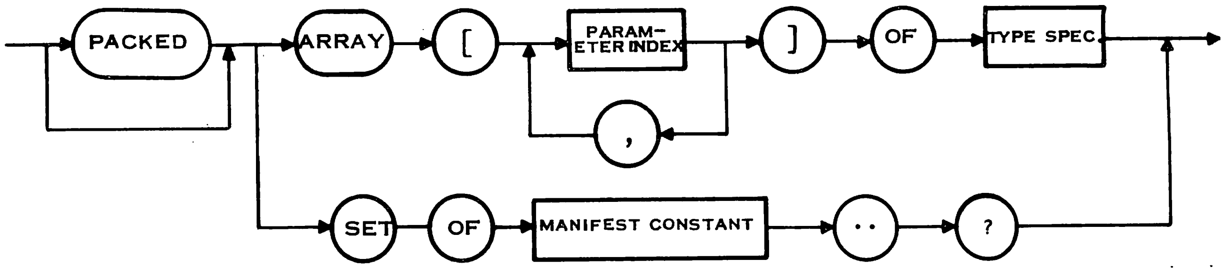
Parameter list:



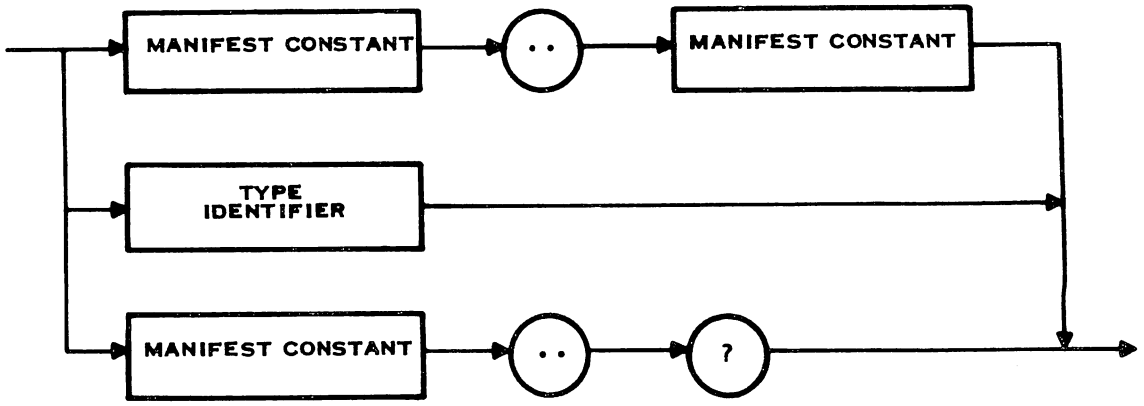
Parameter:



Dynamic parameter type:



Parameter index:



**8.5.1 RULES FOR PARAMETERS.** The following rules apply to parameters:

- The number of formal and actual parameters must be the same.
- A formal value parameter must be compatible with the type of the corresponding actual parameter, as described in paragraph 6.8. For example, if a formal parameter is an array of integers with index 1. . 10, the actual parameter must be an array of integers (or subrange of integers) with index 1. . 10.

- For variable parameters, the type of the actual parameter must be identical to that of its corresponding formal parameter, with the following exceptions:
  - The actual parameter may be of a type which is a subrange of the type of the corresponding formal parameter, or vice versa.
  - If the formal parameter is a dynamic array or set parameter, the actual parameter must be an array or set, respectively, packed or unpacked as the formal parameter, with bounds equal to those specified in the declaration of the formal parameter and not set by ?, and with component type identical to that of the formal parameter.
- For value parameters, the corresponding actual parameter may be any expression of the same type as the parameter.
- For variable parameters the actual must be a variable and may not be a component of a packed structure.
- File parameters must be passed as variable parameters.

**8.5.2 DYNAMIC ARRAY AND SET PARAMETERS.** In order to write routines which accept arguments which are arrays of arbitrary dimension or sets of arbitrary size, it is necessary to use dynamic parameter types. A dynamic parameter type is an array or set with a lower bound which is fixed at compile time (a manifest constant) and an upper bound which is determined at runtime and denoted by a question mark. The subrange expression is of the form:

<manifest constant>. .?

and can be used to specify either an index type of an array or the size of a set as a formal parameter. This subrange matches any index type with the same lower bound. The standard function UB may then be used to determine at runtime the value of the upper bound of the array or set which is passed as an actual parameter.

A multidimensional array may have dynamic upper bounds for one or more dimensions.

The actual parameter corresponding to a dynamic parameter type formal parameter must meet the requirements outlined in the preceding paragraph.

Example:

```
FUNCTION MAX(VECTOR : ARRAY [1. . ?] OF INTEGER) : INTEGER;
VAR HOLDMAX : INTEGER;
BEGIN
    HOLDMAX := VECTOR[1];
    FOR I := 2 TO UB(VECTOR) DO
    (* HERE HOLDMAX = MAX OF THE 1ST I-1 ELEMENTS OF VECTOR *)
        IF HOLDMAX < VECTOR[I]
            THEN HOLDMAX := VECTOR[I];
    MAX := HOLDMAX
END; (* MAX *)
```

**8.5.3 PROCEDURE AND FUNCTION PARAMETERS.** In addition to passing variables as parameters, it is possible to pass procedures or functions as parameters. In the case of a procedure parameter, the actual procedure must have the same number of parameters as the formal parameter declares, and the parameters must be of compatible types. In the case of a function parameter, the result type must be compatible, and the number and types of parameters must agree. The specific procedure or function in the call is performed whenever the routine calls the procedure or function declared as a formal parameter.

Example:

```
PROCEDURE FINDMIN(FUNCTION F(INTEGER):REAL; VAR MIN:INTEGER);
VAR K : INTEGER;
BEGIN K := 1;
    FOR I := 2 TO 10 DO
        IF F(I) < F(K) THEN K := I;
    MIN := K
END;
```

Procedure FINDMIN has one parameter which is a function having a single integer type argument and a real result and has a second parameter which it sets to the integer value between 1 and 10 for which the function is minimum. An example of a call to FINDMIN is as follows:

```
FINDMIN (RCPR,X)
```

Assuming that RCPR is a function that returns the reciprocal of an integer as a real number, integer variable X would be set to 10 when the call is executed. Another example is as follows:

```
FINDMIN (SQRI,X)
```

Assuming that SQRI returns the square of an integer in real number format, integer variable X would be set to 1 when the call is executed.

**8.5.4 PROCEDURES AND FUNCTIONS WITHOUT PARAMETERS.** A procedure may have no parameters in which case the form of the heading is either

PROCEDURE <identifier>;

or

PROCEDURE <identifier>();

and the call is a statement of the form

<identifier>

or

<identifier> ()

Similarly, if a function has no parameters it may be declared with heading of either of the forms

```
FUNCTION <identifier> : <type identifier>;
```

or

```
FUNCTION <identifier> () : <type identifier>;
```

and the function may be called by either of the forms

```
<identifier>
```

or

```
<identifier> ()
```

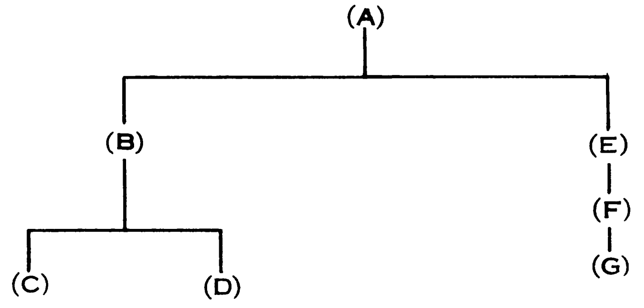
## 8.6 BLOCK STRUCTURE

A Pascal program consists of a program block that contains the declarations and statements of the main program. Among the declarations of a program there may be routine declarations. The main program block includes the headings of any routines declared but excludes the blocks (declarations and statements) of these routines. Similarly, the block of each routine may contain declarations of other routines. The blocks of these routines are not considered to be part of the block in which they are declared. This nesting of routines within the main program and within routines forms a block structure.

Blocks are associated with a level of nesting. The main program is at level 1. Those routines declared in the main program are at level 2. Routines declared in level 2 routines are at level 3, etc. The following example illustrates a possible structure of routine declarations:

```
PROGRAM A
    PROCEDURE B
        PROCEDURE C
        PROCEDURE D
    PROCEDURE E
        PROCEDURE F
            PROCEDURE G
```

Procedures B and E are declared in program A, procedures C and D are declared in procedure B, procedure F is declared in E, and G is declared in F. One way of depicting this structure is by means of a declaration tree, which is illustrated by the following figure:



In the example, program A is at level 1, routines B and E are at level 2, routines C, D, and F are at level 3, and routine G is at level 4.

## 8.7 SCOPE AND EXTENT

The block structure determines the scope of identifiers that denote constants, variables, and routines. The scope of an identifier consists of the portion of the total block structure in which that identifier can be accessed (e.g., a variable may be assigned a value, a variable or constant may be used in a computation, or a routine may be called).

The variables that are declared in the main program block are global variables and apply throughout the entire program. Similarly, the routines and all other declarations of the main program block have a global scope. Declarations in other blocks of the block structure have a limited scope. Referring to the declaration tree example in the preceding paragraph, the scope of declarations in routine B includes routines B, C, and D, the portion of the tree declared in routine B. Similarly, the scope of declarations in routine E includes routines F and G. The declarations in routines C, D, and G declare local identifiers; i.e., these identifiers apply only to the routines in which they are declared.

As previously stated, a routine can call any routine previously declared in the block in which that routine is declared or in any enclosing block. This allows a routine to call itself, a recursive call.

An identifier must have a unique meaning within the block in which it is declared. However, a nonlocal identifier may be redefined in a routine at a lower level. Within the scope of the original declaration, exclusive of the scope of the redefinition, the original declaration applies. Referring to the declaration tree example, an identifier declared in program A could be redefined in routine F. The scope of the declaration in routine F would be routines F and G. The scope of the declaration in program A would include the main program block, and routines B, C, D, and E.

Example:

```
PROGRAM MAIN;
VAR X : REAL;
PROCEDURE P;
    VAR X : INTEGER;
    BEGIN (* P *)
        X := 10
    END (* P *);

BEGIN (* MAIN *)
    . . .
END (* MAIN *).
```

Within procedure P, the identifier X denotes the local integer variable, and the global real variable X is not accessible by the name X.

Note that this name precedence rule applies to constant, type, variable, and routine identifiers and to identifiers used as scalar constants. Also, it is possible to redefine any of the standard identifiers which denote objects such as the procedure READ, although this is not recommended since it is likely to result in a program which is difficult to understand.

The blocks of a TIP program are not the only units of scope. Other units of scope include record declarations, FOR statements, WITH statements, and structured statements. The record declaration is the unit of scope for the field identifiers in the record. A FOR statement is the unit of scope for the control variable. The WITH statement is the unit of scope for the with variables. The structured statement is the unit of scope for the escape label. Redefinition of an identifier within either of these units of scope is applicable only in the unit of scope within which it is redefined.

The scope of a routine relates both to identifiers and to calls to routines, and there are subtle differences. Figure 8-4 shows a program structure consisting of a program block MAIN, and routine blocks RA, RB, RC, RD, RE, and RF. The MAIN block includes declarations of variables a, b, and c, and of routines RA and RB. The routine RA block includes declarations of variables a and f, and of routine RC. The routine RB block includes declarations of variables c, d, and e, and of routines RD and RE. The routine block RC includes a single declaration, that of variable g. The routine block RD includes declarations of variables f, g, and h. The routine block RE includes declarations of variables a and e and of routine RF. The routine block RF includes declarations of variables e and h.

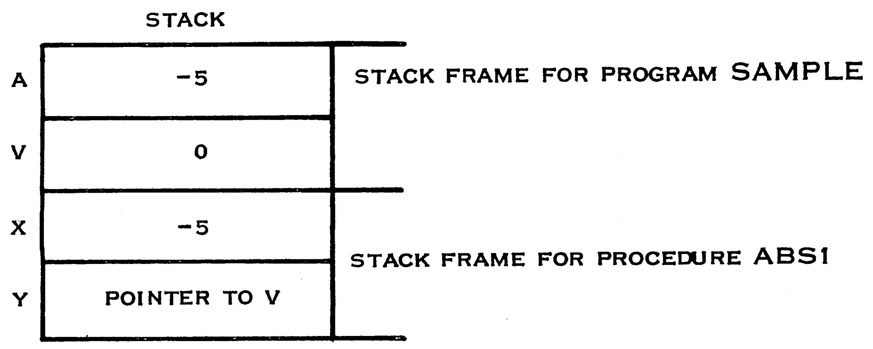
A routine has access to variables declared in the routine and to variables declared in the block in which the routine is declared. At successively higher levels the routine has access to variables declared in the same block as any ancestor of the routine. Variables having the same identifier as those declared at higher levels redefine the identifier for the level at which they are declared and for lower levels. These rules result in the accessability defined for each block in figure 8-4.

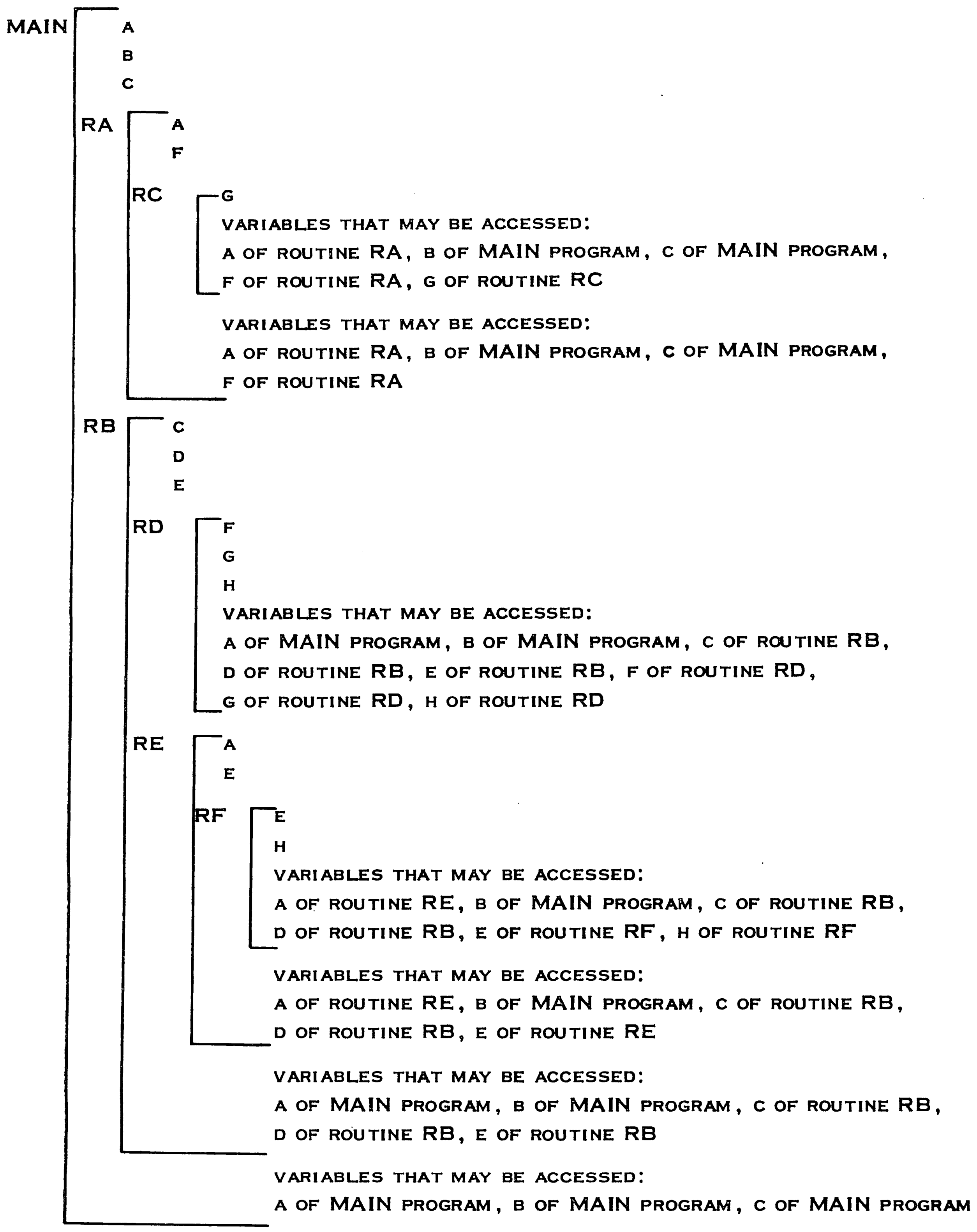
Figure 8-5 shows another program structure consisting of a program block MAIN, and routine blocks RA, RB, RC, RD, RE, RF, RG, and RH. The program block includes declarations of routines RA, RB, and RC. The routine RA block includes the declaration of routine RD. The routine RB block includes declarations of routines RE and RF. The routine RC block includes the declaration of routine RG, which declares routine RH.

A routine can call any other routine declared in the block in which it is declared. A routine may not call a routine that has not been declared previously. However, appropriate FORWARD declarations of routines allow a routine to call any other routine declared in the same block. At successively higher levels, a routine may call routines declared in the same block as any ancestor of the routine, when appropriate FORWARD declarations are included. These rules result in the availability of routines shown in figure 8-5.

The scope and extent of identifiers is related to the allocation of space for the data for routines. This space is obtained from a region of memory called the stack. The space used for the parameters, local variables, and temporary values of a single routine is called a stack frame. When execution of the routine is completed, the stack frame is deallocated and that space is available to be used by other routines.

Using the example program SAMPLE in paragraph 8.5, the stack allocation at the point at which procedure ABS1 has been called is as follows:

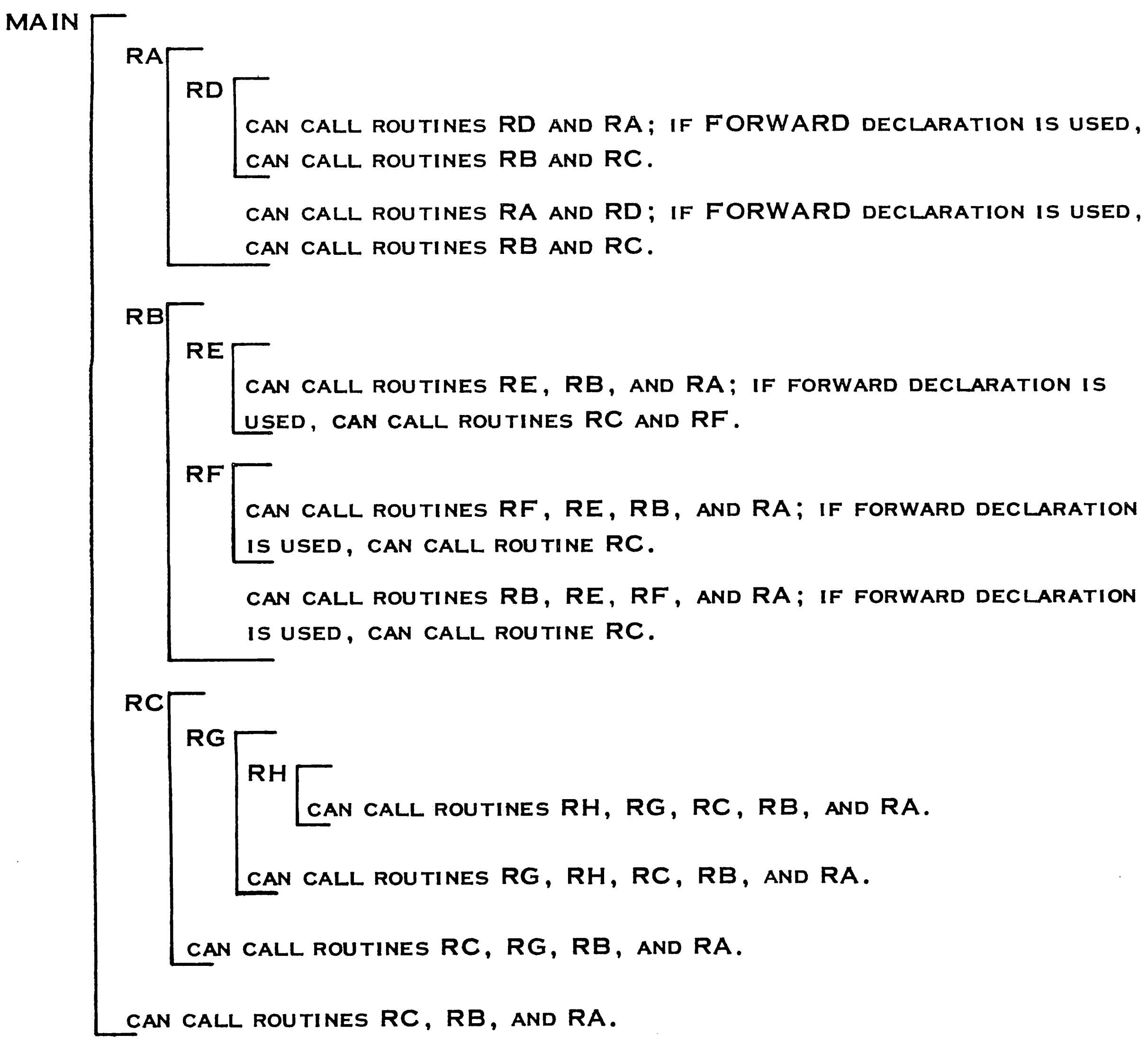

STACK
A
-5
STACK FRAME FOR PROGRAM SAMPLE
V
0
X
-5
STACK FRAME FOR PROCEDURE ABS1
Y
POINTER TO V



```
MAIN  A
      B
      C

  RA   A
       F

    RC   G
         VARIABLES THAT MAY BE ACCESSED:
         A OF ROUTINE RA, B OF MAIN PROGRAM, C OF MAIN PROGRAM,
         F OF ROUTINE RA, G OF ROUTINE RC

         VARIABLES THAT MAY BE ACCESSED:
         A OF ROUTINE RA, B OF MAIN PROGRAM, C OF MAIN PROGRAM,
         F OF ROUTINE RA

  RB   C
       D
       E

    RD   F
         G
         H
         VARIABLES THAT MAY BE ACCESSED:
         A OF MAIN PROGRAM, B OF MAIN PROGRAM, C OF ROUTINE RB,
         D OF ROUTINE RB, E OF ROUTINE RB, F OF ROUTINE RD,
         G OF ROUTINE RD, H OF ROUTINE RD

    RE   A
         E

      RF   E
           H
           VARIABLES THAT MAY BE ACCESSED:
           A OF ROUTINE RE, B OF MAIN PROGRAM, C OF ROUTINE RB,
           D OF ROUTINE RB, E OF ROUTINE RF, H OF ROUTINE RF

           VARIABLES THAT MAY BE ACCESSED:
           A OF ROUTINE RE, B OF MAIN PROGRAM, C OF ROUTINE RB,
           D OF ROUTINE RB, E OF ROUTINE RE

           VARIABLES THAT MAY BE ACCESSED:
           A OF MAIN PROGRAM, B OF MAIN PROGRAM, C OF ROUTINE RB,
           D OF ROUTINE RB, E OF ROUTINE RB

           VARIABLES THAT MAY BE ACCESSED:
           A OF MAIN PROGRAM, B OF MAIN PROGRAM, C OF MAIN PROGRAM
```

(A) 138378

**Figure 8-4. Access to Variables in Program Structure**



```
MAIN
    RA
        RD
            CAN CALL ROUTINES RD AND RA; IF FORWARD DECLARATION IS USED,
            CAN CALL ROUTINES RB AND RC.

            CAN CALL ROUTINES RA AND RD; IF FORWARD DECLARATION IS USED,
            CAN CALL ROUTINES RB AND RC.

    RB
        RE
            CAN CALL ROUTINES RE, RB, AND RA; IF FORWARD DECLARATION IS
            USED, CAN CALL ROUTINES RC AND RF.
        RF
            CAN CALL ROUTINES RF, RE, RB, AND RA; IF FORWARD DECLARATION
            IS USED, CAN CALL ROUTINE RC.

            CAN CALL ROUTINES RB, RE, RF, AND RA; IF FORWARD DECLARATION
            IS USED, CAN CALL ROUTINE RC.

    RC
        RG
            RH
                CAN CALL ROUTINES RH, RG, RC, RB, AND RA.

            CAN CALL ROUTINES RG, RH, RC, RB, AND RA.

        CAN CALL ROUTINES RC, RG, RB, AND RA.

    CAN CALL ROUTINES RC, RB, AND RA.
```

(A) 138379

**Figure 8-5. Access to Routines in Program Structure**

The value of A has been copied into X, and Y, has been set to a pointer to V so that a reference to Y is actually an indirect access to V.

When a routine which is currently active is re-activated, new instances of all local variables that are not common variables and parameters are allocated. The execution of the current activation of the routine always references these currenlty allocated variables. (This technique makes all procedures and functions reentrant).

The method of space allocation results in the identifiers having an extent, i.e., a period of time during which they are accessible. The extent of all statically defined quantities is the duration of execution of the unit of scope in which they are declared. The extent of dynamically allocated variables is the duration of program execution between the call of NEW which creates the variable and the call of DISPOSE which frees the space allocated to them. When DISPOSE is not called, the extent continues to program termination.

## 8.8 SIDE EFFECTS

A function returns a value through the identifier of the function. When a function changes the value of a variable other than the local variables of the function, that change is called a side effect. TIP prevents side effects by restricting assignments, procedure and function calls, and the use of nonlocal variables in user defined functions. The variable (left-hand side) of an assignment statement may not be any of the following:

- A nonlocal variable.
- A variable parameter of the function.
- A COMMON variable.
- A pointer variable followed by @.

User defined functions may not contain:

- Procedure statements that call user-defined procedures or the standard procedure READ.
- Calls to externally defined functions.
- Procedures or externally defined functions as parameters.
- A WITH statement that contains a record variable followed by @.
- Calls to NEW or DISPOSE that have parameters that are not either local variables or value parameters.
- The array into which data is packed by a PACK procedure call must be a local variable or a value parameter.
- The array into which data is placed by an UNPACK procedure call must be a local variable or a value parameter.
- The string into which data is placed by an ENCODE procedure call must be a local variable or a value parameter.
- The variable into which data is placed by a DECODE procedure call must be a local variable or a value parameter.
- The nonfile parameters of a READ procedure and the OVAL parameter of an IOTERM procedure must be local variables or value parameters.
- Procedures RESET, REWRITE, EXTEND, WRITEEOF, SKIPFILES, CLOSE, SETNAME, SETMEMBER, and IOTERM may be used only with parameters of file type that are local variables.

These restrictions may make it necessary to use a procedure for some purposes for which a function might otherwise be used. However, this inconvenience is more than made up for by the reliability which is gained from preventing side-effects.

## 8.9 COMMON AND ACCESS DECLARATIONS

COMMON and ACCESS declarations are used to declare variables which may be shared with other routines falling within the scope of the COMMON declaration, or with externally compiled routines. One characteristic of COMMON variables is that they are not allocated on the stack, and hence are not deallocated when execution of the routine where they are declared terminates. This makes it possible to save the value of "local" variables from one activation of a routine to the next. Another characteristic of COMMON variables is that their extent consists of the execution time of the entire program. The syntax of a COMMON declaration is:

```
COMMON <identifier> {, <identifier>} : <type>
    {; <identifier> {, <identifier>} : <type>} ;
```

An identifier may appear in only one COMMON declaration. The identifier is both the variable name used internally to the program and also the external name by which the implementation makes the COMMON variable available externally. Since the linkage editor recognizes only six characters, COMMON identifiers should be limited to this size.

In order for a COMMON variable to be accessible within a routine, the routine must contain an ACCESS declaration for the COMMON variable. A routine that contains a COMMON declaration must include an ACCESS declaration if it accesses a common variable. The normal scope rules do not apply to ACCESS declarations. Even if a routine falls within the scope of an ACCESS declaration at a higher level, the COMMON variables are not accessible within the routine unless an explicit ACCESS declaration appears in the routine. Each ACCESS declaration must fall within the scope of the COMMON declaration of the identifier for which ACCESS is declared.

The syntax of an ACCESS declaration is

```
ACCESS <identifier> {, <identifier>};
```

There is a compiler option that allows access only to those global variables that are specified in ACCESS declarations. When this option applies, variables named in ACCESS declarations are not necessarily common variables.

COMMON declarations accomplish the following purposes:

- Provide more restricted access than normal declarations.
- Enable variables to be shared with external routines such as FORTRAN routines.
- Provide an extent for variables greater than the duration of execution of the routine in which they are declared.

Example:

```
PROCEDURE P
    COMMON X : INTEGER
    PROCEDURE Q
        VAR Y : INTEGER
        PROCEDURE R
            ACCESS X
                PROCEDURE S
```

The COMMON variable X is accessible only within the body of routine R since no other procedure has an ACCESS declaration.

The next example illustrates the correspondence between a FORTRAN named common block and a TIP common variable.

```
FORTRAN:
        COMMON /TAB/ I, X, M(5)

TI PASCAL:
        COMMON TAB : RECORD
            I : INTEGER;
            X : REAL;
            M : ARRAY[1. . 5] OF INTEGER
          END
```

In FORTRAN, the single external name TAB is associated with the block of variables I, X, and M. Since each individual common variable in TIP corresponds to a common block, this example uses a record with three fields to achieve the same effect as the FORTRAN declaration.

# SECTION IX

# COMPILER OPTIONS

## 9.1 GENERAL

The TIP compiler supports a set of options that control the format and content of the listing, options that control the content of the object file, options that control runtime checks, and miscellaneous options. Options may be enabled or disabled by being named in option comments. Certain options may be named in option comments that affect the entire program; certain other options may be named in an option comment at the start of a routine only; but most options may be named in an option comment at any point in a program at which any comment would be allowed. This provides a high degree of flexibility in the control of options.

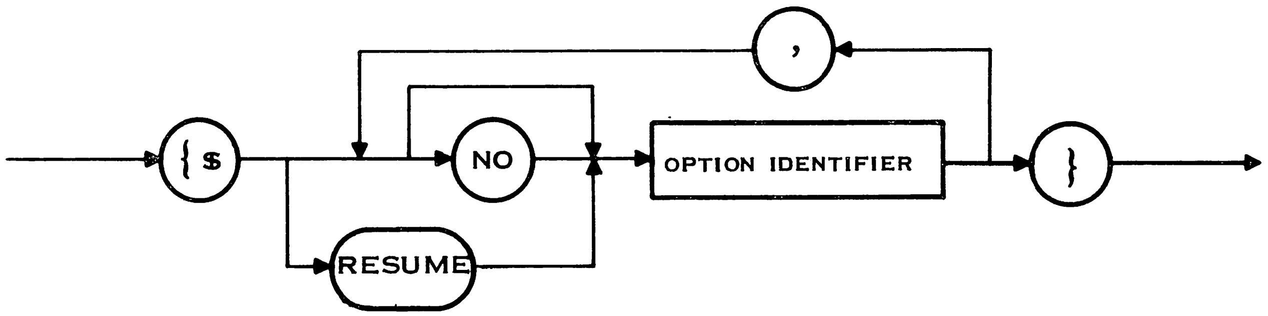
## 9.2 THE OPTION COMMENT

The comment that controls the enabling or disabling of an option is enclosed within braces ({ }). A dollar sign immediately following the left brace (or the alternate character pair (* ) identifies the comment as an option comment. The syntax for an option comment is as follows:

```
<option comment>::= "{ "$ <option> ,<option> " } "
<option> ::= [NO] <option identifier>
             |[RESUME] <option identifier>
```

The syntax diagram is as follows:

Option comment:



The option identifier is the name of an option; one of the option names listed in table 9-1. Each option has a value of TRUE or FALSE, independent of the value of any other option. The default value which each option has at the beginning of the program is shown in table 9-1. The presence of an option name in an option comment sets the value of the option to TRUE, unless the keyword NO or RESUME precedes the option name. When the keyword NO precedes the option name, the value of the option is set to FALSE. When the keyword RESUME precedes the option name, the value of the option is set to the value the option name had upon entry to the routine (if the option comment is in a routine) or upon entry to the program.

**Table 9-1. TIP Compiler Options**

| Option Name | Default Value | Level |
|---|---|---|
| 72COL | TRUE | Statement |
| ASSERTS | TRUE | Statement |
| CKINDEX | FALSE | Statement |
| CKOVER | FALSE | Statement |
| CKPTR | FALSE | Statement |
| CKPREC | FALSE | Statement |
| CKSET | FALSE | Statement |
| CKSUB | FALSE | Statement |
| CKTAG | FALSE | Statement |
| FORINDEX | FALSE | Statement |
| GLOBALS | FALSE | Routine |
| LIST | TRUE | Statement |
| MAP | FALSE | Routine |
| NULLBODY | FALSE | Routine |
| OBJECT | TRUE | Routine |
| PAGE | FALSE (Note 1) | Statement |
| PROBER | FALSE | Routine |
| PROBES | FALSE | Routine |
| ROUND | TRUE | Statement |
| STANDARD | FALSE | Program |
| TRACEBACK | TRUE | Routine |
| UNSAFEOPT | FALSE | Routine |
| WARNINGS | TRUE | Statement |
| WIDELIST | FALSE | Program |

Note: 1. PAGE has a default value of FALSE and is set to FALSE before the beginning of each source line.

If a comment and an option comment are required, the comment must be entered as if the option comment were not there; it may not be embedded within the option comment.

Example:

```
{$LIST} {THIS TURNS ON THE LISTING OPTION}
```

When one or more spaces separate the left brace ({) or left parenthesis and asterisk ((*, the alternate characters) from the dollar sign, the result is a comment, not an option comment. The following example has no effect on the list option:

```
{ $LIST}
```

Option comments are effective within the scope of the routine within which they occur (if they occur within a routine). When a routine is entered (during compilation), the values of options are stored. Upon exit from the routine, the stored values are restored. Thus, with the exception of the PAGE option, the value of an option changed by an option comment remains the value of that option until another option comment containing the option name is processed, or until exit from the routine. The PAGE option is set FALSE at the beginning of each source line.

Table 9-1 also lists the level of each option. The WIDELIST option and the STANDARD option are program level options that can be changed only by entering the option comment to change them ahead of the program heading. The scope of the program begins at the program identifier (following the keyword PROGRAM in the program heading and ends at the period that terminates the program. The program level options must be changed prior to the beginning of this scope.

Routine level options may be changed prior to the program heading, following the semicolon that terminates the program heading or the routine heading, or immediately after the BEGIN keyword that begins the compound statement of the routine. The scope of a routine begins in the routine heading following the routine identifier and before the parameter list. The scope of a routine ends at the semicolon following the END keyword that terminates the compound statement of the routine. The following are the routine level options:

| | | |
|---|---|---|
| GLOBALS | MAP | NULLBODY |
| OBJECT | PROBER | PROBES |
| TRACEBACK | UNSAFEOPT | |

The remaining options are statement level options that can be changed by a comment statement at any point in the program.

When a comment statement names an option at a level which cannot be changed at that point, the option remains unaltered and a warning or error message is issued.

## 9.3 LISTING CONTROL OPTIONS

Listing control options affect the format and/or content of the compiler listing. The options are described in the following paragraphs.

**9.3.1 LIST OPTION.** The LIST option enables or disables the source listing. When the value of the option is TRUE (default) the compiler writes the listing of the source program. When the value of the option is FALSE, the compiler lists only the lines that contain errors, and the error messages. The LIST option is a statement level option.

**9.3.2 WIDELIST OPTION.** The WIDELIST option enables or disables the wide format of the source listing. When the value of the option is TRUE the compiler writes a source line number at the beginning of each source line, and a statement number at the beginning (following the source line number) of each statement within the compound statement of each program or routine block. When the value of the option is FALSE (default), source line and statement numbers are omitted. Figure 9-1 shows a source listing with the WIDELIST option enabled. The WIDELIST option is a program level option.

The WIDELIST option does not control the actual width of the lines of the source listing. The logical record length for the OUTPUT file determines the width of the lines. When the logical record length is 80, the first 72 characters of the source are listed. When the logical record length is 88 or greater the entire source line is printed.

**9.3.3 MAP OPTION.** The MAP option enables or disables the inclusion of a map of identifiers in the source listing. When the value of the option is TRUE the compiler includes a map of variables and commons for each routine and for the main program at the end of each routine and at the end of the program. When the value is FALSE (default), the map is omitted. Figure 9-1 shows a source listing with the MAP option enabled. The MAP option is a routine level option.

As shown in figure 9-1, the map of identifiers lists the first eight characters of the variable name and the kind of identifier. The following kinds are displayed:

- PARAMETER
- VARIABLE
- FIELD
- COMMON

The next column shows the size of the data item in bits or bytes. Alternatively, the column shows a level and displacement in the case of a dynamic array. The level and displacement are the level and displacement of the variable that contains the size of the dynamic array.

The third column shows the displacement of the item in the stack frame for the routine. In the case of variables and parameters, the displacement is a hexadecimal value. In the case of a field, the displacement is in bytes and bits from the displacement of the record that includes the field.

The fourth column may contain either DIRECT, INDIRECT, or PACKED. DIRECT identifies variables and value parameters. INDIRECT identifies reference parameters. PACKED identifies fields of packed records.

The fifth column applies only to packed fields and shows the order of the packing of the fields by identifying the bit positions of the field with Xs.

**9.3.4 PAGE OPTION.** The PAGE option provides a means of forcing a new page in the listing. When the value of the option is TRUE, the compiler performs a new page operation and writes the next line at the top of the new page. The compiler also sets the PAGE option to the FALSE value. When the value of the option is FALSE (default) the compiler does not perform a new page operation unless the current page has been filled. The PAGE option is a statement level option.

**9.3.5 WARNINGS OPTION.** The WARNINGS option enables or disables the listing of warning messages by the compiler. When the value of the option is TRUE (default) the compiler includes warning messages with the source lines of the listing. When the value of the option is FALSE, the compiler omits warning messages in the listing. The WARNINGS option is a statement level option.

## 9.4 OBJECT CODE OPTIONS

Object code options affect the content of the object code file. The options are described in the following paragraphs.

**9.4.1 OBJECT OPTION.** The OBJECT option enables or disables writing of the object code file for one or more blocks of the program. When the value of the option is TRUE (default), the compiler writes the object code. When the value of the option is FALSE, the compiler does not write object code. The OBJECT option is a routine level option.

**9.4.2. NULLBODY OPTION.** The NULLBODY option enables or disables writing of the object code corresponding to the compound statement of a routine or program and its component statements. When the value of the option is TRUE, the compiler does not write a module. When the value of the option is FALSE (default), the compiler writes object code. The NULLBODY option should not be set to TRUE in a user program. It is used in separate compilation under control of the configuration processor (Section XII). The NULLBODY option is a routine level option.

```
 DXPSCL        1.4.X  78.270   TI 990 PASCAL COMPILER   11/06/78 13:54:25

(**WIDELIST,MAP*)
   2     PROGRAM DIGIO;
   3     TYPE CHBUF = ARRAY(.1..6.) OF CHAR;
   4     VAR BUFF           : CHBUF;
   5         I,NUM          : INTEGER;
   6     PROCEDURE CCHAR (BUFF:CHBUF;VAR NUM:INTEGER;I:INTEGER);
   7     BEGIN   (* CCHAR *)
   8       NUM:=0;
   9   3   FOR J:= 1 TO I DO
  10   4     IF BUFF(.J.)>='0' AND BUFF(.J.)<='9' THEN
  11   5       NUM:=NUM*10+ORD(BUFF(.J.))-ORD('0')
  12   6 END;

   MAP OF IDENTIFIERS FOR  CCHAR

IDENTIFIER NAME   KIND          SIZE        STACK               PICTURE
                            (BYTES,BITS) DISPLACEMENT     (PACKED FIELDS ONLY)
                            LEVEL(DISPL)  (BYTE,BIT)

BUFF             PARAMETER  (12,0)       #0028    DIRECT
NUM              PARAMETER  (2,0)        #0034    INDIRECT
I                PARAMETER  (2,0)        #0036    DIRECT

  13     PROCEDURE CINT (NUM:INTEGER);
  14     VAR I           : INTEGER;
  15     BEGIN   (* CINT *)
  16       I:=NUM DIV 10;
  17   3   IF I <> 0 THEN CINT(I);
  18   4   WRITE (CHR(NUM MOD 10 + ORD('0')))
  19   5 END;

   MAP OF IDENTIFIERS FOR  CINT

IDENTIFIER NAME   KIND          SIZE        STACK               PICTURE
                            (BYTES,BITS) DISPLACEMENT     (PACKED FIELDS ONLY)
                            LEVEL(DISPL)  (BYTE,BIT)

NUM              PARAMETER  (2,0)        #0028    DIRECT
I                VARIABLE   (2,0)        #002A    DIRECT

  20     BEGIN   (* DIGIO *)
  21       WRITELN('ENTER 1 TO 5 DIGITS');
  22   3   RESET(INPUT);
  23   4   I:=I+1;
  24   5   WHILE NOT EOLN DO BEGIN (* INPUT CHARS *)
  25   6     READ (BUFF(.I.));
  26   7     I:= I+1;
  27   8   END;   (* INPUT CHARS *)
  28   9   I := I-1;
  29  10   CCHAR(BUFF,NUM,I);
  30  11   NUM:=NUM+25;
  31  12   CINT(NUM);
  32  13   WRITELN;
  33  14 END.

   MAP OF IDENTIFIERS FOR  DIGIO

IDENTIFIER NAME   KIND          SIZE        STACK               PICTURE
                            (BYTES,BITS) DISPLACEMENT     (PACKED FIELDS ONLY)
                            LEVEL(DISPL)  (BYTE,BIT)


       BUFF             VARIABLE   (12,0)       #0080    DIRECT
       I                VARIABLE   (2,0)        #008C    DIRECT
       NUM              VARIABLE   (2,0)        #008E    DIRECT


       MAXIMUM NUMBER OF IDENTIFIERS USED = 14
       INSTRUCTIONS =     33 (LESS    0 WORDS OF DEAD CODE REMOVED)
         CCHAR      LITERALS =      18  CODE =     100  DATA =      58
       INSTRUCTIONS =     24 (LESS    0 WORDS OF DEAD CODE REMOVED)
         CINT       LITERALS =      16  CODE =      90  DATA =      44
       INSTRUCTIONS =     60 (LESS    0 WORDS OF DEAD CODE REMOVED)
         DIGIO      LITERALS =      56  CODE =     238  DATA =     144
```

**Figure 9-1. A Source Listing with WIDELIST and MAP Options Enabled**

**9.4.3 TRACEBACK OPTION.** The TRACEBACK option enables and disables the writing of traceback data in the object code. When the value of the option is TRUE (default), the compiler includes the routine name and its static nesting level with the object code for use when tracing the events involved in an error that causes termination of execution of the program. When the value of the option is FALSE, the compiler omits this data which reduces the length of each module by 6 words. The TRACEBACK option is a routine level option.

**9.4.4 ASSERTS OPTION.** The ASSERTS option enables or disables the insertion of object code corresponding to ASSERT statements into object files. When the value of the option is TRUE (default) the compiler supplies code to implement any ASSERT statements in the program. When the value of the option is FALSE the compiler ignores ASSERT statements. The ASSERTS option is a statement level option.

## 9.5 RUNTIME CHECK OPTIONS

Runtime check options control the insertion of code in object files to enable checks of program execution at runtime. The options are described in the following paragraphs.

**9.5.1 CKINDEX OPTION.** The CKINDEX option enables or disables checking of indexes of arrays. When the value of the option is TRUE, the object code includes code that checks that array indexes are within the specified range at execution time. When the value of the option is FALSE (default), array indexes are not checked at runtime. The CKINDEX option is a statement level option.

**9.5.2 CKOVER OPTION.** The CKOVER option enables or disables checking for overflow when evaluating INTEGER, LONGINT, DECIMAL, and FIXED type expressions. When the value of the option is TRUE, the object code includes code that checks for overflow at execution time. When the value of the option is FALSE (default), overflow is not checked at runtime. The CKOVER option is a statement level option.

**9.5.3 CKPREC OPTION.** The CKPREC option enables or disables checking for loss of most significant precision during conversion of DECIMAL and FIXED types. When the value of the option is TRUE, the object code includes code that checks for loss of the most significant digit during conversion at execution time. When the value of the option is FALSE (default), the results of conversion are not checked at runtime. The CKPREC option is a statement level option.

**9.5.4 CKPTR OPTION.** The CKPTR option enables or disables checking for NIL values of variables of pointer type at execution time. When the value of the option is TRUE, the object code includes code that checks for a pointer value of NIL when a pointer is used to access data at execution time. When the value of the option is FALSE (default), no check of pointer values is made at runtime. The CKPTR option is a statement level option.

**9.5.5 CKSET OPTION.** The CKSET option enables or disables checking of set element expressions. When the value of the option is TRUE, the object code includes code that checks that a set element is within the range of the base type at execution time. When the value of the option is FALSE (default), no check of set elements is made at runtime. The CKSET option is a statement level option.

**9.5.6 CKSUB OPTION.** The CKSUB option enables or disables the checking of subrange assignments and results of PRED and SUCC functions. When the value of the option is TRUE, the object code includes code that checks that values are within bounds following assignments or execution of the PRED or SUCC function at execution time. When the value of the option is FALSE (default), no check of assignments or results of these functions are made at runtime. The CKSUB option is a statement level option.

**9.5.7 CKTAG OPTION.** The CKTAG option enables or disables the checking of the tag fields of record variants. When the value of the option is TRUE, the object code includes code that checks that references to variant parts are consistent with the values of their tag fields at execution time. When the value of the option is FALSE, no checks on variant part references and tag fields are made. The CKTAG option is a statement level option.

**9.5.8 PROBER OPTION.** The PROBER option enables or disables a summary of the number of executions of each routine and of the main block of the program. When the value of the option is TRUE, the object code includes calls to the performance probe handler at entry and exit of each routine at execution time. When the value of the option is FALSE (default), the program does not call the performance probe handler. The PROBER option is a routine level option.

When the PROBER option is set to TRUE for one or more routines or for the entire program, the linked object must include library members PRB$INIT, PRB$TERM, and PRB$PERF. Refer to linking information for the appropriate operating system in an appendix.

The performance probe handler maintains a count of the number of executions of each routine and of the main program (when option PROBER is TRUE throughout the scope of the program). At the completion of execution of the program, the performance probe handler writes a display to textfile OUTPUT as shown in figure 9-2. The display lists the number of executions of each routine and of the main program.

**9.5.9 PROBES OPTION.** The PROBES option enables or disables a summary of usage of the paths of each control structure. When the value of the option is TRUE, the object code includes a call to the completeness probe handler at each path of a CASE, FOR, IF, REPEAT, and WHILE statement. When the value of the option is FALSE (default), the program does not call the completeness probe handler. The PROBES option is a routine level option.

When the PROBES option is set to TRUE for one or more routines or for the entire program, the compiler writes a probes table for each routine in the source listing. Figure 9-3 shows a source listing for which the PROBES option is true. The probes table includes a line for each call to the completeness probe handler, showing a probe number, the source line number to which the probe applies, and the probe type. The probe type identifies the control structure and the path to which the probe applies. Note that an ELSE path is identified for each IF statement whether or not the ELSE clause appears in the IF statement. This path is taken when the expression of the IF statement is not true.

```
          P E R F O R M A N C E    P R O B E    D A T A

           ROUTINE NAME          NUMBER OF EXECUTIONS

           CCHAR                   1
           CINT                    5
           DIGIO                   1
          C O M P L E T E N E S S    P R O B E    D A T A

    ROUTINE NAME    #PROBES     %ACTIVATED     INACTIVE PROBES

     CCHAR             4             75           3
     CINT              2            100
     DIGIO             2            100
        TOTAL NUMBER OF PROBES = 8
        TOTAL NUMBER OF ACTIVATED PROBES = 7
        % OF PROBES ACTIVATED = 87
```

**Figure 9-2. Execution Time Displays for PROBER and PROBES Options**

```
TI PASCAL COMPILER 1.3    DATE = 78. 25      TIME = 17: 5:44

(*$ PROBER,PROBES *)
PROGRAM DIGIO;
TYPE CHBUF = ARRAY (.1..6.) OF CHAR;
VAR  BUFF     : CHBUF;
     I,NUM    : INTEGER;
PROCEDURE CCHAR(BUFF:CHBUF;VAR NUM:INTEGER;I:INTEGER);
BEGIN   (* CCHAR *)
  NUM:= 0;
  FOR J:= 1 TO I DO
    IF BUFF(.J.)>='0' AND BUFF(.J.)<='9' THEN
      NUM:=NUM*10+ORD(BUFF(.J.))-ORD('0')
END;   (* CCHAR  *)
      PROBES FOR  CCHAR
PROBE NUMBER   LINE NUMBER   PROBE KIND
      1              3       FOR LOOP
      2              4       IF THEN
      3              6       IF ELSE
      4              6       FOR END
PROCEDURE CINT (NUM:INTEGER);
VAR  I       :INTEGER;
BEGIN   (* CINT *)
  I:=NUM DIV 10;
  IF I <> 0 THEN CINT(I);
  WRITE (CHR(NUM MOD 10 + ORD('0')))
END; (* CINT *)
      PROBES FOR  CINT
PROBE NUMBER   LINE NUMBER   PROBE KIND
      1              3       IF THEN
      2              4       IF ELSE
BEGIN   (* DIGIO *)
  WRITELN('ENTER 1 TO 5 DIGITS');
  RESET(INPUT);
  I:= 1;
  WHILE NOT EOLN DO BEGIN (* INPUT CHARS *)
     READ(BUFF(.I.));
     I:=I+1;
  END; (* INPUT CHARS *)
  I:=I-1;
  CCHAR(BUFF,NUM,I);
  NUM:=NUM+25;
  CINT(NUM);
  WRITELN
END.  (* DIGIO *)
      PROBES FOR  DIGIO
PROBE NUMBER   LINE NUMBER   PROBE KIND
      1              5       WHILE LOOP
      2              8       WHILE END


MAXIMUM NUMBER OF IDENTIFIERS USED = 13
INSTRUCTIONS =    52 (LESS   0 WORDS OF DEAD CODE REMOVED)
  CCHAR      LITERALS =     24  CODE =    174  DATA =     58
INSTRUCTIONS =    38 (LESS   0 WORDS OF DEAD CODE REMOVED)
  CINT       LITERALS =     20  CODE =    146  DATA =     44
INSTRUCTIONS =    74 (LESS   0 WORDS OF DEAD CODE REMOVED)
  DIGIO      LITERALS =     60  CODE =    296  DATA =    144
```

**Figure 9-3. Compiler Source Listing - PROBER and PROBES Options Enabled**

When the PROBES option is set to TRUE for one or more routines or for the entire program, the linked object must include library members PRB$INIT, PRB$TERM, and PRB$COMP. Refer to linking information for the appropriate operating system in an appendix.

The completeness probe handler monitors the execution of the program, making note of each path of a CASE, FOR, IF, REPEAT, and WHILE statement that is taken within those routines for which PROBES is TRUE. When execution of the program completes, the completeness probe handler writes a display to textfile OUTPUT as shown in figure 9-2. The display lists the number of probes, the percentage of usage of the probes, and the probe numbers of inactive (unused) paths for each routine and for the main program. The display also shows the total number of probes, the number activated (paths taken), and the percentage of usage of the probes.

## 9.6 MISCELLANEOUS OPTIONS

Miscellaneous options include the remaining options that are not readily categorized. The options are described in the following paragraphs.

**9.6.1 72COL OPTION.** The 72COL option enables or disables the 72 character limit for source program lines. When the value of the option is TRUE (default), only the first 72 characters of the source line are scanned by the compiler. When the value of the option is FALSE, all characters of the source line are scanned by the compiler. The 72COL option is a statement level option.

**9.6.2 FORINDEX OPTION.** The FORINDEX option enables or disables the issuing of warning messages when names of FOR control variables are identical to names of other accessible variables. When the value of the option is TRUE the compiler issues a warning message if the name of a FOR control variable is identical to the name of an accessible variable. When the value of the option is FALSE (default), the names of FOR control variables are not checked. The FORINDEX option is a statement level option.

**9.6.3 GLOBALS OPTION.** The GLOBALS option enables or disables a limitation of the use of global variables. When the value of the option is TRUE only global variables that are named in an ACCESS declaration are accessible within a routine. When the value of the option is FALSE (default), the rules of scope of variables alone determine the accessibility of global variables other than COMMON variables. The GLOBALS option is a routine level option.

**9.6.4 ROUND OPTION.** The ROUND option enables or disables the rounding of results of type DECIMAL. When the value of the option is TRUE (default), results of type DECIMAL are rounded to the nearest value of the specified precision. When the value of the option is FALSE, results of type DECIMAL are truncated to the specified precision. The ROUND option is a statement level option.

**9.6.5 UNSAFEOPT OPTION.** The UNSAFEOPT option enables or disables optimization that may only be performed correctly if all variables are disjoint (variables accessible by only one variable name). The UNSAFEOPT option allows the compiler to retain more than one variable in the registers at a time, which results in a significant reduction in the size of the object code for some programs. The UNSAFEOPT should not be specified for a program in which a variable is accessed by more than one name. This can occur when a routine has two parameters called by reference and the call specifies the same variable for both parameters. This can also occur from the use of the LOCATION function to access a variable.

When the value of the option is TRUE, the optimization is enabled. When the value of the option is FALSE (default), the optimization is disabled. The UNSAFEOPT option is a routine level option.

# SECTION X

# FORMATTING SOURCE CODE

## 10.1 GENERAL

This section includes guidelines to follow when coding a source program, describes NESTER, a utility that assists in preparation of source code, and includes instructions for operating NESTER.

## 10.2 CODING

Typically, the Model 990 Computer TIP source program is written on a disk file using the Text Editor. The input device is usually the Model 911 VDT, which supports the complete character set of TIP shown in paragraph 3.2. The characters for each line of code may be entered starting at any character position, but must not extend beyond character position 72 if 72COL option is true.

As described in paragraph 3.2, there are alternate character combinations and an alternate character defined to use when the complete TIP character set is not available. The user may use the alternate characters when the source code has to be transportable to an environment that does not support the complete character set.

The user may use either upper or lower case letters in the declarations and statements of the source program. The TIP compiler translates lower case letters to upper case letters before processing the code, and therefore sees an identifier entered in lower case letters as identical to the same identifier entered in upper case letters.

## 10.3 SOURCE FORMATTER

The source formatter (NESTER) restructures a source module to provide indentation consistent with the logical structure. As described in paragraph 8.6, a Pascal program is precisely structured. The format of source lines allows indentations of the source lines that can show the program structure more effectively. However, entry of the source lines in accordance with the structure can be tedious, particularly with a VDT. NESTER accepts source code lines however they may be indented, or without indentation, and indents the source lines in accordance with the logical structure of the program.

In addition, NESTER inserts a sequence number in character positions 73 through 80 of each line. The sequence numbers begin at zero and are incremented by 10.

**10.3.1 NESTER FUNCTIONS.** NESTER uses five parameters in establishing the source program format that corresponds to the logical organization of the program. These parameters and their default values, along with four other options, are listed in table 10-1. Two of these parameters, DTAB and CCOL, apply to the declarations of the program. The rules that apply to each of the declarations are:

- The PROGRAM declaration begins in the leftmost character position (column 1).

  Example:

  ```
  PROGRAM RANDOM;
  ```

- The LABEL declaration begins in the leftmost character position (column 1) and the label numbers follow on the same line.

  Example:

  ```
  LABEL 10,20,30;
  ```

- The CONST declaration begins in the leftmost character position (column 1) and constants are tabulated into columns separated by the value of the Constant Column parameter CCOL.

  Example:

  ```
  CONST A = 2;          B = 3;
    C = 4;              D = 5;
    E = 6;              F = 7;
  ```

- The TYPE declaration begins in the leftmost character position (column 1). The first type declaration follows on the same line.

  Example:

  ```
  TYPE A = ARRAY (.0..10.) OF INTEGER;
  ```

- Any subsequent type declarations are indented by the value of the Declaration Tab parameter DTAB.

  Example:

  ```
  TYPE A = ARRAY (.0..10.) OF INTEGER;
    B = (C,D,E);
  ```

- In a record declaration, the first field identifier follows the declaration; subsequent field identifiers are indented by the value of DTAB on a new line.

  Example:

  ```
  TYPE A = ARRAY (.0..10.) OF INTEGER;
    B = (C,D,E);
    REC = RECORD A:INTEGER;
      B:REAL;
      CASE C:BOOLEAN OF
        FALSE: (D:INTEGER;
          E: INTEGER);
      END;
  ```

- When the component type of an array is a structured type, the structured type is indented by the value of DTAB on a new line.

  Example:

  ```
  TYPE A = ARRAY (.0..10.) OF INTEGER;
    B = (C,D,E);
    REC = RECORD A:INTEGER;
      B:REAL;
      CASE C:BOOLEAN OF
        FALSE: (D:INTEGER;
          E: INTEGER);
      END:
    MATRIX = ARRAY (.0..9.) OF
      ARRAY (.0..9.) OF
        ARRAY (.0..2.) OF INTEGER;
  ```

- The VAR declaration begins in the leftmost character position (column 1). The first variable declaration follows on the same line. Subsequent variable declarations are indented by the value of DTAB and each starts a new line.

  Example:

  ```
  VAR ARY : A;
    I,J,K,L:INTEGER;
    R:REC;
  ```

- The COMMON declaration begins in the leftmost character position (column 1). The first common declaration follows on the same line. Subsequent variable declarations are indented by the value of DTAB and each starts a new line.

  Example:

  ```
  COMMON R:BOOLEAN;
    I: INTEGER;
    X: REAL;
    M: ARRAY 1..5 OF INTEGER;
  ```

- The ACCESS declaration begins in the leftmost character position (column 1). The common variables follow on the same line.

  Example:

  ```
  ACCESS COM1,COM2;
  ```

- The routine declarations begin with the keyword PROCEDURE or FUNCTION in the leftmost character position (column 1). The parameter list follows on the same line.

  Example:

  ```
  PROCEDURE A(B,C:INTEGER);
  ```

Table 10-1. NESTER Options

| Option Keyword | Meaning | Default |
|---|---|---|
| DTAB | Declaration tab value | 4 |
| CCOL | Constant column increment | 20 |
| STAB | Statement tab value | 2 |
| SCOL | Statement column increment | 1 |
| SLIM | Statement column limit | 30 |
| ADJT | Adjust comments to right margin | TRUE |
| FILL | Allows concatenation of source lines in accorance with options in effect. When FALSE, items on a new line of input are placed on a new line in the output. | TRUE |
| CONV | Convert characters in source statements to those of a transportable character set. | FALSE |
| WIDE | Accept 80-column source statements. | FALSE |

In addition to the rules for the declarations, there is a rule with respect to the breaking of a line in the declaration section. A line starts with the declaration keyword or an identifier and is usually separated from the next line with a semicolon (and the end-of-line indication appropriate to the input device). When NESTER breaks a line, the continuation on the succeeding line is indented by the value of DTAB.

Two other parameters, STAB and SCOL, apply specifically to the statement portions of the program, i.e., the compound statements that contain the statements that specify the processing of the program. The rules that apply to the statement portions are as follows:

- The BEGIN and END statements of the compound statement that contains the statements of the block begin in the leftmost character position (column 1).
- The component statements within the compound statement are indented by the value of STAB.
- Simple statements (assignment, PROCEDURE, ESCAPE, GOTO, and ASSERT) are placed on one line, tabulated into columns separated by the value of SCOL.
- A statement is placed on a new line if it is too long to fit on the current line.
- When a statement is too long for a single line, the continuation line is indented by the value of STAB.
- Structured statements other than compound statements (IF, CASE, FOR, WHILE, REPEAT, and WITH) always begin on a new line.
- Component statements within a structured statement are indented by the value of STAB.

- Keywords BEGIN and END do not cause indentation and normally do not begin a new line.
- Statement labels and escape labels start in character position two and may force the labeled statement to start a new line.
- The ELSE portion of an IF statement starts a new line and is positioned at the same character position as the matching IF.
- The UNTIL portion of a REPEAT statement starts a new line and is positioned at the same character position as the matching REPEAT.
- Each CASE label is indented by the value of STAB.
- Each CASE alternative statement (following keyword OTHERWISE) is indented by the value of STAB beyond the character position at which the keyword OTHERWISE begins.

Option ADJT enables or disables the adjustment of comments. When the value of ADJT is true, (default) comments that are less than 70 character in length are right justified on the line on which they are entered. Comments that are more than 69 characters in length are positioned to begin in character position one. Comments that cross line boundaries are not moved. When the value of ADJT is false, comments are not moved.

The column limit value SLIM applies to both declarations and statements that are arranged in columns. Specifically, these are constants and simple statements. No constant or statement is started beyond the column limit. For example, the default value of 30 allows only two columns of constants in a CONST declaration, and only as many simple statements as can be placed on a line without starting a statement beyond character position 30.

The FILL option enables or disables the concatenation of characters from two or more lines on one line. When the value of the option is true (default), constants in a CONST declaration may be arranged in columns and simple statements may be concatenated on one line using the value of the CCOL and SCOL options, respectively. When the value of the option is false, constants and simple statements remain on the line on which they are entered.

The CONV option converts the characters in source statements that are not transportable to characters that are more generally available. Specifically, lower case letters are converted to upper case characters, brackets ([ ]) are replaced with equivalent character combinations ((. and .)), and braces ({ }) are replaced with character combinations ((* and *)).

The WIDE option accepts 80-column source statements and reformats the source code to 72-column output. When the input source statement contains a string constant longer than 72 characters, NESTER issues an error message.

Figure 10-1 shows the effect of the default options on the statement portion of a routine.

```
BEGIN I:=J;                                        (* BEGINNING OF BODY *)
  IF I>K THEN                                                   (* IF *)
    WITH R DO BEGIN                                           (* WITH *)
      B:=1.0; C:=FALSE; D:=K;
      E:=J; END                                           (* WITH END *)
  ELSE                                                        (* ELSE *)
    CASE R.B OF                                               (* CASE *)
      FALSE: FOR I:=1 TO 10 DO ARY(.I.):=I;
      OTHERWISE BEGIN I:=0;
        REPEAT ARY(.I.):=0; I:=I+1;
        UNTIL I>=10;
        END;
      END;                                                (* CASE END *)
END;                                                   (* END OF BODY *)
```

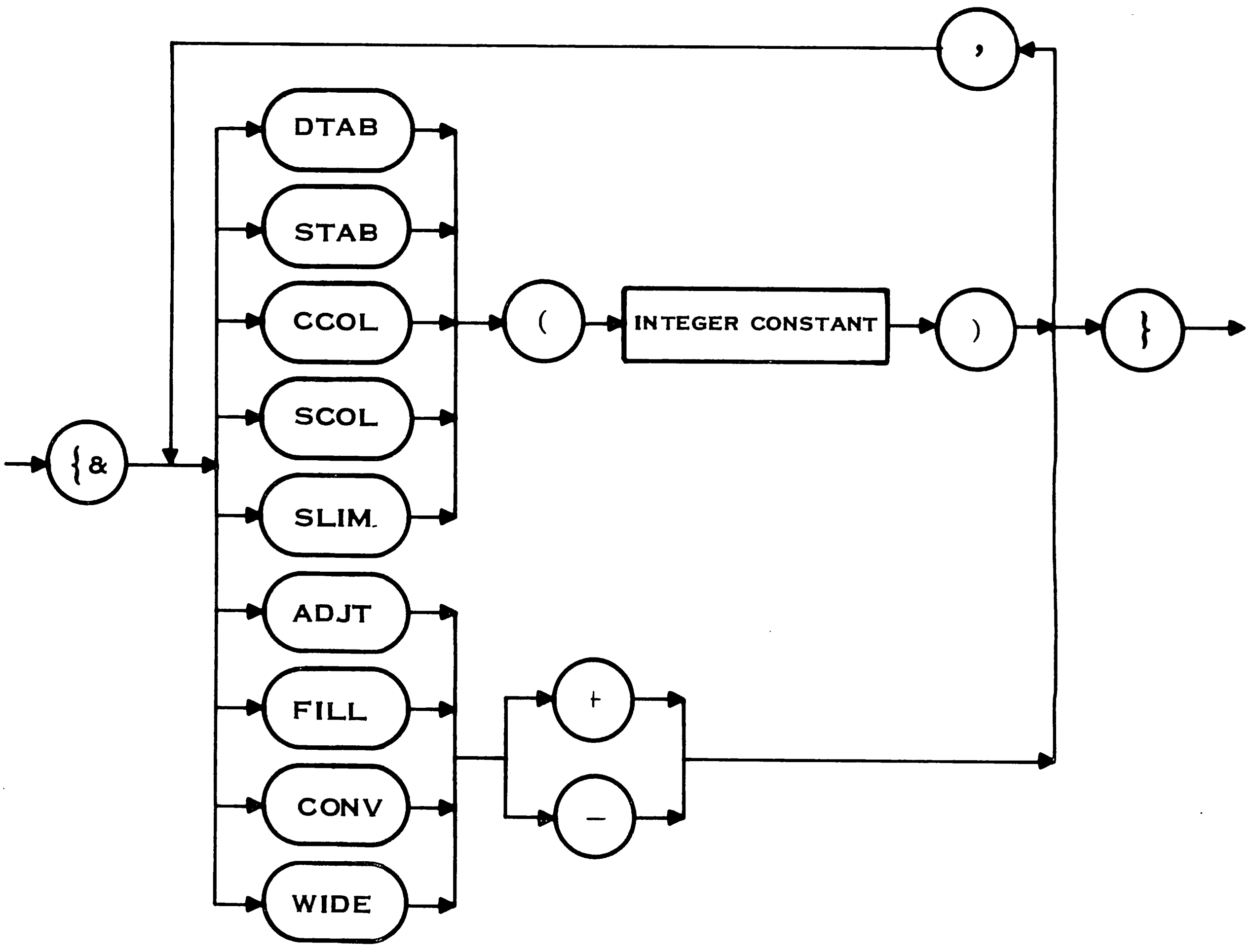
**Figure 10-1. NESTER Statement Example**

**10.3.2 NESTER OPTION COMMENT.** The values of the NESTER options are changed by entering option comments. The value supplied in an option comment continues to apply until another value is supplied in a subsequent option comment. The option comments become a part of the source program and are processed by the compiler as comments; i.e., they have no effect on compiler options. The syntax of an option comment is as follows:

```
<option comment> ::= "{ "&<option>[,<option>]" }"
<option> ::= <integer-valued option>(integer constant) |
             <Boolean-valued option><plus or minus>
<integer-valued option> ::= DTAB|STAB|CCOL|SCOL|SLIM
<Boolean-valued option> ::= ADJT|FILL|CONV|WIDE
<plus or minus> ::= +|-
```

The syntax diagram is as follows:

Option comment:



No blanks are allowed in the option comment. The option names may be entered in any sequence.

Examples:

{&DTAB(2),STAB(5),FILL-,SLIM(72)}
{&DTAB(4),CCOL(20),STAB(2),SCOL(1),SLIM(30),ADJT+,FILL+ }

The first example changes the value of DTAB to 2 and the value of STAB to 5. It sets the value of the FILL option to false, and changes the value of SLIM to 72. This value of SLIM allows maximum packing of constants on a single line, and the maximum packing of simple statements on a single line. However, by setting FILL to false, the packing allowed by the value of SLIM would not occur.

The second example restores the option values to the default values.

**10.3.3 INTERACTIVE MODE.** In the preceding description it was assumed that NESTER reformats a single source program in each execution. This mode of using NESTER is called the batch mode. NESTER may be used in an interactive mode in which the user specifies more than one source file (program) to be reformatted, and separate files into which the resulting source modules are to be written. The interactive mode also allows the user to specify global options that apply to all source files except as overridden by option comments within the source code.

NESTER uses five files as follows:

| | |
|---|---|
| OUTPUT | Commands executed, and error messages |
| SYSMSG | System messages |
| INPUT | Input commands |
| NESTSRC | Source module to be reformatted |
| NESTOUT | Reformatted source module |

In the batch mode, NESTER requires that the NESTSRC and NESTOUT files be specified. The INPUT file should be empty or should contain information other than valid NESTER commands (to be described in a subsequent paragraph). When the input file is specified as a VDT, an empty file results if the ENTER key (Model 911 VDT) is pressed instead of entering data when NESTER requests input from the user.

For the interactive mode, NESTSRC and NESTOUT must be pathnames of libraries, or must be omitted. The library member names of source files and of nested output files, or synonyms of the pathnames of source and nested source files are supplied in commands.

**10.3.4 NESTER COMMANDS.** NESTER commands specify library member names or file synonyms for source files and nested output files to be processed by NESTER, and global options for the run. NESTER commands are input from the INPUT file. The syntax for the NEST command is as follows:

```
<nest command>::= *NEST<source><destination>;

<source>::= [<library synonym>] (<member>) | <file synonym>

<destination>::=[<library synonym>] (<member>) | <file synonym>
```

When the library synonym for the source is omitted, the library synonym entered as NESTSRC is used by default. When the library synonym for the destination is omitted and a library synonym for the source was entered, the library synonym for the source is also used for the destination. When neither the library synonym for the source nor the library synonym for the destination is entered, the library synonym entered as NESTOUT is used by default. When the operands of the command are not file synonyms, library synonyms, or names of members of libraries named as NESTSRC or NESTOUT, NESTER terminates with a normal completion message without writing nested output files. A *NEST command is entered for each source file to be reformatted.

The syntax for the *OPTIONS command is as follows:

```
<option command>::= *OPTIONS <option> { ,<option>} ;
```

The option is of the same form specified for the option comment in paragraph 10.3.2. Options specified in option commands are global; they may be overridden by local options specified in option comments within the source file. A global option continues to apply until a new value for the option is supplied in a subsequent option command.

**10.3.5 EXECUTING NESTER.** TIP software includes an SCI procedure XNESTER for executing the source formatter, NESTER. Enter XNESTER at any time DX10 requests a command. DX10 requests the following information:

```
          SOURCE
   NESTED SOURCE
        COMMANDS
COMMAND LISTINGS
        MESSAGES
            MODE
```

The first five items require access names of devices or files, as follows:

- SOURCE — The access name of a file (batch mode) or of a library (interactive mode) that contains source code to be formatted.
- NESTED SOURCE — The access name of a file (batch mode) or of a library (interactive mode) for the formatted source code.
- COMMANDS — The access name of a device or file for input commands. (Leave blank for batch mode.)
- COMMAND LISTING — The access name of a device for listing commands and errors (should be ME when commands are entered at terminal).
- MESSAGES — The access name of a device or file for system messages.

The MODE item is either FOREGROUND or BACKGROUND. When NESTER is executed in the background mode, the terminal is available for entry of other commands or for foreground execution of another program. The foreground mode is the default mode. When NESTER is executed in the foreground mode, the terminal may not be used by any program other than NESTER until NESTER terminates execution.

NESTER uses five files as follows:

| | |
|---|---|
| OUTPUT | Commands executed, and error messages |
| SYSMSG | System messages |
| INPUT | Input commands |
| NESTSRC | Source module to be reformatted |
| NESTOUT | Reformatted source module |

In the batch mode, NESTER requires that the NESTSRC and NESTOUT files be specified. The INPUT file should be empty or should contain information other than valid NESTER commands (described in paragraph 10.3.4). When the input file is specified as a VDT, an empty file results if the ENTER key (Model 911 VDT) is pressed instead of entering data when NESTER requests input from the user.

For the interactive mode, NESTSRC and NESTOUT must be pathnames of libraries, or must be omitted. The library member names of source files and of nested output files, or synonyms of the pathnames of source and nested source files, are supplied in commands.

**10.3.6 NESTER ERROR MESSAGES.** NESTER checks the statements in the source file for simple syntax errors and prints error numbers when errors are found. The nested source code resulting from nesting source statements that contain these errors is usually incorrectly nested. For this reason, even though it is allowed to specify the same file or member for both source file and nested output file, it should not be done. If the file contains errors, the source is replaced by incorrectly nested source code.

The error messages corresponding to the error numbers are listed in table 10-2. Most of the error numbers agree with the error number of the corresponding error used by the compiler, and with those of standard Pascal.

**10.3.7 NESTER EXAMPLE.** Figure 10-2 shows the contents of the nested output file written by NESTER using default option values.

**Table 10-2. NESTER Errors**

| Number | Message |
|---|---|
| 2 | Identifier expected. |
| 4 | ) expected. |
| 5 | : expected. |
| 6 | Illegal symbol. |
| 8 | Of expected. |
| 9 | ( expected. |
| 10 | Error in type. |
| 11 | [ expected. |
| 12 | ] expected. |
| 13 | End expected. |
| 14 | ; expected. |
| 15 | Integer expected. |
| 16 | = expected. |
| 17 | Begin expected. |
| 50 | Error in constant. |
| 51 | := expected. |
| 52 | Then expected. |
| 53 | Until expected. |
| 54 | Do expected. |
| 55 | To/downto expected. |
| 58 | Error in factor. |
| 106 | Number expected. |
| 201 | Error in real constant; digit expected. |
| 202 | String constant too long or crosses a card boundary. |
| 255 | Too many errors on this source line. |

```
PROGRAM LABELS;
(*-----------------------------------------------------------------------
   PROGRAM LABELS:
   PURPOSE :    THIS PROGRAM READS AN ADDRESS LABEL AND PRINTS MULTIPLE
                COPIES OF THAT LABEL.
   FILES USED : INPUT  - FOR USER-SUPPLIED PARAMETERS AND THE LABEL
                CRTFIL - USED FOR PROMPTING INPUT
                OUTPUT - MULTIPLE COPIES OF THE LABEL
   PROCEDURES CALLED : INTERACT, READANDPRINT
----------------------------------------------------------------------*)
VAR CRTFIL : TEXT ;                         (*USED TO PROMPT INPUT*)
    CHARSPERLINE : INTEGER;         (*NUMBER OF CHARACTERS PER LINE*)
    LINESPERLABEL : INTEGER;            (*NUMBER OF LINES PER LABEL*)
    COPYCOUNT : INTEGER;                (*NUMBER OF COPIES TO PRINT*)
PROCEDURE INTERACT;
(*-----------------------------------------------------------------------
  PROCEDURE INTERACT:
  PURPOSE : INTERACT PROMPTS THE USER, REQUESTING CERTAIN INPUTS.
  OUTPUTS : CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
            LINESPERLABEL - NUMBER OF LINES PER LABEL
            COPYCOUNT - NUMBER OF LABELS TO PRINT
----------------------------------------------------------------------*)
BEGIN                                                    (*INTERACT*)
  REWRITE( CRTFIL );
  WRITELN( CRTFIL, 'HOW MANY CHARACTERS PER LINE?' );
  RESET(INPUT); READ( CHARSPERLINE );
  WRITELN( CRTFIL, 'HOW MANY LINES PER LABEL?' );
  READLN; READ( LINESPERLABEL );
  WRITELN( CRTFIL, 'HOW MANY LABELS?' );
  READLN; READ( COPYCOUNT ); WRITELN(CRTFIL, 'NOW INPUT THE LABEL');
END;                                                     (*INTERACT*)
PROCEDURE READANDPRINT;
(*-----------------------------------------------------------------------
  PROCEDURE READANDPRINT;
  PURPOSE : READANDPRINT READS A LABEL AND PRINTS MULTIPLE COPIES OF IT.
  PROCEDURES CALLED : GETLINE, PRINTLABEL
----------------------------------------------------------------------*)
TYPE
    LINE = PACKED ARRAY (.1..CHARSPERLINE.) OF CHAR;
VAR
    LABELIMAGE : ARRAY (.1..LINESPERLABEL.) OF LINE;
PROCEDURE GETLINE(VAR THISLINE : LINE);
(*-----------------------------------------------------------------------
  PROCEDURE GETLINE;
  PURPOSE : GETLINE READS A SINGLE LINE OF A LABEL.
  INPUTS :  CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
  OUTPUTS : THISLINE - THE LINE THAT WAS READ.
----------------------------------------------------------------------*)
VAR CH : INTEGER;
BEGIN                                                     (*GETLINE*)
  READLN; CH := 1;
  WHILE CH <= CHARSPERLINE AND NOT EOLN(INPUT) DO BEGIN
    READ( THISLINE(.CH.) ); CH := CH + 1;
    END;                       (*FILL IN REST OF LINE WITH BLANKS*)
```

Figure 10-2. NESTER Example (Sheet 1 of 2)

```
  FOR J := CH TO CHARSPERLINE DO THISLINE(.J.) := ' ';
END;                                                            (*GETLINE*)
PROCEDURE PRINTLABEL;
(*-----------------------------------------------------------------------------
  PROCEDURE PRINTLABEL;
  PURPOSE : PRINTLABEL PRINTS ONE COPY OF THE LABEL.
  INPUTS :  LINESPERLABEL - NUMBER OF LINES PER LABEL
            CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
            LABELIMAGE - THE LABEL TO BE PRINTED
-----------------------------------------------------------------------------*)
BEGIN                                                        (*PRINTLABEL*)
  FOR L := 1 TO LINESPERLABEL DO BEGIN
    FOR CH := 1 TO CHARSPERLINE DO WRITE( LABELIMAGE(.L.)(.CH.) );
    WRITELN; END;
END;                                                         (*PRINTLABEL*)
BEGIN                                                      (*READANDPRINT*)
  FOR L := 1 TO LINESPERLABEL DO GETLINE( LABELIMAGE(.L.) );
  FOR K := 1 TO COPYCOUNT DO PRINTLABEL;
END;                                                       (*READANDPRINT*)
BEGIN                                                            (*LABELS*)
  INTERACT; READANDPRINT;
END.                                                             (*LABELS*)
```

**Figure 10-2. NESTER Example (Sheet 2 of 2)**

# SECTION XI

# THE PASCAL COMPILER

## 11.1 GENERAL
The Pascal compiler consists of three separate tasks: SILT1, the first of two phases of the Source to Intermediate Language Translator; SILT2, the second phase; and CODEGEN, the final phase of compilation, which translates the intermediate language into object code. Table 11-1 lists the files used by the three tasks. The following descriptions of the tasks refer to the files.

## 11.2 SILT1
SILT1 parses the source statements and translates them into an intermediate representation that is used as input to SILT2. SILT1 reads the source program from file INPUT and writes the intermediate (token) representation on file TOKENS. SILT1 detects syntax errors and writes information about any errors detected on file ERRFILE to be included in the listing written by SILT2. SILT1 also writes the first eight characters of the name of each routine on file SYSMSG for display.

## 11.3 SILT2
SILT2 transforms the intermediate representation on file TOKENS into an intermediate language, which it writes on file CILFIL as input to CODEGEN. SILT2 also writes a descriptor file, DESCFIL, for input to CODEGEN. SILT2 detects semantic errors in the source program, and writes a source listing. Any errors written to ERRFILE by SILT1 are combined with any errors detected by SILT2 and the source code from file INPUT in writing the listing on file OUTPUT. SILT2 writes the first eight characters of the name of each routine on file SYSMSG (similar to SILT1) and writes an error message when any errors have been detected by either SILT1 or SILT2.

## 11.4 CODEGEN
CODEGEN transforms the intermediate language on CILFIL into object code using the descriptors on DESCFIL, and writes the object code to a sequential file, OBJECT. CODEGEN writes to an internal temporary file TEMPFIL from which it retrieves the data when required. CODEGEN completes the listing on file OUTPUT by writing the number of instructions, the amount of dead code removed, and storage requirements for each module (main program and each routine). CODEGEN also writes the first eight characters of each module name on file SYSMSG as do SILT1 and SILT2.

Several figures throughout the manual show source listing examples. Figure 11-1 is an example of a source listing using the default options of the compiler. Figure 11-2 is an example of the display of SYSMSG written during a compilation.

**11.5 EXECUTING THE COMPILER.** A System Command Interpreter (SCI) procedure, XTIP, is used to execute the compiler. The user enters the procedure name XTIP at any time DX10 requests a command. DX10 requests the following information:

```
     SOURCE
     OBJECT
    LISTING
   MESSAGES
       MEM1
       MEM2
       MEM3
       MODE
```

The first four items require access names of devices or files, as follows:

- SOURCE — The access name of the TIP source file.
- OBJECT — The access name of a file to which object code is written.
- LISTING — The access name of a device or file for the source listing.
- MESSAGES — The access name of a device or file for system messages.

The next three items request memory space for stack and heap for the compiler. Each item is an ordered pair; the first entry is a number of bytes for stack and is followed by a comma; the second is a number of bytes for heap. The number of bytes in each case specifies multiples of 1024 (K) bytes. The memory space items are as follows:

- MEM1 — Stack and heap required for SILT1. Default is 6,10.
- MEM2 — Stack and heap required for SILT2. Default is 13,4.
- MEM3 — Stack and heap required for CODEGEN. Default is 10,8.

The last item is the execution mode, FOREGROUND or BACKGROUND. When the compiler is executed in the background mode, the terminal is available for entry of other commands or for foreground execution of another program. The default mode is the background mode. When the compiler is executed in the foreground mode, the terminal may not be used by any program other than the compiler until the compiler terminates execution.

When either SILT1 or SILT2 detects an error, the message file written by SILT2 contains a message indicating that errors have been detected, and CODEGEN does not execute. The listing shows the errors.

Each phase of the compiler sets a condition code, assigning the code as the value of synonym $$CC. The code is zero for normal termination and a nonzero value for abnormal termination. Specifically, values of $$CC following execution of the compiler have the following significance:

| | |
|---|---|
| 0000 | No errors or warnings |
| $4000_{16}$ | Warnings issued |
| $6000_{16}$ | Nonfatal errors detected |
| $8000_{16}$ | Fatal errors detected |
| $C000_{16}$ | Abnormal termination - compiler terminated with runtime error |

Condition code synonym $$CC may be tested in a batch stream to alter the execution of the batch stream when an error is detected. The value placed in $$CC is only valid following the execution of the compiler (prior to execution of DX10 utilities that set $$CC). Its value may be stored in another synonym by setting that synonym to the value of @$$CC with an .SYN SCI primitive. An attempt to determine the value of $$CC by executing an LS command always shows the value of $$CC as zero because the LS command utility sets $$CC to zero.

## 11.6 ERROR HANDLING

Syntax errors are detected by SILT1, and semantic errors are detected by SILT2. At the time that SILT2 writes the listing, all error checking has been performed and ERRFILE contains information with respect to errors detected by SILT1. SILT2 includes error messages in the listing when appropriate and writes an indication that errors have been detected on file SYSMSG.

```
TI PASCAL COMPILER 1.3     DATE = 78. 10        TIME = 14:16:33




PROGRAM TRIANGLE;
VAR   X,Y,Z :INTEGER;
      TRI    :BOOLEAN;
      AREA,S:REAL;
BEGIN   (* TRIANGLE *)
WRITELN('            X            Y             Z',
       '       TRIANGLE?  AREA');
RESET(INPUT);
REPEAT
  READ(X,Y,Z);
  TRI:=TRUE;
  TRI:=X>0 AND TRI;
  TRI:=Y>0 AND TRI;
  TRI:=Z>0 AND TRI;
  IF NOT TRI THEN
    WRITELN(X,Y,Z,'       NO')
    ELSE    BEGIN   (* SIDES POSITIVE *)
      TRI:=X+Y>Z AND TRI;
      TRI:=X+Z>Y AND TRI;
      TRI:=Y+Z>X AND TRI;
      IF NOT TRI THEN
        WRITELN(X,Y,Z,'       NO')
        ELSE    BEGIN   (* SIDES OK *)
          S:=0.5*(X+Y+Z);
          AREA:=SQRT(S*(S-X)*(S-Y)*(S-Z));
          WRITELN(X,Y,Z,'       YES',AREA)
          END;  (* SIDES OK *)
      END;  (*SIDES POSITIVE*)
  UNTIL X = 0 AND Y = 0 AND Z = 0
END (* TRIANGLE *).

MAXIMUM NUMBER OF IDENTIFIERS USED = 7
INSTRUCTIONS =    232 (LESS    0 WORDS OF DEAD CODE REMOVED)
  TRIANGLE  LITERALS =      130  CODE =      920  DATA =      164
```

**Figure 11-1. Source Listing Example, Default Options**

The compiler categorizes errors as warnings, nonfatal errors, and fatal errors. A warning is a syntax error that SILT corrected, or an irregularity that may or may not actually be an error. A nonfatal error is an error that does not necessarily prevent CODEGEN from writing an object module or modules for the program. A fatal error is one that causes SILT2 to produce incorrect intermediate language; if CODEGEN executes, the resulting object module will issue a runtime error when it is executed.

```
SILT1        EXECUTION BEGINS
INTERACT
GETLINE
PRINTLAB
READANDP
LABELS
NORMAL TERMINATION
STACK USED =    4164   HEAP USED = 2130


SILT2        EXECUTION BEGINS
INTERACT
GETLINE
PRINTLAB
READANDP
LABELS
NO ERRORS IN PROGRAM
NORMAL TERMINATION
STACK USED = 11076   HEAP USED = 1876

CODEGEN      EXECUTION BEGINS
INTERACT
GETLINE
PRINTLAB
READANDP
LABELS
NORMAL TERMINATION
STACK USED = 10154   HEAP USED = 1618
```

**Figure 11-2. TIP Compiler SYSMSG File Display**

SILT2 attempts to list an error number on the line immediately following the line that contained the error. However, since certain errors such as "semicolon expected" may not be detected until the first symbol on the next line has been scanned, there are a few cases for which the error is indicated one line too late. There are other cases in which the compiler is not able to identify the actual error; subsequent code appears to be in error because of an error in preceding code. Syntax errors are shown in the form !n in which n is the error number. The exclamation point is positioned as closely as possible beneath the symbol in error. Semantic errors are shown in the form **** ERROR # n **** at the beginning of the line following the line on which the error occurs. The message cannot be positioned below the symbol because SILT2 processes an intermediate representation rather than the source code. When errors are detected, SILT2 writes additional error information at the end of the source listing. The information includes the numbers of errors in each category and a table of error numbers and corresponding error messages for each number that is printed in the listing. Most of the error numbers used by the TIP compiler have the same significance as the same numbers when used by Pascal as defined in *Pascal User Manual and Report,* K. Jensen and N. Wirth, Springer-Verlag, 1975.

**Table 11-1. Files Required by the Compiler Tasks**

| Task | File Name | I/O | File Usage |
|---|---|---|---|
| SILT1 | INPUT | I | TIP Source Program |
| | OUTPUT | O | Used as dump file in case of abnormal termination |
| | SYSMSG | O | Messages |
| | * TOKENS | O | Parse tokens (input to SILT2) |
| | * ERRFILE | O | Syntax errors (input to SILT2) |
| SILT2 | INPUT | I | TIP Source Program |
| | OUTPUT | O | Source listing with error messages, if any. |
| | SYSMSG | O | Messages |
| | * TOKENS | I | Parse tokens |
| | * ERRFILE | I | Syntax errors |
| | * CILFIL | O | Common Intermediate Language (input to CODEGEN) |
| | * DESCFIL | O | Module descriptor file (input to CODEGEN) |
| CODEGEN | OBJECT | O | Object file |
| | OUTPUT | O | Final listing information |
| | SYSMSG | O | Messages |
| | * CILFIL | I | Common Intermediate Language |
| | * DESCFIL | I | Module descriptor file |
| | * TEMPFIL | I/O | Internal temporary file |

*Files marked with an asterisk are internal temporary files.

Figure 11-3 shows a source listing of code that contains two errors. Notice that the error summary states that there are four nonfatal errors, and none of either other category. The messages corresponding to the error numbers that appear in the listing are printed at the bottom of the listing. The 25th line from the top contains the first error. The number 51 shown on the following line corresponds to the message := expected. Immediately above the exclamation point that precedes the number is an equal sign; the colon was omitted from the assignment statement. Three error numbers are shown following the last line of the program. The corresponding messages indicate that a statement, an END keyword, and a period have been omitted. Obviously the END. on the preceding line has not been recognized by the compiler. The actual error is on the second line containing the comment INPUT CHARS. An at sign (@) was entered in place of an asterisk with the parenthesis to close the comment. As a result of this error, the compiler considered the remainder of the program (through the *) at the right end of the last line) to be comment. The compiler was unable to find the normal termination of the program.

The TIP error messages are listed in Appendix E.

```
TI PASCAL COMPILER 1.3    DATE = 78. 31      TIME = 11: 4:34

PROGRAM DIGIO;
TYPE CHBUF = ARRAY (.1..6.) OF CHAR;
VAR  BUFF    : CHBUF;
     I,NUM   : INTEGER;
PROCEDURE CCHAR(BUFF:CHBUF;VAR NUM:INTEGER;I:INTEGER);
BEGIN  (* CCHAR *)
  NUM:= 0;
  FOR J:= 1 TO I DO
    IF BUFF(.J.)>='0' AND BUFF(.J.)<='9' THEN
      NUM:=NUM*10+ORD(BUFF(.J.))-ORD('0')
END;  (* CCHAR  *)
PROCEDURE CINT (NUM:INTEGER);
VAR  I       :INTEGER;
BEGIN  (* CINT *)
  I:=NUM DIV 10;
  IF I <> 0 THEN CINT(I);
  WRITE (CHR(NUM MOD 10 + ORD('0')))
END; (* CINT *)
BEGIN   (* DIGIO *)
  WRITELN('ENTER 1 TO 5 DIGITS');
  RESET(INPUT);
  I:= 1;
  WHILE NOT EOLN DO BEGIN (* INPUT CHARS *)
     READ(BUFF(.I.));
     I=I+1;
      !51
  END; (* INPUT CHARS @)
  I:=I-1;
  CCHAR(BUFF,NUM,I);
  NUM:=NUM+25;
  CINT(NUM);
  WRITELN
END.  (* DIGIO *)

,43,13,23

MAXIMUM NUMBER OF IDENTIFIERS USED = 13

NUMBER OF ERRORS -- FATAL = 0  NONFATAL = 4  WARNINGS = 0
  13  E  'END' EXPECTED
  23  E  '.' EXPECTED
  43  E  STATEMENT EXPECTED
  51  E  ':=' EXPECTED
```

**Figure 11-3. Source Listing Containing Errors**

# SECTION XII

# SEPARATE COMPILATION

## 12.1 GENERAL

Development of a large program is significantly less expensive when modules of the program can be recompiled for correction or for change without recompiling the entire program. In a block-structured language such as TIP separate compilation is more difficult than in assembly language or high level languages that are not structured. The scope rules of TIP and the capability of passing parameters either by value or by reference make this true. To separately compile a TIP routine all global declarations must be included in the source code so that the environment identical to that in which the routine executes is provided. Global declarations in this context include the declaration sections of all routines within which the routine is nested. The process of manually merging the declaration section is tedious and error-prone.

## 12.2 REQUIREMENTS FOR SEPARATE COMPILATION

The TIP compiler produces a separate object module for each program and for each routine of the program. Two object modules result from compiling the following code:

```
PROGRAM A;
VAR X,Y,Z : INTEGER;
PROCEDURE B(W:INTEGER); FORWARD;
      PROCEDURE B;
      BEGIN (* B *)
        W := Y
      END (* B *);
BEGIN (* A *)
  B (X)
END (* A *).
```

The two object modules are concatenated in a single file by the compiler; they could be separated (using the text editor, for example) and stored as separate library members. They could be reconcatenated before link editing, or could be specified to the link editor by specific INCLUDE commands in the control file.

When the main program module A needs to be recompiled, a new module for A can be compiled and the new module linked with the existing module for B. To recompile a new module for A, simply omit the code for procedure B, and compile the following:

```
PROGRAM A;
VAR X,Y,Z : INTEGER;
PROCEDURE B(W : INTEGER); FORWARD;
BEGIN (* A *)
 B(X)
END (* A *).
```

Since procedure B is omitted, the forward declaration of procedure B must be included so that the call to procedure B in program A will result in the correct linkage.

To recompile routine B correctly, the compiler must have the declarations of the main program as well as those of routine B. The source code is as follows:

```
PROGRAM A;
VAR X,Y,Z : INTEGER;
PROCEDURE B(W : INTEGER); FORWARD;
      PROCEDURE B;
      BEGIN (* A *)
       W := Y
      END (* B *);
BEGIN (*$NO OBJECT*)
END (* A *).
```

The NO OBJECT option suppresses the production of an object module for A, and only the object module for B is produced. This module may be linked with the existing module for A to obtain a new version of the entire program.

The NO OBJECT option is required because even the BEGIN END keywords alone for module A would have produced an object module. It would have been necessary to delete this module in order to properly link the existing A module with the new B module.

A different approach to the problem of separately compiling a module of a program is to store individual source modules in a library. The source code in the example could be separated as follows:

```
PROGRAM A;
VAR X,Y,Z : INTEGER;
PROCEDURE B(W : INTEGER); FORWARD;
BEGIN (* A *)
 B(X)
END (* A *).

PROCEDURE B;
BEGIN (* B *)
 W := Y
END (* B *);
```

The first of the two source modules shown may be used without alteration to recompile module A. Combination of the two with appropriate text editing is required to recompile module B.

The requirements of separate compilation of a program having more routines or nested routines to two or more levels are somewhat more complex. The preceding example provides a general idea of what must be done. Specifically the declarations of all routines within which the routine to be separately compiled is nested, and the declarations of the main program must be included. On the other hand, only the statement section of the routine being separately compiled is included. The preparation of source code for separate compilation of a routine can be done manually using utilities. However, the TIP software includes the Configuration Processor (CONFIG) to perform these operations.

## 12.3 THE CONFIGURATION PROCESSOR

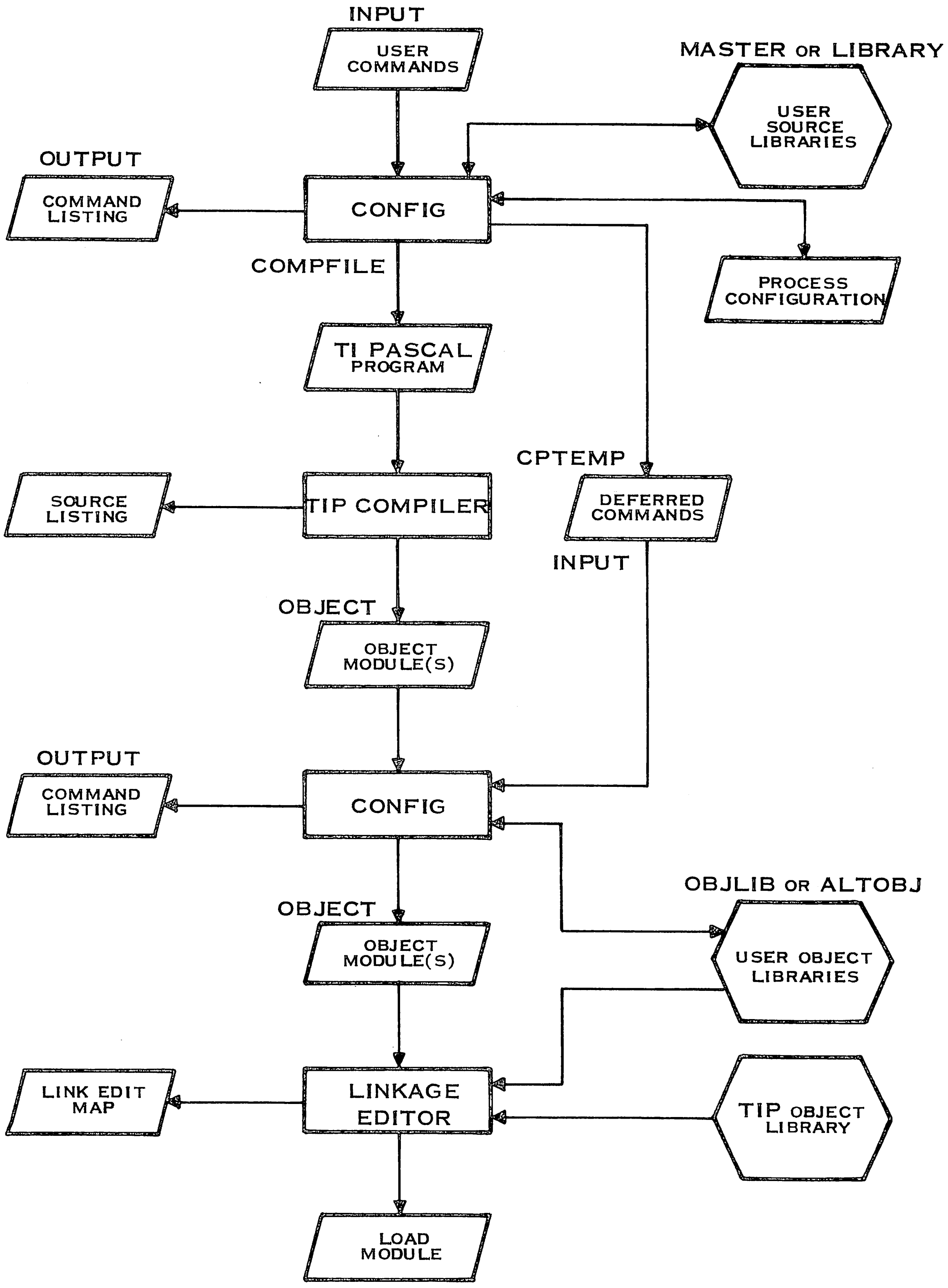
The Configuration Processor supports separate compilation of TIP modules by performing the following functions:

- Maintaining a library of source modules to be combined as required for separate compilation of each module of a program.
- Preparing a source program for each separate compilation.
- Maintaining a library of object modules of a program, from which appropriate object modules are linked.

**12.3.1 FUNCTIONAL DESCRIPTION OF CONFIG.** The functions of CONFIG are shown in figure 12-1, a flowchart of the separate compilation operation. The following description assumes that user source libraries have been prepared that include the source modules required for the separate compilation. These libraries could have been produced by a text editor; i.e., the user could have written the source code as separate modules in a source library. Alternatively, the library could have been created from a source program by a source program splitting utility (SPLITPGM) described in a subsequent paragraph.

The separate compilation using CONFIG consists of the following steps:

- The user executes CONFIG, directing its actions with user commands. In response to these commands, CONFIG communicates with the user source libraries and prepares the desired TI Pascal program. CONFIG also writes a process configuration as specified in the user commands, which describes the hierarchical structure of the program. In addition, CONFIG writes a file of deferred commands for a subsequent execution of CONFIG, and a command listing file that contains the commands and a copy of the process configuration.
- The TIP Compiler processes TI Pascal program written by CONFIG. The compiler produces a separate object module for each routine being compiled, and a source listing.
- CONFIG executes again, using the commands in the deferred command file written during the previous run of CONFIG. The object modules written by the compiler are concatenated on a file; CONFIG separates the modules, writing them as members of the user object libraries. Optionally, CONFIG may collect a full set of object modules to be supplied to the Linkage Editor.
- The Linkage Editor links the object modules with modules from the TIP object library to form a load module (linked object module) and writes the link edit map listing.



(A) 138380

Figure 12-1. Flow of Separate Compilation Using CONFIG

**12.3.2 FORMAT OF SOURCE MODULES.** Source modules for input to CONFIG must be separated and stored as members of user source libraries. They must conform to the following rules:

- A source module consists of one program, procedure, or function in which all contained procedures and functions have been replaced by forward declarations.
- Each procedure and function must be declared in a forward declaration to ensure that each calling sequence is correctly defined.
- Keyword BEGIN of the compound statement that contains the statements of the program, procedure, or function is in character positions 1 through 5. The component statements must be indented.
- Keyword END of the compound statement that contains the statements of the program, procedure, or function is in character positions 1, 2, and 3. The component statements must be indented.
- Compiler option NULLBODY should not be specified in any of the source modules.
- Character position 1 should never contain an asterisk (*).
- Character position 1 should never contain a minus sign (-) unless character position 2 also contains a minus sign.
- A comment in the declaration section that begins in character position 1 must be closed by a brace (}) in character position 72 or an asterisk and parenthesis (*)) in character positions 71 and 72 of the same or a succeeding line.

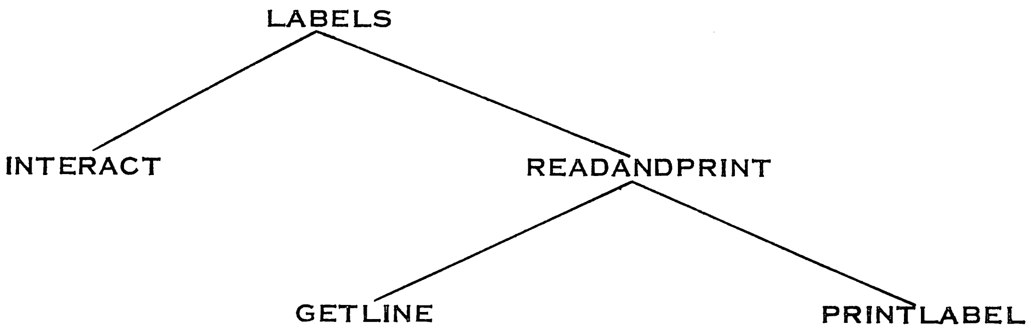
CONFIG recognizes one or more comments in the declaration section of a module preceding the TYPE or VAR declaration as the documentation section of the module. Comments in this section must begin in character position 1 and close in character position 72 of the same or a succeeding line, and may be listed separately from the source code. The LISTDOC command that specifies the modules for which the documentation section is to be listed is described in a subsequent paragraph.

Utility NESTER may be used to comply with indentation requirements (third and fourth rules), and utility SPLITPGM may be used to divide a source program into source modules in accordance with the first rule.

**12.3.3 PROCESS CONFIGURATION.** CONFIG must determine the structure of the program: i.e., the name of the main program, the names of the routines, and the name of the routine within which each routine is declared (or the main program name for global routines). The primary data structure that contains this information is called the process configuration; it is written, maintained, and used by CONFIG. User commands specify the structure and contents of the process configuration.

In some BNF productions for configuration processor commands, angle brackets (<>) are used as terminal symbols. When an angle bracket in a BNF production is a terminal symbol, it is enclosed in quotation marks (" ").

The process configuration is structured as a tree with each node representing a source module of the program. The root node of the process represents the main program module. Consider the following program structure:



The process configuration is represented in tabular form as follows:

```
PROCESS NAME        SOURCE LOCATION        OBJECT LOCATION         FLAGS SET
------------------  ---------------------  ----------------------  -------------
LABELS              < LIBRARY ,LABELS>
  INTERACT          < LIBRARY ,INTERACT>
  READADNP          < LIBRARY ,READANDP>
    GETLINE         < LIBRARY ,GETLINE>
    PRINTLAB        < LIBRARY ,PRINTLAB>
```

The example process configuration corresponds to the commands in the example in paragraph 12.3.3.4. The source locations listed are the locations at which CONFIG accesses the source modules. The library name is the default value (paragraph 12.3.8) because no DEFAULT SOURCE command (paragraph 12.3.8.6) has been entered. No object locations are listed because no DEFAULT OBJECT command (paragraph 12.3.8.7) has been entered. No flags (paragraph 12.3.6) are listed because the initial states of the flags have not been altered.

The commands used to define process configurations are:

- *BUILD PROCESS
- *ADD
- *CAT PROCESS

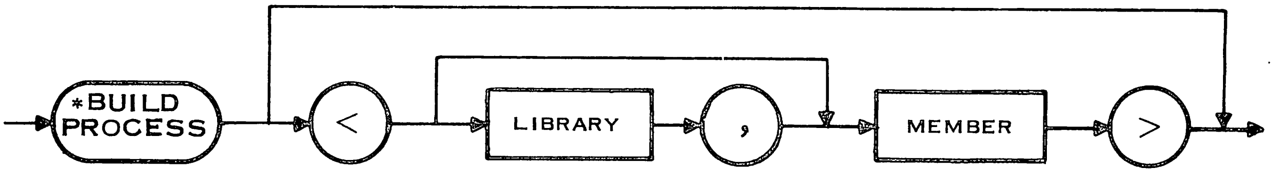
**12.3.3.1 BUILD PROCESS Command.** The BUILD PROCESS command initializes a configuration process as the current configuration process. The syntax of the command is as follows:

<build process command> ::= *BUILD PROCESS [<location>]

<location> ::= "<"[<library>,] <member>">"

The syntax diagram is as follows:

Build process command:



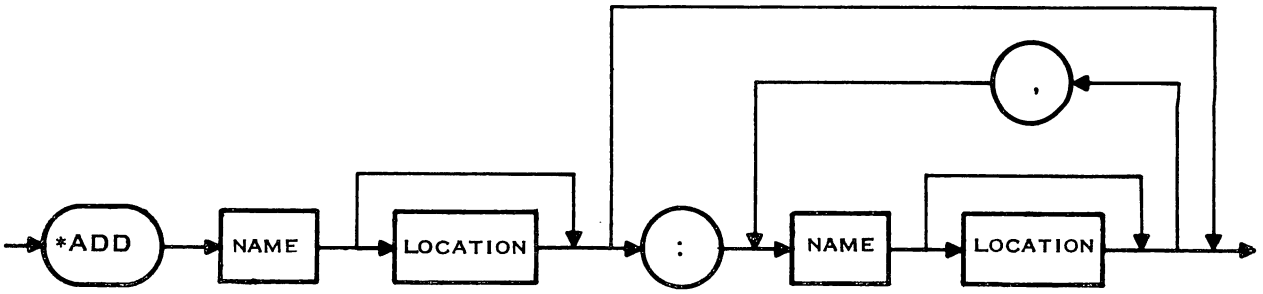
The location parameter is optional. It may be omitted when the location is specified in the CAT PROCESS command. The location consists of a library synonym and a member name. The library synonym may be omitted; the default source library (LIBRARY, unless it has been altered by a DEFAULT SOURCE command) is used.

**12.3.3.2 ADD Command.** The ADD command specifies the name of the root node for a process configuration and, optionally, a location at which the source module for the node is cataloged. An alternate form of the command specifies the name and location of one or more nodes as sons of a specified node. The syntax of the command is as follows:

```
<add command> ::= *ADD <name>[<location>][:<name> [<location>]
  {,<name>[<location>]} ]
```

The syntax diagram is as follows:

Add command:

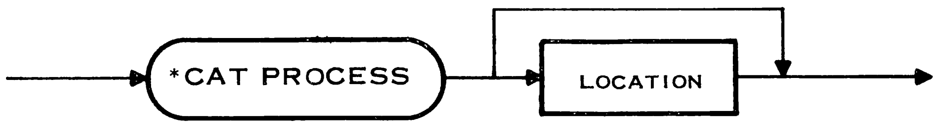


The first ADD command following a BUILD PROCESS command has only one name, that of the root node of a process configuration. The location parameter is the location as defined for the BUILD PROCESS command (paragraph 12.3.3.1) and is optional. When no location is entered, the node name is used as the member of the default source library. Subsequent ADD commands require that the first name parameter is the name of a previously defined node. The location, if entered, is ignored. Name parameters to the right of the colon (:) define additional nodes that are sons of the node named in the first name parameter. The location, if entered, specifies a library and member or a member of the default source library for the module. If no location is entered, the node name is used as the member name of the default source library (LIBRARY, unless it has been altered by a DEFAULT SOURCE command).

**12.3.3.3 CAT PROCESS Command.** The CAT PROCESS command causes the current process to be stored at the specified location. The syntax for the command is as follows:

The syntax diagram is as follows:

Cat process command:



The location is the location as defined for the BUILD PROCESS command (paragraph 12.3.3.1). When a location has been previously specified for the current process configuration (in a BUILD PROCESS or USE PROCESS command), the location may be omitted. When a location is specified, the location in the CAT PROCESS command applies, replacing any previous location.

**12.3.3.4 Process Configuration Command Example.** The following is an example of a set of commands to define a process configuration. The commands in the example define the process configuration described in paragraph 12.3.3:

```
*BUILD PROCESS
*ADD LABELS
*ADD LABELS : INTERACT
*ADD LABELS : READANDP
*ADD READANDP: GETLINE
*ADD READANDP: PRINTLAB
*CAT PROCESS <LIBRARY,PROCESS>
```

The BUILD PROCESS command initializes a process configuration and the first ADD command defines the root node of the structure as the main program, LABELS. The next two ADD commands define two sons of the root node, INTERACT and READANDP. The last two ADD commands define two sons of node READANDP, GETLINE and PRINTLAB. The CAT PROCESS command specifies that the process is cataloged as member PROCESS of a library the pathname of which is the value of synonym LIBRARY. Since the BUILD PROCESS command did not specify a location for the process configuration, the CAT PROCESS command requires a location parameter.

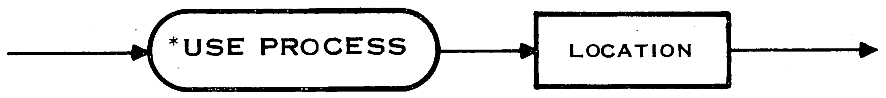
Figure 12-3 shows the source code for the program structure used in the process configuration example.

**12.3.3.5 USE PROCESS Command.** The USE PROCESS command specifies an existing process configuration as the current process configuration. The syntax of the command is as follows:

<use process command> ::= *USE PROCESS <location>

The syntax diagram is as follows:

Use process command:



The location is the location as defined for the BUILD PROCESS command (paragraph 12.3.3.1). The location must be specified, and must be the location of a cataloged process configuration. The process configuration becomes the current process configuration for the remainder of the run. Either a USE PROCESS command or a BUILD PROCESS command must define a current process configuration before any command that operates on a process configuration is entered.

**12.3.4 COMPILATION.** The principal use of CONFIG is in compiling a program or in separately compiling one or more object modules of a program. As shown in figure 12-1, compilation consists of an execution of CONFIG that produces a source module that includes the necessary source library members from which the required module or modules may be compiled, and a file of deferred commands for a subsequent run of CONFIG. The compiler executes, using the source module written by CONFIG, and provides an object file containing the desired module or modules. A second execution of CONFIG uses the deferred command file to control the separation of the object file into an object module library and/or writing an object file for direct input to the linkage editor.

The user commands for compilation include a BUILD PROCESS command, ADD commands, and CAT PROCESS command to define a process configuration, or a USE PROCESS command to specify a previously built process configuration. A COMPILE command to specify the object modules to be compiled is also required. The deferred command file includes a USE PROCESS command, a SPLIT OBJECT command, and an EXIT command. The additional commands used in compilation are described in the following paragraphs.

The USE PROCESS command placed in the deferred command file differs from that entered by the user. The CAT PROCESS command does not store the entire process configuration; the USE PROCESS command previously described would not access all the information required for separating the object modules and/or writing the object file. The USE PROCESS command placed in the deferred commands file is in the following format:

```
*USE PROCESS #
```

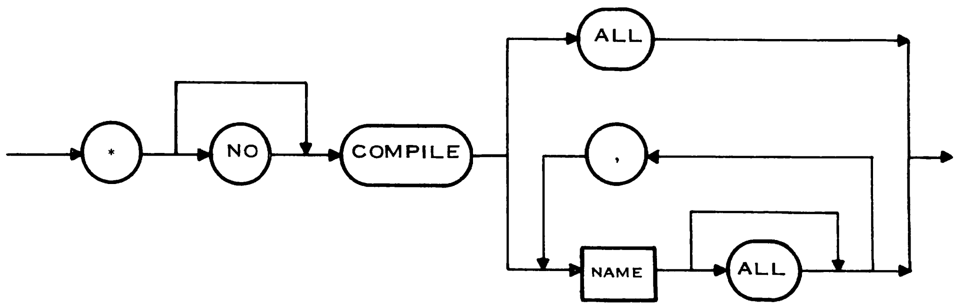
The pound sign (#) indicates that the external representation of the process configuration follows the USE PROCESS command in the deferred commands file. This is the mechanism by which CONFIG passes the entire process configuration to the succeeding run of CONFIG that separates object modules.

**12.3.4.1 COMPILE Command.** The COMPILE command causes CONFIG to prepare a source module for compilation and specifies the module or modules to be compiled. The syntax for the command is as follows:

```
<compile command> ::= *[NO] COMPILE ALL | *[NO] COMPILE <name>[ALL]
{,<name> [ALL]}
```

The syntax diagram is as follows:

Compile command:



The optional keyword NO allows the user to inhibit compilation of the module or modules named in the command. When the keyword ALL is entered following the keyword COMPILE with no name parameter, the command controls compilation of the entire program. The name parameter is the name of the node corresponding to a module to be compiled. When the keyword ALL follows a name parameter, the command controls compilation of the module corresponding to the named node, and the modules for all its descendants. Additional name parameters each optionally followed by keyword ALL may be entered.

The following guidelines apply to selection of modules for recompilation:

- When a statement within the compound statement of a program or routine is changed, recompile the module that contains the program or routine.
- When a declaration of a program is changed (global declaration), recompile the entire program.
- When a declaration of a routine is changed, recompile the module that contains the declaration, and the modules of all nodes that are descendants of the node that contains the declaration.

The COMPILE command controls preparation of the source module for the compilation, including either the declarations of a program or routine, both the declarations and the statements, or neither. When the entire program is to be compiled, the source module contains both the declarations and statements of all routines and of the program. When a module is to be compiled, the source module contains the declarations of the program and routines that correspond to all nodes that are ancestors of the module and the declarations and statements of the module to be compiled. When the module and all its descendants are to be compiled, the declarations of modules that are ancestors of the module are included with the declarations and statements of the module itself and of all modules that are descendants.

**12.3.4.2 SPLIT OBJECT Command.** The SPLIT OBJECT command is placed in the deferred command file when a COMPILE command is included in the INPUT file. The command should not be entered by the user. The format of the command is:

```
*SPLIT OBJECT
```

The SPLIT OBJECT command causes CONFIG to catalog each object module of the object file written by the compiler as a member of a library. The library synonym and/or member name may be specified in a USE OBJECT command or a DEFAULT OBJECT command. Otherwise the default object library is used with the node name as the member name. The default object library is ALTOBJ.

**12.3.4.3 EXIT Command.** The EXIT command is placed in the deferred command file following all other commands placed in the file. The format of the command is:

```
*EXIT
```

When CONFIG reads the EXIT command in the INPUT file it terminates processing. The user may enter the command to abort execution of CONFIG. No files are saved.

**12.3.4.4 Compilation Examples.** The following commands in the INPUT file cause CONFIG to provide a source file that contains all source modules for a program and a deferred command file to catalog all object modules in the object file written by the compiler:

```
*USE PROCESS <LIBRARY, PROCES>
*COMPILE ALL
```

In this example, the process configuration has previously been built and cataloged at location <LIBRARY,PROCES>. If a BUILD PROCESS command, a set of ADD commands, and a CAT PROCESS command were included instead of the USE PROCESS command, a new process configuration would be built and cataloged. The COMPILE command applies to the program that corresponds to the current process configuration, whether the process configuration was built in a previous run of CONFIG or was built in the same run.

Figure 12-2 shows the contents of file OUTPUT following the initial run of CONFIG. The commands are listed first, followed by a tabular representation of the process configuration. The flags set in the process configuration are the result of the COMPILE command as described in a subsequent paragraph. The pathnames corresponding to the files used are listed next, followed by the pathnames assigned to the library synonyms defined by CONFIG.

Figure 12-3 shows the source listing of the compiler run. Notice that CONFIG has inserted some comment lines, but that otherwise the program is identical to the NESTER output (Figure 10-2) of the same source program.

Figure 12-4 shows the contents of file OUTPUT for the deferred processing run of CONFIG. The deferred commands are listed and the object modules are also listed. The node name, and location is shown for each module, followed by the termination record of the module.

A partial compilation of two modules of the same program is performed using the following commands:

```
*USE PROCESS <LIBRARY, PROCES>
*COMPILE PRINTLAB, INTERACT
```

```
CONFIGURATION PROCESSOR                          78. 17                       14:23:
*USE PROCESS <LIBRARY, PROCES>
*COMPILE ALL


PROCESS NAME           SOURCE LOCATION      OBJECT LOCATION      FLAGS SET
-------------------    -------------------  -------------------  ----------------
LABELS                 <LIBRARY ,LABELS  >                       0 1
  INTERACT             <LIBRARY ,INTERACT>                       0 1
  READANDP             <LIBRARY ,READANDP>                       0 1
    GETLINE            <LIBRARY ,GETLINE >                       0 1
    PRINTLAB           <LIBRARY ,PRINTLAB>                       0 1


INPUT     = S109
CRTFIL    = S109
OUTPUT    = DS02.GAS.EXAMP.CF1
COMPFILE  = .COMPFT09
CPTEMP    = .CPTEMP09
OBJECT    = .OBJECT09

MASTEP    = DS02.GAS.EXAMP.SPC
LIBRAPY   = DS02.GAS.EXAMP.SPC
OBJLIF    = DS02.GAS.EXAMP.OBJ
ALTOBJ    = DS02.GAS.EXAMP.OBJ
```

**Figure 12-2. Contents of OUTPUT File, Initial CONFIG Run, Full Compilation**

```
TI PASCAL COMPILER 1.3     DATE = 78. 17       TIME = 14:25:21

(*+           LABELS
    +            INTERACT
    ,            READANDP
     +              GETLINE
     ,              PRINTLAB
 ---         *)
PROGRAM LABELS;
(*-----------------------------------------------------------------------
   PROGRAM LABELS;
   PURPOSE :    THIS PROGRAM READS AN ADDRESS LABEL AND PRINTS MULTIPLE
                COPIES OF THAT LABEL.
   FILES USED : INPUT  - FOR USER-SUPPLIED PARAMETERS AND THE LABEL
                CRTFIL - USED FOR PROMPTING INPUT
                OUTPUT - MULTIPLE COPIES OF THE LABEL
   PROCEDURES CALLED : INTERACT, READANDPRINT
----------------------------------------------------------------------*)
VAR CRTFIL : TEXT ;                            (*USED TO PROMPT INPUT*)
    CHARSPERLINE : INTEGER;          (*NUMBER OF CHARACTERS PER LINE*)
    LINESPERLABEL : INTEGER;             (*NUMBER OF LINES PER LABEL*)
    COPYCOUNT : INTEGER;                 (*NUMBER OF COPIES TO PRINT*)
PROCEDURE INTERACT; FORWARD;
PROCEDURE READANDPRINT; FORWARD;
(*+                                                                   *)
PROCEDURE INTERACT;
(*-----------------------------------------------------------------------
  PROCEDURE INTERACT;
  PURPOSE : INTERACT PROMPTS THE USER, REQUESTING CERTAIN INPUTS.
  OUTPUTS : CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
            LINESPERLABEL - NUMBER OF LINES PER LABEL
            COPYCOUNT - NUMBER OF LABELS TO PRINT
----------------------------------------------------------------------*)
BEGIN                                                    (*INTERACT*)
  REWRITE( CRTFIL );
  WRITELN( CRTFIL, 'HOW MANY CHARACTERS PER LINE?' );
  RESET(INPUT); READ( CHARSPERLINE );
  WRITELN( CRTFIL, 'HOW MANY LINES PER LABEL?' );
  READLN; READ( LINESPERLABEL );
  WRITELN( CRTFIL, 'HOW MANY LABELS?' );
  READLN; READ( COPYCOUNT ); WRITELN(CRTFIL, 'NOW INPUT THE LABEL');
END;                                                     (*INTERACT*)
(*,                                                                   *)
PROCEDURE READANDPRINT;
(*-----------------------------------------------------------------------
  PROCEDURE READANDPRINT;
  PURPOSE : READANDPRINT READS A LABEL AND PRINTS MULTIPLE COPIES OF IT.
  PROCEDURES CALLED : GETLINE, PRINTLABEL
----------------------------------------------------------------------*)
TYPE
   LINE = PACKED ARRAY (.1..CHARSPERLINE.) OF CHAR;
VAR
   LABELIMAGE : ARRAY (.1..LINESPERLABEL.) OF LINE;
PROCEDURE GETLINE(VAR THISLINE : LINE); FORWARD;
```

Figure 12-3. Source Listing, Full Compilation Example (Sheet 1 of 2)

```
PROCEDURE PRINTLABEL; FORWARD;
(*+                                                                    *)
PROCEDURE GETLINE(*VAR THISLINE : LINE*);
(*-------------------------------------------------------------------------
  PROCEDURE GETLINE(VAR THISLINE : LINE);
  PURPOSE : GETLINE READS A SINGLE LINE OF A LABEL.
  INPUTS :  CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
  OUTPUTS : THISLINE - THE LINE THAT WAS READ.
-------------------------------------------------------------------------*)
VAR CH : INTEGER;
BEGIN                                                        (*GETLINE*)
  READLN; CH := 1;
  WHILE CH <= CHARSPERLINE AND NOT EOLN(INPUT) DO BEGIN
    READ( THISLINE(.CH.) ); CH := CH + 1;
    END;                          (*FILL IN REST OF LINE WITH BLANKS*)
  FOR J := CH TO CHARSPERLINE DO THISLINE(.J.) := ' ';
END;                                                         (*GETLINE*)
(*,                                                                    *)
PROCEDURE PRINTLABEL;
(*-------------------------------------------------------------------------
  PROCEDURE PRINTLABEL;
  PURPOSE : PRINTLABEL PRINTS ONE COPY OF THE LABEL.
  INPUTS :  LINESPERLABEL - NUMBER OF LINES PER LABEL
            CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
            LABELIMAGE - THE LABEL TO BE PRINTED
-------------------------------------------------------------------------*)
BEGIN                                                     (*PRINTLABEL*)
  FOR L := 1 TO LINESPERLABEL DO BEGIN
    FOR CH := 1 TO CHARSPERLINE DO WRITE( LABELIMAGE(.L.)(.CH.) );
    WRITELN; END;
END;                                                      (*PRINTLABEL*)
(*-                                                                    *)
BEGIN                                                   (*READANDPRINT*)
  FOR L := 1 TO LINESPERLABEL DO GETLINE( LABELIMAGE(.L.) );
  FOR K := 1 TO COPYCOUNT DO PRINTLABEL;
END;                                                    (*READANDPRINT*)
(*-                                                                    *)
BEGIN                                                         (*LABELS*)
  INTERACT; READANDPRINT;
END.                                                          (*LABELS*)

MAXIMUM NUMBER OF IDENTIFIERS USED = 15
INSTRUCTIONS =     69 (LESS    0 WORDS OF DEAD CODE REMOVED)
  INTERACT  LITERALS =     130  CODE =     308  DATA =      40
INSTRUCTIONS =     34 (LESS    0 WORDS OF DEAD CODE REMOVED)
  GETLINE   LITERALS =      20  CODE =     128  DATA =      50
INSTRUCTIONS =     33 (LESS    0 WORDS OF DEAD CODE REMOVED)
  PRINTLAB  LITERALS =      18  CODE =     116  DATA =      46
INSTRUCTIONS =     58 (LESS    0 WORDS OF DEAD CODE REMOVED)
  READANDP  LITERALS =      28  CODE =     190  DATA =      52
INSTRUCTIONS =     33 (LESS    0 WORDS OF DEAD CODE REMOVED)
  LABELS    LITERALS =      42  CODE =     134  DATA =     166
```

**Figure 12-3. Source Listing, Full Compilation Example (Sheet 2 of 2)**

```
CONFIGURATION PROCESSOR                       78. 17                    14:31:44
*USE PROCESS
*SPLIT OBJECT
INTERACT = <ALTOBJ   ,INTERACT>:      INTERACT  78. 17      14:29:12     TIPSCL
GETLINE  = <ALTOBJ   ,GETLINE >:      GETLINE   78. 17      14:29:30     TIPSCL
PRINTLAB = <ALTOBJ   ,PRINTLAB>:      PRINTLAB  78. 17      14:29:57     TIPSCL
READANDP = <ALTOBJ   ,READANDP>:      READANDP  78. 17      14:30:48     TIPSCL
LABELS   = <ALTOBJ   ,LABELS  >:      LABELS    78. 17      14:31: 5     TIPSCL
*EXIT
```

**Figure 12-4. Contents of OUTPUT File, Deferred Processing, Full Compilation**

The COMPILE command for this example specifies compiling modules INTERACT and PRINTLAB. The source module for the compilation requires the declarations of those modules that are ancestors of these modules; specifically, the declarations of modules LABELS and READANDPRINT. The source module also must include both the declarations and statements of modules PRINTLAB and INTERACT.

Figure 12-5 shows the contents of file OUTPUT for the initial CONFIG run. As in the full compilation example, the file contains the commands and a tabular representation of the process configuration. The flags set in the process configuration (described in a subsequent paragraph) correspond to the portions of modules that are to be combined in the source file that CONFIG builds. The pathnames are identical to those in the preceding example.

Figure 12-6 shows the source listing of the compiler run. The declaration portion of module LABELS is first, including the forward declarations of routines INTERACT and READANDPRINT. Next is module INTERACT, followed by the declaration portion of module READANDPRINT. Forward declarations of routines GETLINE and PRINTLABEL are included, even though no portion of module GETLINE is included in the partial compilation. Module PRINTLABEL is next, followed by the statement portions of routine READANDPRINT and program LABELS. To inhibit the compiler from writing modules READANDP and LABELS, the statement portions of these modules supplied by CONFIG consist of BEGIN keywords followed by NULLBODY option comments and END keywords.

```
CONFIGURATION PROCESSOR                      78. 17                   15: 6:18
*USE PROCESS <LIBRARY, PROCES>
*COMPILE PRINTLAB, INTERACT


PROCESS NAME          SOURCE LOCATION      OBJECT LOCATION     FLAGS SET
-------------------- ------------------- ------------------- -----------------
LABELS                <LIBRARY ,LABELS  >                      0
  INTERACT            <LIBRARY ,INTERACT>                      0 1
  READANDP            <LIBRARY ,READANDP>                      0
    GETLINE           <LIBRARY ,GETLINE >
    PRINTLAB          <LIBRARY ,PRINTLAB>                      0 1

INPUT    = ST09
CRTFIL   = ST09
OUTPUT   = DS02.GAS.EXAMP.CF1
COMPFILE = .COMPFT09
CPTEMP   = .CPTEMP09
OBJECT   = .OBJECT09

MASTER   = DS02.GAS.EXAMP.SRC
LIBRARY  = DS02.GAS.EXAMP.SRC
OBJLIB   = DS02.GAS.EXAMP.OBJ
ALTOBJ   = DS02.GAS.EXAMP.OBJ
```

**Figure 12-5. Contents of OUTPUT File, Initial Run, Partial Compilation**

```
TI PASCAL COMPILER 1.3    DATE = 78. 17      TIME = 15: 7:33

(*+
   +          INTERACT
   ,
    +          PRINTLAB
 ---       *)
PROGRAM LABELS;
(*-----------------------------------------------------------------------
   PROGRAM LABELS;
   PURPOSE :    THIS PROGRAM READS AN ADDRESS LABEL AND PRINTS MULTIPLE
                COPIES OF THAT LABEL.
   FILES USED : INPUT  - FOR USER-SUPPLIED PARAMETERS AND THE LABEL
                CRTFIL - USED FOR PROMPTING INPUT
                OUTPUT - MULTIPLE COPIES OF THE LABEL
   PROCEDURES CALLED : INTERACT, READANDPRINT
---------------------------------------------------------------------*)
VAR CRTFIL : TEXT ;                                (*USED TO PROMPT INPUT*)
    CHARSPERLINE : INTEGER;               (*NUMBER OF CHARACTERS PER LINE*)
    LINESPERLABEL : INTEGER;                  (*NUMBER OF LINES PER LABEL*)
    COPYCOUNT : INTEGER;                      (*NUMBER OF COPIES TO PRINT*)
PROCEDURE INTERACT; FORWARD;
PROCEDURE READANDPRINT; FORWARD;
(*+                                                                     *)
PROCEDURE INTERACT;
(*-----------------------------------------------------------------------
  PROCEDURE INTERACT;
  PURPOSE : INTERACT PROMPTS THE USER, REQUESTING CERTAIN INPUTS.
  OUTPUTS : CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
            LINESPERLABEL - NUMBER OF LINES PER LABEL
            COPYCOUNT - NUMBER OF LABELS TO PRINT
---------------------------------------------------------------------*)
BEGIN                                                      (*INTERACT*)
  REWRITE( CRTFIL );
  WRITELN( CRTFIL, 'HOW MANY CHARACTERS PER LINE?' );
  RESET(INPUT); READ( CHARSPERLINE );
  WRITELN( CRTFIL, 'HOW MANY LINES PER LABEL?' );
  READLN; READ( LINESPERLABEL );
  WRITELN( CRTFIL, 'HOW MANY LABELS?' );
  READLN; READ( COPYCOUNT ); WRITELN(CRTFIL, 'NOW INPUT THE LABEL');
END;                                                       (*INTERACT*)
(*,                                                                     *)
PROCEDURE READANDPRINT;
(*-----------------------------------------------------------------------
  PROCEDURE READANDPRINT;
  PURPOSE : READANDPRINT READS A LABEL AND PRINTS MULTIPLE COPIES OF IT.
  PROCEDURES CALLED : GETLINE, PRINTLABEL
---------------------------------------------------------------------*)
TYPE
    LINE = PACKED ARRAY (.1..CHARSPERLINE.) OF CHAR;
VAR
    LABELIMAGE : ARRAY (.1..LINESPERLABEL.) OF LINE;
PROCEDURE GETLINE(VAR THISLINE : LINE); FORWARD;
PROCEDURE PRINTLABEL; FORWARD;
```

**Figure 12-6. Source Listing, Partial Compilation Example (Sheet 1 of 2)**

```
(*+                                                                    *)
PROCEDURE PRINTLABEL;
(*---------------------------------------------------------------------
  PROCEDURE PRINTLABEL;
  PURPOSE : PRINTLABEL PRINTS ONE COPY OF THE LABEL.
  INPUTS :  LINESPERLABEL - NUMBER OF LINES PER LABEL
            CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
            LABELIMAGE '- THE LABEL TO BE PRINTED
----------------------------------------------------------------------*)
BEGIN                                                    (*PRINTLABEL*)
  FOR L := 1 TO LINESPERLABEL DO BEGIN
    FOR CH := 1 TO CHARSPERLINE DO WRITE( LABELIMAGE(.L.)(.CH.) );
    WRITELN; END;
END;                                                     (*PRINTLABEL*)
(*-                                                                    *)
BEGIN (*NULLBODY *)
END   (* READANDP *);
(*-                                                                    *)
BEGIN (*NULLBODY *)
END   (* LABELS   *).

MAXIMUM NUMBER OF IDENTIFIERS USED = 14
INSTRUCTIONS =      69 (LESS    0 WORDS OF DEAD CODE REMOVED)
  INTERACT  LITERALS =      130  CODE =     308  DATA =      40
INSTRUCTIONS =      33 (LESS    0 WORDS OF DEAD CODE REMOVED)
  PRINTLAB  LITERALS =       18  CODE =     116  DATA =      46
```

**Figure 12-6. Source Listing, Partial Compilation Example (Sheet 2 of 2)**

Figure 12-7 shows the contents of file OUTPUT for the deferred processing. The same deferred commands are used as for full compilation and the object modules written by the compiler are listed. The newly compiled modules for the specified routines (INTERACT and PRINTLABEL) replace the previously compiled modules as members of library ALTOBJ.

**12.3.5 SOURCE LISTING.** CONFIG supports the listing of the source modules of source libraries specified in a process configuration. Two commands are provided. The LIST command specifies listing of one or more complete source modules. The LISTDOC command lists the documentation section of one or more source modules.

The documentation section of a source module consists of one or more comments at the beginning of the declaration section, preceding the TYPE declaration, if any, or the VAR declaration. The brace ( { ) or parenthesis and asterisk ((*) that begin the comment must be in character position 1 or character positions 1 and 2 respectively. The closing brace ( } ) or asterisk and parenthesis (*)) must be *in character* position 72 or character positions 71 and 72 of the same or of a subsequent line. The declaration section may consist of a multiline comment as in the example or of a group of comments.

The listings are written after all commands in the INPUT file have been processed and show the effect, if any, of any of these commands on the listing.

**12.3.5.1 LIST Command.** The LIST command causes CONFIG to list one or more source modules specified in the current process configuration. The syntax for the command is as follows:

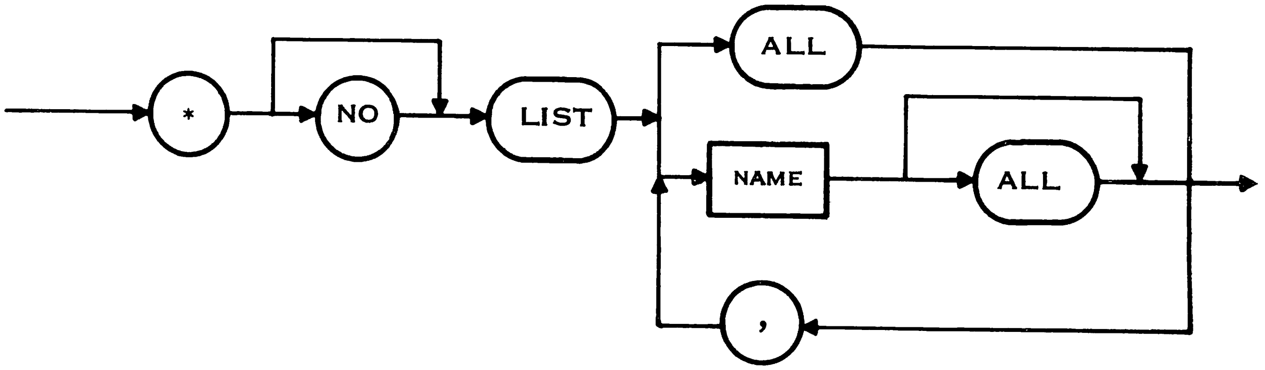
<list command> ::= *[NO] LIST ALL | *[NO] LIST <name>[ALL] {,<name>[ALL]}

```
CONFIGURATION PROCESSOR                        78. 17                       15:10:28
*USE PROCESS
*SPLIT OBJECT
INTERACT = <ALTOBJ    ,INTERACT>:         INTERACT  78. 17      15: 9:17     TIPSCL
PRINTLAB = <ALTOBJ    ,PRINTLAB>:         PRINTLAB  78. 17      15: 9:51     TIPSCL
*EXIT
```

**Figure 12-7. Contents of OUTPUT File, Deferred Processing, Partial Compilation**

The syntax diagram is as follows:

List command:



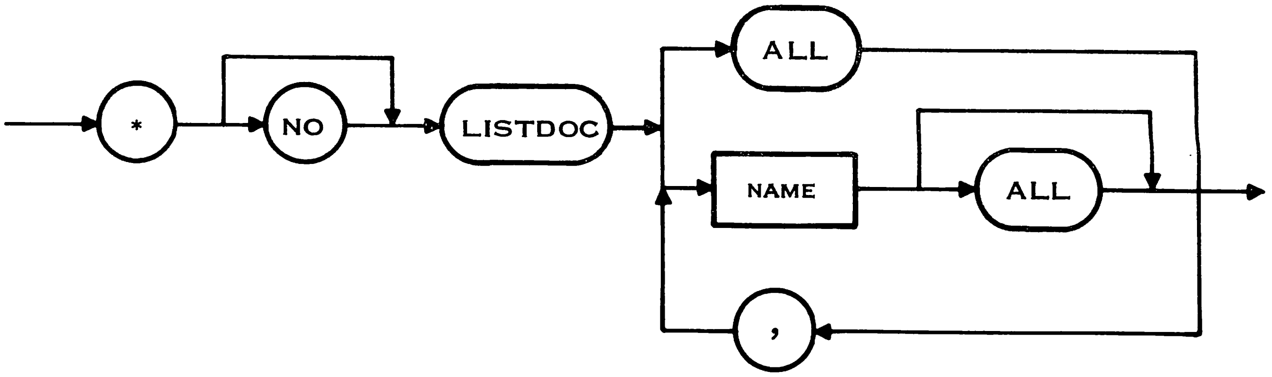
The name parameter is the name of a node in the current process configuration. The source module corresponding to each named node is listed. When the keyword ALL is entered alone, all source modules of the program are listed. When the keyword ALL is entered following a name parameter, the command lists the specified module and all descendants.

**12.3.5.2 LISTDOC Command.** The LISTDOC command causes CONFIG to list the documentation section of one or more source modules specified in the current process configuration. The syntax for the command is as follows:

<listdoc command> ::= *[NO] LISTDOC ALL|*[NO] LISTDOC <name>[ALL] ,<name>[ALL]

The syntax diagram is as follows:

Listdoc command:



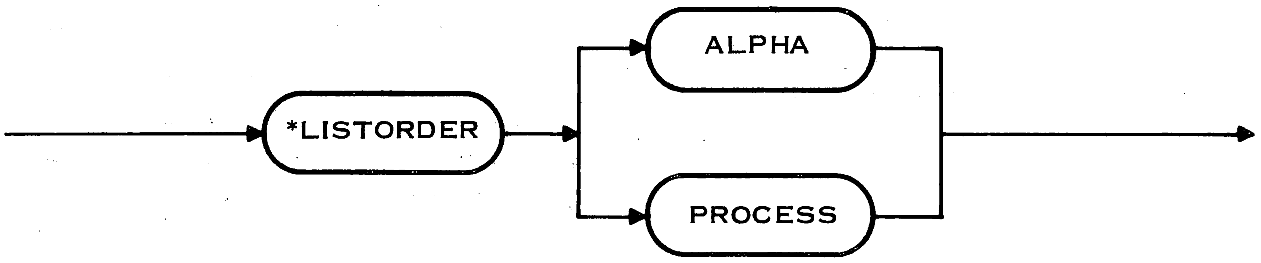
The name parameter is the name of a node in the current process configuration. The documentation section of the source module corresponding to each named node is listed. When the keyword ALL is entered alone, the documentation sections of all source modules of the program are listed. When the keyword ALL is entered following a name parameter, the command lists the documentation sections of the specified module and all descendants.

**12.3.5.3 LISTORDER Command.** The LISTORDER command specifies the listing order for the LIST and LISTDOC commands. The syntax of the command is as follows:

<listorder command> ::= *LISTORDER ALPHA | *LISTORDER PROCESS

The syntax diagram is as follows:

Listorder command:



The keyword ALPHA specifies alphabetic order by node name for source modules listed by a LIST command or documentation sections of source modules listed by a LISTDOC command. The keyword PROCESS specifies listing the source modules in the order in which they appear in the process configuration.

The LIST and LISTDOC commands list source modules in the order in which they appear in the process configuration, unless a LISTORDER command has specified alphabetic order. The alphabetic order lists modules in alphabetic order by node name, and applies to all LIST and LISTDOC commands.

Example:

```
*LISTORDER ALPHA
```

The example command specifies listing source modules and documentation sections of source modules in alphabetic order by node name.

**12.3.5.4 Listing Examples.** Figure 12-8 lists the contents of the OUTPUT file for a listing example. The commands shown are a USE PROCESS command that accesses a previously cataloged process configuration as the current process configuration and a LIST command that lists the entire program. The tabular representation of the process configuration is essentially the same as for the preceding examples. The flag section shows that a different flag is set for each node. The flag section is described in a subsequent paragraph.

The source modules are listed in the order in which they are listed in the process configuration. Notice that each module contains only the declarations and statements of the program or routine. None of the modules could be compiled alone; the LABELS module would fail because it contains two forward declarations of routines, but does not contain the code for the routines. The other modules would fail because they do not start with a PROGRAM heading and do not end with a period (.). The modules form a source library from which CONFIG can write a source module to compile any one or more of the modules of the program.

Figure 12-9 lists the contents of the OUTPUT file for an example of listing the documentation sections of a program. The commands shown are a USE PROCESS command that accesses a previously cataloged process configuration and a LISTDOC command that lists the entire program. The tabular representation of the program shows a different flag set for each node. The flag section is described in a subsequent paragraph.

The documentation section of a source module consists of one or more comments at the beginning of the declaration section, preceding the TYPE declaration, if any, or the VAR declaration. The brace ({ ) or parenthesis and asterisk ((*) that begin the comment must be in character position 1 or character positions 1 and 2 respectively. The closing brace ( }) or asterisk and parenthesis (*)) must be in character position 72 or character positions 71 and 72 of the same or of a subsequent line. The documentation section may consist of a multiline comment as in the example or of a group of comments.

**12.3.6 FLAGS.** The process configuration contains a set of flags for each node that control the processing of the node. Each flag is either on or off. Flags are turned on or off by commands. When all commands have been processed, the states of all flags resulting from processing the commands is passed to the deferred processing run of CONFIG in the external representation of the process configuration that follows the USE PROCESS # command in the deferred command file.

There are two categories of flags: system flags and user flags. System flags are predefined and are set to an initial state when a process configuration is built or accessed. The states of system flags are not stored when the process configuration is stored. The system flags, their significance, and their initial states are listed in table 12-1. User flags are described in a subsequent paragraph.

The COMPILE command (paragraph 12.3.4.1) turns the DECLARATION flag on for each module for which the declarations are required in the source file being written. The command turns on both the DECLARATION and BODY flags for modules being compiled. Similarly, the NO COMPILE commands turn off the DECLARATION and BODY flags appropriately.

```
CONFIGURATION PROCESSOR                     78. 17                    13:41:27
*USE PROCESS <LIBRARY, PROCES>
*LIST LABELS ALL


PROCESS NAME          SOURCE LOCATION     OBJECT LOCATION     FLAGS SET
--------------------  ------------------  ------------------  ----------------
LABELS                <LIBRARY ,LABELS  >                     2
  INTERACT            <LIBRARY ,INTERACT>                     2
  READANDP            <LIBRARY ,READANDP>                     2
    GETLINE           <LIBRARY ,GETLINE >                     2
    PRINTLAB          <LIBRARY ,PRINTLAB>                     2


INPUT    = TIPDISC2.GAS.EXAMP.CONFIGI
CRTFIL   = ST10
OUTPUT   = TIPDISC2.GAS.EXAMP.CONFIGO2
COMPFILE = DUMY
CPTEMP   = .CPTEMP10
OBJECT   = DUMY

MASTER   = DS02.GAS.EXAMP.SRC
LIBRARY  = DS02.GAS.EXAMP.SRC
OBJLIB   = DS02.GAS.EXAMP.OBJ
ALTOBJ   = DS02.GAS.EXAMP.OBJ

CONFIGURATION PROCESSOR                     78. 17                    13:41:38
LABELS   = DS02.GAS.EXAMP.SRC(LABELS   )


PROGRAM LABELS:                                                          000010
(*-----------------------------------------------------------------------000020
   PROGRAM LABELS:                                                       000030
   PURPOSE :    THIS PROGRAM READS AN ADDRESS LABEL AND PRINTS MULTIPLE  000040
                COPIES OF THAT LABEL.                                    000050
   FILES USED : INPUT   - FOR USER-SUPPLIED PARAMETERS AND THE LABEL     000060
                CRTFIL  - USED FOR PROMPTING INPUT                       000070
                OUTPUT  - MULTIPLE COPIES OF THE LABEL                   000080
   PROCEDURES CALLED : INTERACT, READANDPRINT                            000090
-----------------------------------------------------------------------*)000100
VAR CRTFIL : TEXT ;                         (*USED TO PROMPT INPUT*)     000110
    CHARSPERLINE : INTEGER;          (*NUMBER OF CHARACTERS PER LINE*)   000120
    LINESPERLABEL : INTEGER;            (*NUMBER OF LINES PER LABEL*)    000130
    COPYCOUNT : INTEGER;                (*NUMBER OF COPIES TO PRINT*)    000140
PROCEDURE INTERACT; FORWARD;                                             000150
PROCEDURE READANDPRINT; FORWARD;                                         000160
BEGIN                                                   (*LABELS*)       000170
  INTERACT; READANDPRINT;                                                000180
END.                                                    (*LABELS*)       000190
```

Figure 12-8. Contents of OUTPUT File for LIST Operation (Sheet 1 of 3)

```
CONFIGURATION PROCESSOR                       78. 17                        13:41:45
INTERACT = DS02.GAS.EXAMP.SRC(INTERACT)


PROCEDURE INTERACT;                                                         000010
(*------------------------------------------------------------------------000020
  PROCEDURE INTERACT;                                                       000030
  PURPOSE : INTERACT PROMPTS THE USER, REQUESTING CERTAIN INPUTS.           000040
  OUTPUTS : CHARSPERLINE - NUMBER OF CHARACTERS PER LINE                    000050
            LINESPERLABEL - NUMBER OF LINES PER LABEL                       000060
            COPYCOUNT - NUMBER OF LABELS TO PRINT                           000070
-----------------------------------------------------------------------*)000080
BEGIN                                                  (*INTERACT*)         000090
  REWRITE( CRTFIL );                                                        000100
  WRITELN( CRTFIL, 'HOW MANY CHARACTERS PER LINE?' );                       000110
  RESET(INPUT); READ( CHARSPERLINE );                                       000120
  WRITELN( CRTFIL, 'HOW MANY LINES PER LABEL?' );                           000130
  READLN; READ( LINESPERLABEL );                                            000140
  WRITELN( CRTFIL, 'HOW MANY LABELS?' );                                    000150
  READLN; READ( COPYCOUNT ); WRITELN(CRTFIL, 'NOW INPUT THE LABEL');        000160
END;                                                   (*INTERACT*)         000170

CONFIGURATION PROCESSOR                       78. 17                        13:41:50
READANDP = DS02.GAS.EXAMP.SRC(READANDP)


PROCEDURE READANDPRINT;                                                     000010
(*------------------------------------------------------------------------000020
  PROCEDURE READANDPRINT;                                                   000030
  PURPOSE : READANDPRINT READS A LABEL AND PRINTS MULTIPLE COPIES OF IT.000040
  PROCEDURES CALLED : GETLINE, PRINTLABEL                                   000050
-----------------------------------------------------------------------*)000060
TYPE                                                                        000070
    LINE = PACKED ARRAY (.1..CHARSPERLINE.) OF CHAR;                        000080
VAR                                                                         000090
    LABELIMAGE : ARRAY (.1..LINESPERLABEL.) OF LINE;                        000100
PROCEDURE GETLINE(VAR THISLINE : LINE); FORWARD;                            000110
PROCEDURE PRINTLABEL; FORWARD;                                              000120
BEGIN                                              (*READANDPRINT*)         000130
  FOR L := 1 TO LINESPERLABEL DO GETLINE( LABELIMAGE(.L.) );                000140
  FOR K := 1 TO COPYCOUNT DO PRINTLABEL;                                    000150
END;                                               (*READANDPRINT*)         000160
```

Figure 12-8. Contents of OUTPUT File for LIST Operation (Sheet 2 of 3)

```
CONFIGURATION PROCESSOR                    78. 17                     13:41:52
GETLINE   = DS02.GAS.EXAMP.SRC(GETLINE )


PROCEDURE GETLINE(*VAR THISLINE : LINE*);
(*--------------------------------------------------------------------------
  PROCEDURE GETLINE(VAR THISLINE : LINE);
  PURPOSE : GETLINE READS A SINGLE LINE OF A LABEL.
  INPUTS :  CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
  OUTPUTS : THISLINE - THE LINE THAT WAS READ.
--------------------------------------------------------------------------*)
VAR CH : INTEGER;
BEGIN                                                       (*GETLINE*)
  READLN; CH := 1;
  WHILE CH <= CHARSPERLINE AND NOT EOLN(INPUT) DO BEGIN
    READ( THISLINE(.CH.) ); CH := CH + 1;
    END;                         (*FILL IN REST OF LINE WITH BLANKS*)
  FOR J := CH TO CHARSPERLINE DO THISLINE(.J.) := ' ';
END;                                                        (*GETLINE*)

CONFIGURATION PROCESSOR                    78. 17                     13:41:54
PRINTLAB = DS02.GAS.EXAMP.SRC(PRINTLAB)


PROCEDURE PRINTLABEL;
(*--------------------------------------------------------------------------
  PROCEDURE PRINTLABEL;
  PURPOSE : PRINTLABEL PRINTS ONE COPY OF THE LABEL.
  INPUTS :  LINESPERLABEL - NUMBER OF LINES PER LABEL
            CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
            LABELIMAGE - THE LABEL TO BE PRINTED
--------------------------------------------------------------------------*)
BEGIN                                                    (*PRINTLABEL*)
  FOR L := 1 TO LINESPERLABEL DO BEGIN
    FOR CH := 1 TO CHARSPERLINE DO WRITE( LABELIMAGE(.L.)(.CH.) );
    WRITELN; END;
END;                                                     (*PRINTLABEL*)
```

**Figure 12-8. Contents of OUTPUT File for LIST Operation (Sheet 3 of 3)**

The LIST command turns on the LIST flag for modules to be listed, and the NO LIST command turns the LIST flag OFF for the specified module or modules. Similarly, the LISTDOC command turns on the LISTDOC flag and the NO LISTDOC command turns the LISTDOC flag OFF.

The use of the CHANGED flag which is set by the EDIT command is described in a subsequent paragraph. The NEST flag may not be set or cleared by the user.

The COLLECT flag is turned on and off by the Flag command described in paragraph 12.3.6.2. When the flag is turned on, the module corresponding to the node for which the COLLECT flag is turned on is written to the OBJECT file during the deferred processing run. The COLLECT flag implements optional output to a file to be specified in an INCLUDE command in the link edit control file, making it unnecessary for the linkage editor to search the library for the module.

When the COLLECT flag is set for any node of the process configuration, CONFIG places the following command in the deferred command file following the *SPLIT OBJECT command:

```
*COLLECT OBJECT
```

The command is executed during the deferred processing run at the point at which the command is read. When the location of the object module has not been specified by a USE OBJECT command or by a preceding collect operation, CONFIG searches the library specified by ALTOBJ for the module. When the module is not on the ALTOBJ library, CONFIG next searches the library specified by OBJLIB.

The SPLIT flag is turned on and off by the Flag command described in paragraph 12.3.6.2. The flag is initially on, causing all modules to be cataloged as members of the specified object library. A module for which the SPLIT flag is not set is not cataloged during the deferred processing run.

The CHECK flag is turned on and off by the Flag command described in paragraph 12.3.6.2. The flag is initially on, causing the check of the IDT of the module to be made on all modules. When the CHECK flag is on, the IDT of the module is compared to the name of the node; processing is terminated and an error message is written when the name and IDT are not identical. The IDT of the module is not checked when the CHECK flag for the node has been turned off.

The user may define up to 21 user flags using the SETFLAG command. User flags are turned on and off by the flag command. The states of user flags are stored when the process configuration is stored. The conditional flag command described in a subsequent paragraph may be used to test user flags and set system flags.

```
CONFIGURATION PROCESSOR                        78. 17                         13:42:58
*USE PROCESS <LIBRARY, PROCES>
*LISTDOC LABELS ALL


PROCESS NAME            SOURCE LOCATION       OBJECT LOCATION      FLAGS SET
-------------------     -----------------     -----------------    ---------------
LABELS                  <LIBRARY ,LABELS  >                        3
  INTERACT              <LIBRARY ,INTERACT>                        3
  READANDP              <LIBRARY ,READANDP>                        3
    GETLINE             <LIBRARY ,GETLINE >                        3
    PRINTLAB            <LIBRARY ,PRINTLAB>                        3


INPUT    = TIPDISC2.GAS.EXAMP.CONFIGI2
CRTFIL   = ST10
OUTPUT   = TIPDISC2.GAS.EXAMP.CONFIG03
COMPFILE = DUMY
CPTEMP   = .CPTEMP10
OBJECT   = DUMY

MASTER   = DS02.GAS.EXAMP.SRC
LIBRARY  = DS02.GAS.EXAMP.SRC
OBJLIB   = DS02.GAS.EXAMP.OBJ
ALTOBJ   = DS02.GAS.EXAMP.OBJ

LABELS   = DS02.GAS.EXAMP.SRC(LABELS  )
(*-------------------------------------------------------------------------000020
   PROGRAM LABELS;                                                        000030
   PURPOSE :    THIS PROGRAM READS AN ADDRESS LABEL AND PRINTS MULTIPLE  000040
                COPIES OF THAT LABEL.                                     000050
   FILES USED : INPUT  - FOR USER-SUPPLIED PARAMETERS AND THE LABEL       000060
                CRTFIL - USED FOR PROMPTING INPUT                         000070
                OUTPUT - MULTIPLE COPIES OF THE LABEL                     000080
   PROCEDURES CALLED : INTERACT, READANDPRINT                             000090
------------------------------------------------------------------------*)000100


INTERACT = DS02.GAS.EXAMP.SRC(INTERACT)
(*-------------------------------------------------------------------------000020
  PROCEDURE INTERACT;                                                     000030
  PURPOSE : INTERACT PROMPTS THE USER, REQUESTING CERTAIN INPUTS.         000040
  OUTPUTS : CHARSPERLINE - NUMBER OF CHARACTERS PER LINE                  000050
            LINESPERLABEL - NUMBER OF LINES PER LABEL                     000060
            COPYCOUNT - NUMBER OF LABELS TO PRINT                         000070
------------------------------------------------------------------------*)000080


READANDP = DS02.GAS.EXAMP.SRC(READANDP)
(*-------------------------------------------------------------------------000020
  PROCEDURE READANDPRINT;                                                 000030
  PURPOSE : READANDPRINT READS A LABEL AND PRINTS MULTIPLE COPIES OF IT.000040
  PROCEDURES CALLED : GETLINE, PRINTLABEL                                 000050
------------------------------------------------------------------------*)000060
```

**Figure 12-9. Contents of OUTPUT File for LISTDOC Operation (Sheet 1 of 2)**

```
GETLINE  = DS02.GAS.EXAMP.SRC(GETLINE )
(*-----------------------------------------------------------------------
  PROCEDURE GETLINE(VAR THISLINE : LINE);
  PURPOSE : GETLINE READS A SINGLE LINE OF A LABEL.
  INPUTS :  CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
  OUTPUTS : THISLINE - THE LINE THAT WAS READ.
--------------------------------------------------------------------*)

PRINTLAB = DS02.GAS.EXAMP.SRC(PRINTLAB)
(*-----------------------------------------------------------------------
  PROCEDURE PRINTLABEL;.
  PURPOSE : PRINTLABEL PRINTS ONE COPY OF THE LABEL.
  INPUTS :  LINESPERLABEL - NUMBER OF LINES PER LABEL
            CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
            LABELIMAGE - THE LABEL TO BE PRINTED
--------------------------------------------------------------------*)
```

**Figure 12-9. Contents of OUTPUT File for LISTDOC Operation (Sheet 2 of 2)**

**Table 12-1. System Flags**

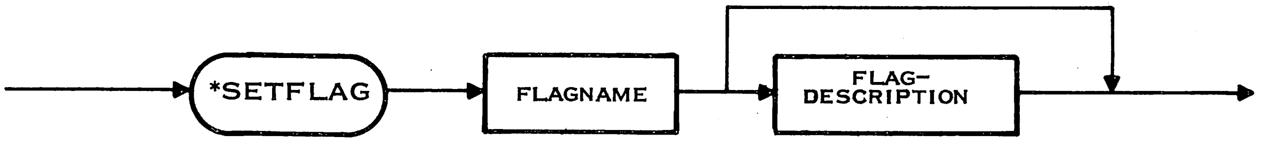
| Flag Number | Flag Name | Description | Initial Value |
|---|---|---|---|
| 0 | DECLARATION | Set when declarations of this module are required in source file. | OFF |
| 1 | BODY | Set when statements of this module are required in source file. | OFF |
| 2 | LIST | Set when the source module is to be listed. | OFF |
| 3 | LISTDOC | Set when the documentation section of this module is to be listed. | OFF |
| 4 | CHANGED | Set when contents of a source module are changed by an edit operation. | OFF |
| 5 | NEST | Not currently used. | |
| 6 | SPLIT | Set when the object module is to be written as a member of library OBJLIB or ALTOBJ. | ON |
| 7 | COLLECT | Set when the object module is to be written on the OBJECT file. | OFF |
| 8 | CHECK | Set when the IDT of the module is to be compared to the name of the node. | ON |

**12.3.6.1 SETFLAG Command.** The SETFLAG command defines or deletes the definition of a user flag. The syntax of the commands is as follows:

<setflag command> ::= * SETFLAG <flagname>[<flag-description>]

The syntax diagram is as follows:

Setflag command:



The flagname consists of one to eight characters and may not be a CONFIG keyword. The flag description is a string of up to 64 characters that describes the flag. The flag description begins with the first nonblank character following the flagname and extends to the first asterisk (*), normally the asterisk that begins the next command. The flag description may contain blanks, and serves as a comment to identify the flag. When the flag description is omitted, the definition of that flag is deleted, and the flag is turned off in all nodes in the program.

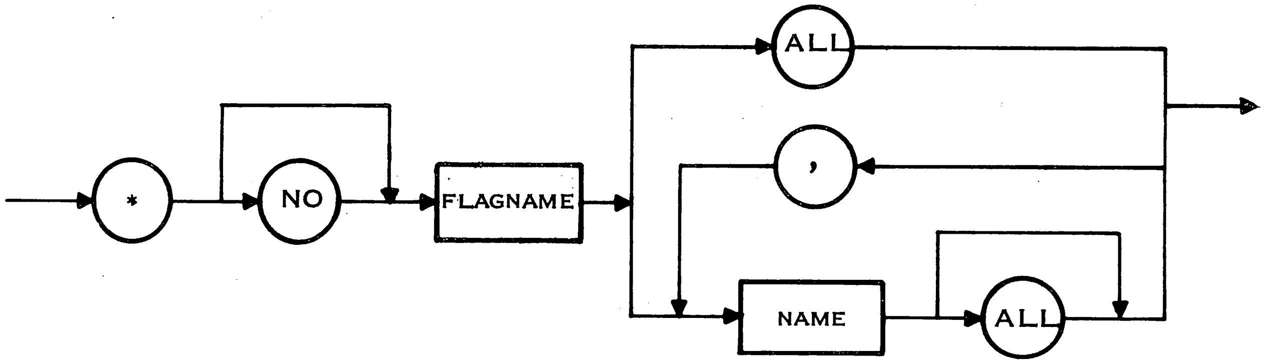
**12.3.6.2 Flag Command.** The Flag command turns certain system flags and all user flags on or off. The syntax for the command is as follows:

<flag command> ::= *[NO]<flagname>ALL | *[NO] <flagname><name>[ALL] {,<name>[ALL]}

<flagname> ::= SPLIT|COLLECT|CHECK|<user flagname>

The syntax diagram is as follows:

Flag command:



When the optional keyword NO is entered the flag is turned off. Otherwide the flag is turned on. When the keyword ALL immediately follows the flagname, the flag is turned on or off in all nodes of the current process configuration. The name parameter specifies a node in which the flag is turned on or off. When the name parameter is followed by the optional keyword ALL, the FLAG is turned on or off in the named node and all its descendants.

**12.3.6.3 Conditional Flag Command.** The Conditional Flag command tests a specified flag and turns another specified flag on or off according to the result. The syntax of the command is as follows:

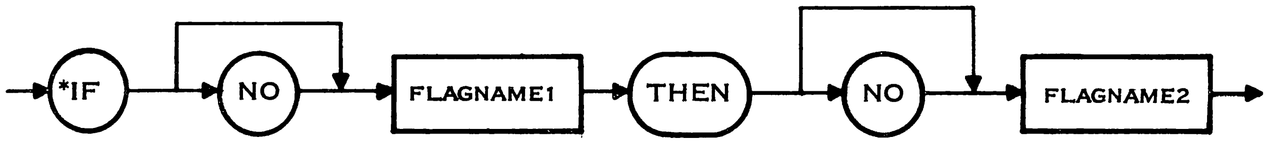
<conditional flag command>::= * IF [NO]<flagname1>THEN[NO]<flagname2>

<flagname1> ::= COMPILE|LIST|LISTDOC|CHANGED|SPLIT|COLLECT|CHECK|<user flagname>

<flagname2> ::= COMPILE|LIST|LISTDOC|SPLIT|COLLECT|CHECK|<user flagname>

The syntax diagram is as follows:

Conditional flag command:



The flagname1 parameter specifies a flag to be tested in all nodes of the current process configuration. The optional keyword NO preceding filename1 specifies the state tested for; when NO is entered the flag is tested for the off state; when NO is omitted the flag is tested for the on state. In each node for which the test is successful flagname2 is turned on when optional keyword NO is omitted, or off when NO is entered.

Notice that COMPILE is allowed as a flagname in the conditional flag command even though it is not the name of a flag. When COMPILE is entered as flagname1 the BODY flag is tested. When COMPILE is entered as flagname2, the BODY flag is turned on or off as specified. When the BODY flag is turned on, the DECLARATION flag is turned on also.

**12.3.6.4 Flag Examples.** The Flag commands allow the user to control the processing of the program by setting or resetting the system flags. The following is an example of a Flag command:

```
*COLLECT INTERACT
```

The command turns on the COLLECT flag for module INTERACT, and causes CONFIG to include the following in the deferred command file following the *SPLIT OBJECT command:

```
*COLLECT OBJECT
```

The deferred processing run of CONFIG writes the module for INTERACT (and any others for which the COLLECT flag is on) to the OBJECT file. The OBJECT file can be specified in an INCLUDE command to the linkage editor.

Another example of a Flag command is:

```
*NO SPLIT ALL
```

This command turns off the SPLIT flags for all modules. The deferred processing run of CONFIG does not catalog the object modules. Unless the COLLECT flag is set for one or more modules the use of this example is of doubtful value.

The Conditional Flag command allows the user to selectively turn on or off the system flags. The following is an example of a Conditional Flag command:

```
*IF CHANGED THEN COMPILE
```

As described in a subsequent paragraph, the CHANGED flag is turned on by the EDIT command when the module is edited. The command in the example turns on the BODY flag also for those modules. It also turns on the DECLARATION flag when it is off.

User flags may be defined to support an overlay structure allowing the user to specify appropriate processing for overlays. The following are examples of the use of the SETFLAG command to define user flags:

```
*SETFLAG OVRLAY1 OVERLAY 1 MODULE
*SETFLAG OVRLAY2 OVERLAY 2 MODULE
```

Flag commands turn on the flags in the appropriate modules:

```
*OVRLAY1 READIN ALL
*OVRLAY2 PRINT ALL
```

The result of these commands is that user flag OVRLAY1 is turned on in module READIN and its descendants and user flag OVRLAY2 is turned on in module PRINT and its descendants. The command in the following example would cause CONFIG to collect overlay 2 modules in file OBJECT:

```
*IF OVRLAY2 THEN COLLECT
```

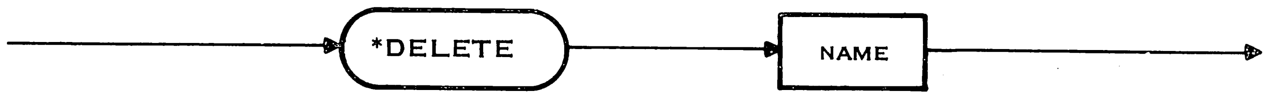
**12.3.7 MODIFYING A PROCESS CONFIGURATION.** The examples in this section show the building of a process configuration and using it for additional processing. However, the user may modify the current process configuration in several ways. ADD commands (paragraph 12.3.3.2) may be used to add modules to the program structure defined in the process configuration. DELETE commands may be used to delete nodes, and MOVE commands may be used to modify the structure by moving nodes to other points in the structure. The DISPLAY command may be used to display the process configuration. Object locations may be specified for nodes with USE OBJECT commands, and source locations may be specified or changed with USE commands. Default libraries for source and object modules may be changed with DEFAULT SOURCE and DEFAULT OBJECT commands.

**12.3.7.1 DELETE Command.** The DELETE command deletes a module and its descendants, if any, from the current process configuration. The syntax of the command is as follows:

<delete command> ::= * DELETE<name>

The syntax diagram is as follows:

Delete command:



The name parameter is the name of the node to be deleted. When the named node has descendants, the descendants are deleted also.

Example:

```
* DELETE READANDP
```

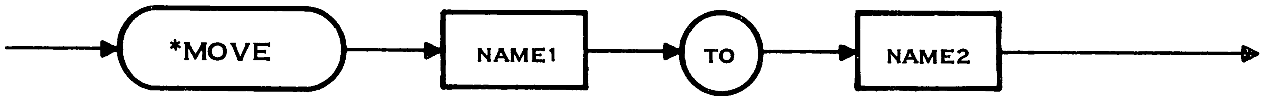
The example command deletes node READANDP, and its descendants, GETLINE and PRINTLAB. It could be followed by one or more ADD commands to change the names of these nodes of the process configuration.

**12.3.7.2 MOVE Command.** The MOVE command moves a module and all of its descendants to become a son of another module. The syntax of the command is as follows:

<move command> ::= *MOVE<name1>TO<name2>

The syntax diagram is as follows:

Move Command:



The name1 parameter is the name of a node to be moved. The name2 parameter is the name of another node in the structure. Name2 may not be a descendant of name1. The node specified as name1 and all its descendants, if any, are moved. The node specified as name1 becomes a son of the node specified as name2. The MOVE commands implies changes in the declarations of routines within the source code.

Example:

```
*MOVE GETLINE TO LABELS
```

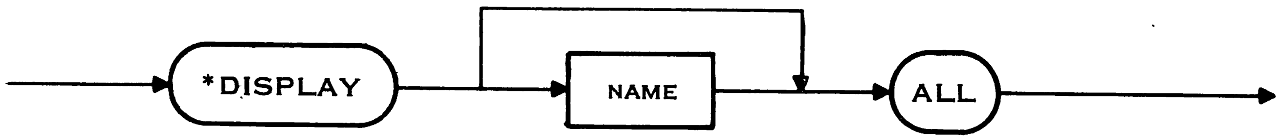
The example command moves node GETLINE (a son of READANDP) to become a son of LABELS. The forward declaration of GETLINE in the declaration portion of READANDP would have to be moved to the declaration portion of LABELS in order to compile any module correctly.

**12.3.7.3 DISPLAY Command.** The DISPLAY command displays the tabular representation of all or part of the current process configuration on file CRTFILE. The syntax is as follows:

<display command> ::= *DISPLAY[<name>]ALL

The syntax diagram is as follows:

Display command:



When the optional name parameter is omitted, the entire current process configuration is displayed. When the name parameter is included, the name is the name of a node, and the portion of the process configuration that lists the named node and its descendants is displayed.

The DISPLAY command allows the user to display the process configuration to note the effect of the commands that have been processed.

Example:

```
*DISPLAY READANDP ALL
```

The resulting display of the current process configuration includes nodes READANDP, GETLINE, and PRINTLAB.

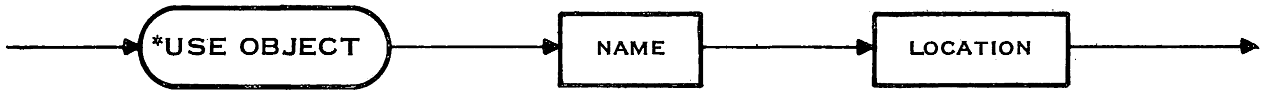
The format of the display is similar to that written to file OUTPUT, shown in figure 12-9. The names of the nodes are displayed, indented to show the program structure. The source and object locations are displayed, and the numbers of system flags that are not in their initial states are displayed. The initial states of flags 6 and 8 (SPLIT and CHECK) are on; the initial states of the other system flags are off. Except for flags 6 and 8 the display of the flag number indicates that it is on. An asterisk (*) is displayed with number 6 and 8 because the display of either of these numbers means that the flag is off.

**12.3.7.4 USE OBJECT Command.** The USE OBJECT command specifies a location for the object module of a specified node. The syntax of the command is as follows:

<use object command> ::= *USE OBJECT<name><location>

The syntax diagram is as follows:

Use object command:



The name parameter is the name of the node to which the object location applies. The location parameter is in the format defined in paragraph 12.3.3.1 for the BUILD PROCESS command. When the library is not specified in the location parameter, the location for the object module remains unspecified.

Example:

```
*USE OBJECT INTERACT <OBJL1, INTERACT>
```

The example specifies that the object module for node INTERACT is to be cataloged as member INTERACT of the library whose synonym is OBJL1.

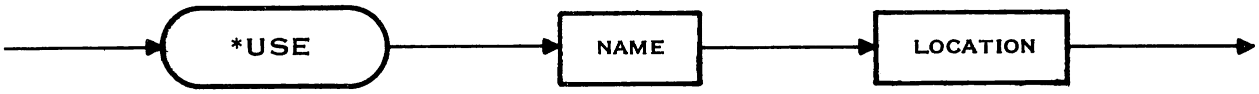
When a USE OBJECT command specifies a member name or a library synonym and member name for the object module for a node, CONFIG writes the module to the library member when it performs the deferred processing. Otherwise CONFIG writes the module to the default object library using the node name as the member name. The representation of the process configuration does not contain an object location unless a USE OBJECT or DEFAULT OBJECT command has been entered.

**12.3.7.5 USE Command.** The USE command specifies a location for the source module for a specified node. The syntax of the command is as follows:

<use command> ::= *USE <name> <location>

The syntax diagram is as follows:

Use command:



The name parameter is the name of the node to which the source location applies. The location parameter is in the format defined in paragraph 12.3.3.1 for the BUILD PROCESS command.

Example:

```
*USE GETLINE <SLIB1,INPLIN>
```

The example specifies that the source module for node GETLINE is to be cataloged as member INPLIN of the library whose synonym is SLIB1.

Either a USE command or an ADD command may be used to specify a source library synonym and member name for a source module. The USE command may be used to assign a different library synonym and/or member name.

**12.3.8 LIBRARIES.** The following library synonyms are initially defined in CONFIG:

| | |
|---|---|
| • MASTER | Intended for source modules of tested (fully developed) programs. |
| • OBJLIB | Intended for object modules corresponding to source modules in MASTER. |
| • LIBRARY | Intended for source modules of programs under development. |
| • ALTOBJ | Intended for object modules corresponding to source modules in LIBRARY. |

The default source library synonym LIBRARY is the logical default because it is intended for programs under development. Similarly, the default object library synonym ALTOBJ is appropriate because it is intended for object modules corresponding to the source modules in LIBRARY. Either default value may be changed using the DEFAULT SOURCE or DEFAULT OBJECT commands (paragraphs 12.3.29 and 12.3.30).

CONFIG maintains a library table in the process configuration. The first four entries in the table are the initially defined library synonyms MASTER, LIBRARY, OBJLIB, and ALTOBJ. When a library synonym is entered in any of the commands that may include a library synonym parameter, the synonym is added to the library table (unless it already appears in the table). The commands are BUILD PROCESS, CAT PROCESS, USE PROCESS, ADD, USE, USE OBJECT, DEFAULT SOURCE, DEFAULT OBJECT, SETLIB and EDIT.

Other synonyms may be substituted for the initially defined library synonyms. The MASTER command specifies a library synonym to replace MASTER. Similarly, the LIBRARY, OBJLIB, and ALTOBJ commands specify library synonyms to replace LIBRARY OBJLIB, and ALTOBJ, respectively.

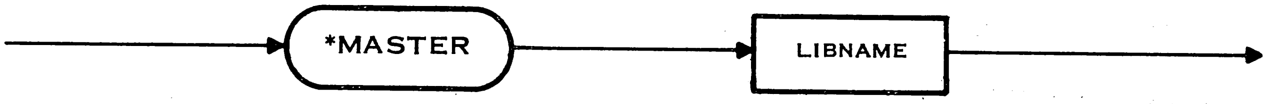
A library is identified to CONFIG using a library synonym, a synonym the value of which is the pathname of a library file. A synonym should be assigned using the appropriate operating system command prior to executing CONFIG. Alternatively, a synonym may be assigned by a SETLIB command.

**12.3.8.1 MASTER Command.** The MASTER command specifies a library synonym to replace the initially defined library synonym MASTER. The syntax of the command is as follows:

<master command> ::= *MASTER <libname>

The syntax diagram is as follows:

Master command:



The libname parameter is a library synonym that replaces synonym MASTER as the first entry in the library table. When the MASTER command is used, it should precede the ADD commands that define the nodes of the process configuration.

Example:

```
*MASTER SRCLIB1
```

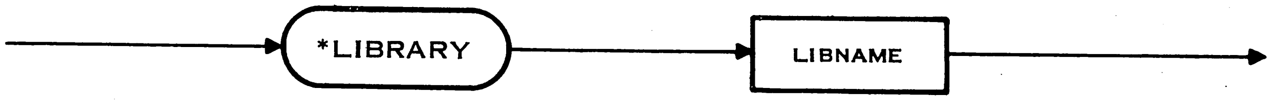
The example command replaces MASTER as the library synonym in the first entry of the library table with SRCLIB1.

**12.3.8.2 LIBRARY Command.** The LIBRARY command specifies a library synonym to replace the initially defined library synonym LIBRARY. The syntax of the command is as follows:

<library command> ::= *LIBRARY <libname>

The syntax diagram is as follows:

Library command:



The libname parameter is a library synonym that replaces synonym LIBRARY as the second entry in the library table. When the LIBRARY command is used, it should precede the ADD commands that define the nodes of the process configuration.

Example:

```
*LIBRARY SRCLIB2
```

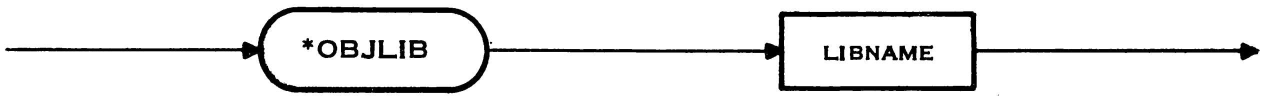
The example command replaces LIBRARY as the library synonym in the second entry of the library table with SRCLIB2.

**12.3.8.3 OBJLIB Command.** The OBJLIB command specifies a library synonym to replace the initially defined library synonym OBJLIB. The syntax of the command is as follows:

<objlib command> ::= *OBJLIB <libname>

The syntax diagram is as follows:

Objlib command:



The libname parameter is a library synonym that replaces synonym OBJLIB as the third entry in the library table. When the OBJLIB command is used, it should precede the ADD commands that define the nodes of the process configuration.

Example:

```
*OBJLIB OBJLIB1
```

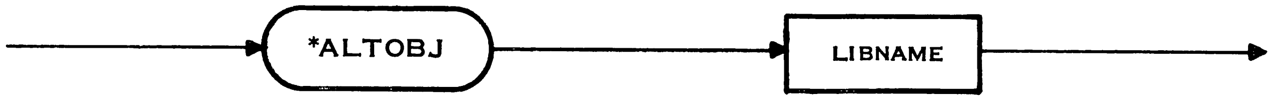
The example command replaces OBJLIB as the library synonym in the third entry of the library table with OBJLIB1.

**12.3.8.4 ALTOBJ Command.** The ALTOBJ command specifies a library synonym to replace the initially defined library synonym ALTOBJ. The syntax of the command is as follows:

<altobj command> ::= *ALTOBJ <libname>

The syntax diagram is as follows:

Altobj command:



The libname parameter is a library synonym that replaces synonym ALTOBJ as the fourth entry in the library table. When the ALTOBJ command is used, it should precede the ADD commands that define the nodes of the process configuration.

Example:

```
*ALTOBJ OBJLIB2
```

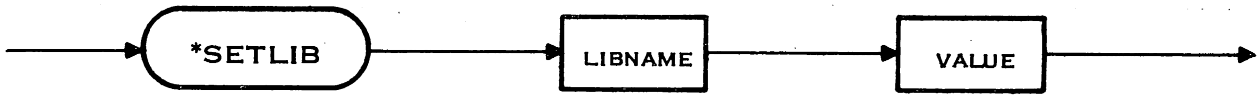
The example command replaces ALTOBJ as the library synonym in the fourth entry of the library table with OBJLIB2.

**12.3.8.5 SETLIB Command.** The SETLIB command defines a library synonym and assigns a value to the synonym. The syntax of the command is as follows:

<setlib command> ::= *SETLIB <libname><value>

The syntax diagram is as follows:

Setlib command:



The libname parameter is a library synonym that meets the requirements for a TIP identifier (paragraph 3.2.1). However, only the first eight characters are used by CONFIG. The value is the pathname of the library file to which the synonym applies.

Example:

```
*SETLIB SRCLIB3   DSC2.PASCAL.SOURCE.GARY
```

The example command adds library synonym SRCLIB3 to the library table (unless it is in the table already) and accesses the operating system to assign the value DSC2.PASCAL.SOURCE.GARY to synonym SRCLIB3.

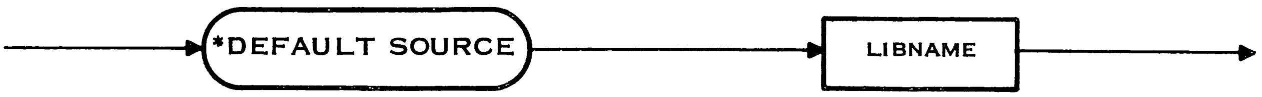
The SETLIB command should be used with caution to prevent defining too many synonyms for the library file pathname. Depending on the operating system, the synonym may have to be redefined to the operating system when the process configuration is accessed by a future run of CONFIG.

**12.3.8.6 DEFAULT SOURCE Command.** The DEFAULT SOURCE command specifies the library synonym for the default source library. The synonym applies to modules defined by subsequent ADD commands. The syntax of the command is as follows:

<default source command> ::= *DEFAULT SOURCE <libname>

The syntax diagram is as follows:

Default source command:



The libname parameter is the synonym having the pathname of the library as its value. Synonym MASTER is the initially defined alternate source library synonym; however any library synonym may be used. The DEFAULT SOURCE command does not alter the stored process configuration; it only applies to the current run.

Example:

```
*DEFAULT SOURCE SLIB1
```

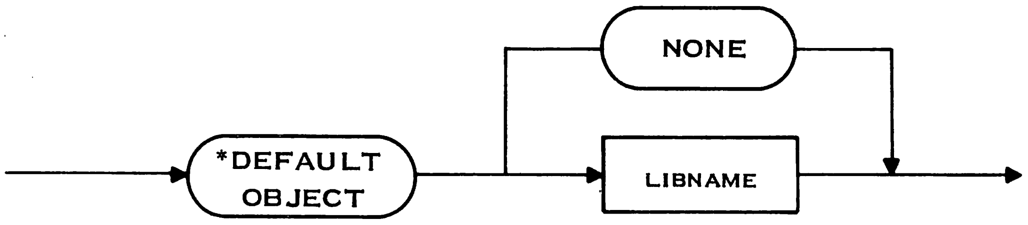
The example command specifies library synonym SLIB1 as the default source library. Source locations that do not explicitly include a library synonym use SLIB1 as the library synonym until the default value is changed by another DEFAULT SOURCE command. The default source library synonym is LIBRARY until the first DEFAULT SOURCE command is entered.

**12.3.8.7 DEFAULT OBJECT Command.** The DEFAULT OBJECT command specifies the library synonym for the default object library. The syntax of the command is as follows:

```
<default object command> ::=  *DEFAULT OBJECT <libname> |
                              *DEFAULT OBJECT NONE
```

The syntax diagram is as follows:

Default object command:



The libname parameter is the synonym having the pathname of the library as its value. Synonym OBJLIB is the initially defined alternate object library synonym; however, any library synonym may be used.

When keyword NONE is entered instead of a libname parameter, the initial default is restored. The DEFAULT OBJECT command does not alter the stored process configuration; it only applies to the current run.

Prior to the entry of a DEFAULT OBJECT command, and subsequent to the entry of a DEFAULT OBJECT NONE command, no object locations are shown in the representation of the process configuration on the OUTPUT file, and ALTOBJ is the default object library on which object modules are stored.

Example:

```
*DEFAULT OBJECT OBJL1
```

The example command specifies library synonym OBJL1 as the default object library. Object locations that do not explicitly include a library synonym use OBJL1 as the library synonym until the default value is changed by another DEFAULT OBJECT command.

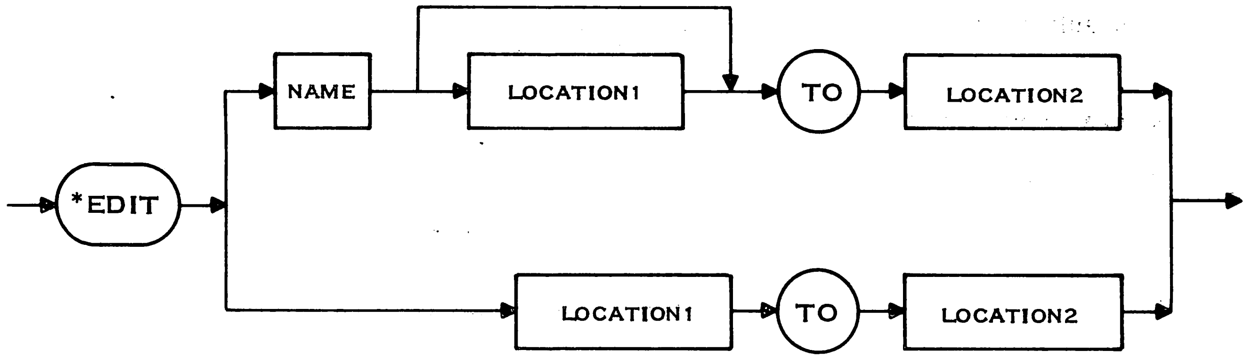
**12.3.9 TEXT EDITING.** CONFIG provides a line-oriented text editing capability for editing source modules, whether or not these modules apply to a node or nodes of the current process configuration. The EDIT command specifies the source module to be edited and copies the file with specified alterations to another location. Insert commands insert one or more source lines at a specified point and Replace commands replace specified source lines with one or more source lines.

**12.3.9.1 EDIT Command.** The EDIT command specifies a source module to be edited and copies the file with specified alterations to another location. The EDIT command for a source module that is specified as a node of the current process turns on the CHANGED flag for that node. The syntax for the command is as follows:

<edit command> ::= *EDIT [<name>][<location>] TO <location2>

The syntax diagram is as follows:

Edit command:



The name parameter is the node name corresponding to the source module to be edited. The location parameters are in the format defined in paragraph 12.3.3.1 for the BUILD PROCESS command. The source module at location1 is edited and the result is copied and cataloged at location2.

When the name parameter is used, the source module associated with the named node of the current process configuration is edited and the CHANGED flag for the node is turned on. The location for the source module of the node is changed to location2.

When the name parameter is omitted, the source module at location1 is edited. The source module may or may not be associated with the current process configuration; however the CHANGED flag is not set in either case.

Example:

```
*EDIT INTERACT TO <LIBRARY,NEWMOD>
```

This command specifies that the Insert and Replace commands that follow apply to the source module corresponding to node INTERACT. The edited module is cataloged as member NEWMOD of the library corresponding to library synonym LIBRARY and becomes the source module for node INTERACT. The source location for node INTERACT becomes LIBRARY, NEWMOD and the CHANGED flag is set for the node.

```
*EDIT <LIBRARY, ORIG> TO <SRCLIB1, NEWMOD>
```

This command specifies editing the source module at location <LIBRARY, ORIG> and cataloging the resulting module at location <SRCLIB1, NEWMOD>. The source module at location LIBRARY, ORIG may or may not be a node of a process configuration; the flags of process configuration nodes are not altered by this command.

**12.3.9.2 INSERT Command.** The Insert command specifies inserting one or more source lines in the source module identified in the preceding EDIT command. The lines to be inserted follow the command and the command parameter specifies the line in the source module after which the lines are inserted. The syntax of the command is as follows:

<insert command> ::= -<line>

The syntax diagram is as follows:

Insert command:

The line parameter is the line number of the line of the source module following which the subsequent lines are inserted. More than one Insert command may be supplied for editing a source module; these may be interspersed with Replace commands described in the next paragraph. These commands must be ordered by line number, in ascending numerical order.

Example:

- 40

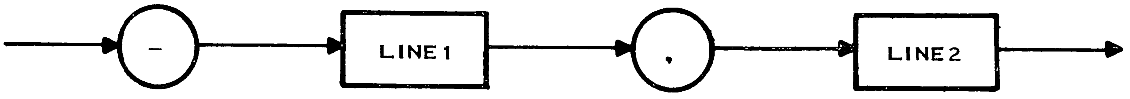
Insert the lines that follow this command in the source module after line 40.

**12.3.9.3 REPLACE Command.** The Replace command specifies replacing one or more source lines in the source module identified in the preceding EDIT command. The lines that replace the source module lines follow the command, and the command parameters specify the lines to be *replaced*. The syntax of the command is as follows:

<replace command> ::= -<line1>,<line2>

The syntax diagram is as follows:

Replace command:



The line1 and line2 parameters are the line numbers of lines in the source module. The range of source lines starting with line1 and extending through line2 are replaced by the lines that follow the Replace command. The number of replacing lines may be larger, smaller, or equal to the number of replaced lines. More than one Replace command may be entered for editing a source module; these may be interspersed with Insert commands previously described. These commands must be ordered by line number, in ascending numerical order.

Example:

```
-50, 70
```

Replace lines 50, 60, and 70 with the lines that follow this command.

**12.3.10 REQUIRED FILES.** CONFIG requires seven files, listed in table 12-2. When the files are not specified, CONFIG uses the default names. The default names consist of the file names (or the first six characters of file names longer than six characters) with the digits of the station number concatenated at the right. For example, the default name for the file supplied to the compiler is COMPFI08 when CONFIG is executed at terminal ST08.

**Table 12-2. Files Required for CONFIG**

| Name | I/O | Description |
|---|---|---|
| INPUT | I | Contains CONFIG commands. |
| OUTPUT | O | Contains CONFIG listings. |
| SYSMSG | O | Contains system messages. |
| COMPFILE | O | Contains source code selected by CONFIG for compilation. |
| CRTFILE | O | Contains input commands and error messages. |
| CPTEMP | I/O | Contains deferred processing commands. |
| OBJECT | I/O | Contains object file from CODEGEN and object file written by CONFIG. |

**12.3.11 EXECUTING CONFIG.** The Configuration Processor may be executed using either of two SCI procedures: XCONFIG and XCONFIGI. Procedure XCONFIG requests access names for all files; procedure XCONFIGI assigns the user's terminal for three of the files, for interactive entry of commands. Either procedure name may be entered at any time DX10 requests a command. When XCONFIG is entered, DX10 requests the following information:

```
COMMANDS
  CRT FILE
   LISTING
 MESSAGES
     MODE
   SOURCE
   OBJECT
  MEMORY
```

All but two of the items require access names of devices or files. The file names referred to are listed in table 12-2. The items are as follows:

- COMMANDS — The access name of a file (file name INPUT) that contains CONFIG commands or of a device at which commands are to be entered.

- CRT FILE — The access name of a device (file name CRTFILE) to display commands and error messages.
- LISTING — Access name of a device or file (file name OUTPUT) for listing.
- MESSAGES — Access name of a device or file (file name SYSMSG) for system messages.
- SOURCE — Access name of a file (file name COMPFILE) for source file output.
- OBJECT — Access name of a file (file name OBJECT) for object file.

The response to the MODE item is BATCH, FOREGROUND, or BACKGROUND, specifying the mode of execution. The background and foreground modes are as defined for the compiler, paragraph 11.5. The batch mode is described in a subsequent paragraph. The MEMORY item requires an ordered pair specifying stack and memory requirements as for the compiler.

When XCONFIGI is entered, DX10 requests the following information:

```
 LISTING
 SOURCE
  OBJECT
MEMORY
```

Items COMMAND, CRT FILE, and MESSAGES are assigned to ME, the synonym for the user's terminal. The MODE item is not requested; FOREGROUND is supplied by the system. Other items are identical to those for the XCONFIG procedure. The two procedures may be used interchangeably; when the command file, display file, and system message file would be assigned to the user's terminal, use XCONFIGI; otherwise, use XCONFIG.

The distinction between the run of CONFIG that prepares the source file for compilation and the run of CONFIG that splits the object file and catalogs the object modules is the command file. When the command file is the deferred processing command file written by a previous run of CONFIG (or a command file that contains those commands), the deferred processing is performed. Otherwise, the initial processing is performed.

The condition code $$CC is set by CONFIG to indicate the termination status as follows:

| | | |
|---|---|---|
| 0000 | — | Normal termination. |
| $4000_{16}$ | — | Warning conditions detected. |
| $6000_{16}$ | — | Errors detected. |
| $C000_{16}$ | — | Abnormal termination. |

**NOTE**

Refer to paragraph 11.5 for further information on $$CC.

Procedure XTIP does not execute CODEGEN when the SILT phases detect errors. When compiling a program that consists of many modules that are to be cataloged as a library by CONFIG, and CODEGEN is executed, the result may be that several of the resulting modules are error free, and would not need to be recompiled. Procedures XSILT and XCODE may be executed instead of procedure XTIP to execute CODEGEN unconditionally. These procedures may be used with CONFIG.

When XSILT is entered, DX10 requests the following items:

```
      SOURCE
     LISTING
    MESSAGES
        MEM1
        MEM2
        MODE
```

The first three items require access names of devices or files, as follows:

- SOURCE — The access name of the TIP source file.
- LISTING — The access name of a device or file for the source listing.
- MESSAGES — The access name of a device or file for system messages.

The next two items require ordered pairs specifying stack and memory requirements for the two phases of SILT. The default values are 6,10 for SILT1 and 13,4 for SILT2. The MODE is either FOREGROUND or BACKGROUND, as defined for the compiler, paragraph 11.5.

When XCODE is entered, DX10 requests the following items:

```
      OBJECT
     LISTING
    MESSAGES
         MEM
        MODE
```

The first three items require access names of devices or files; OBJECT is the access name of a file to which object code is written. The access names entered for the other two in the XSILT command apply in the XCODE command also. The MEM item is an ordered pair to specify stack and heap for CODEGEN; the default is 10,8. The MODE item should agree with that entered for XSILT, either FOREGROUND or BACKGROUND.

CONFIG may be executed using a batch stream and may usually be executed more easily in a batch stream. Figure 12-10 shows a sample batch stream to execute CONFIG, SILT1 and SILT2, and the deferred processing run of CONFIG. When executing this batch stream, the user should enter a WAIT command immediately following the XB command; the CONFIG commands are entered interactively at this point. Press the ENTER key at the terminal following the last command to terminate entry of commands and continue execution. The batch stream creates the required files on a user directory; enter the actual pathname of the directory in the first .SYN command.

Notice that the synonym LIBRARY is assigned to the pathname of the user directory with .SRC concatenated at the right. This will work only if the source modules being supplied as input to CONFIG have been cataloged as members of a library having that pathname or if the locations in the process configuration do not use the default library synonym, LIBRARY. The location parameters of the BUILD PROCESS, ADD, and CAT PROCESS commands may be used to specify pathnames other than those supplied in the batch stream, as required. Alternatively, file names may be modified to be compatible with the batch stream.

```
BATCH
*       SAMPLE BATCH STREAM TO PERFORM A CONFIG / COMPILE / CONFIG
*       SEQUENCE.
*
*             DELETE TIP "SECRET" SYNONYMS TO RELEASE SPACE
*             IN THE SYNONYM TABLE.
*
 P$SYN
*
*             ASSIGN LIBRARY SYNONYMS.
*
 .SYN   USER    = whatever your user directory is
 .SYN   MASTER  = USER.SRC
 .SYN   LIBRARY = USER.SRC
 .SYN   OBJLIB  = USER.OBJ
 .SYN   ALTOBJ  = USER.OBJ
*
*             EXECUTE CONFIGURATION PROCESSOR INTERACTIVELY
*             TO STAGE THE TIP SOURCE FOR COMPILATION.
*
 XCONFIGI     LISTING = USER.CF1
*
*             EXECUTE THE COMPILER.
*
 XSTLT        LISTING = USER.CM1, MESSAGES = USER.CM2
 XCODE
*
*             EXECUTE THE CONFIGURATION PROCESSOR TO SPLIT
*             THE OBJECT MODULES.
*
 XCONFIG      LISTING = USER.CF2, MESSAGES = USER.CF3
*
*             BUILD A SINGLE LISTING FILE.
*
 CC     INPUT   = (USER.CF1,USER.CM1,USER.CM2,USER.CF2,USER.CF3),
        OUTPUT  = USER.LIST,
        REPLACE = YES
*
*             DELETE TEMPORARY FILES (IF DESIRED).
*
 DF     PATHNAME = USER.CF1
 DF     PATHNAME = USER.CM1
 DF     PATHNAME = USER.CM2
 DF     PATHNAME = USER.CF2
 DF     PATHNAME = USER.CF3
EBATCH
```

**Figure 12-10. Batch Stream for Separate Compilation**

## 12.4 SPLIT PROGRAM UTILITY

The split program utility SPLITPGM divides a TIP source program into modules and catalogs these modules as members of a library file. SPLITPGM also writes an input command file for CONFIG which contains the commands required to build the process configuration corresponding to the original source program structure.

A program should meet the requirements for submission to CONFIG (paragraph 12.3.2) because the normal use of the library provided by SPLITPGM is as input to CONFIG. Specifically, SPLITPGM requires the forward declarations and the indentations that CONFIG also requires. The required indentations may most easily be provided by submitting the source code to NESTER prior to executing SPLITPGM.

Adding the forward declarations to the output of NESTER may be done using the text editor. The listing of the output of NESTER in figure 10-2 includes the following procedure declaration on line 420:

```
PROCEDURE GETLINE (VAR THISLINE : LINE);
```

This becomes a forward declaration when the keyword FORWARD is added, as follows:

```
PROCEDURE GETLINE (VAR THISLINE : LINE); FORWARD;
```

The declarations and statements for the procedure require a heading that does not include the parameter list. This may be added as a comment to promote documentation of the program, as follows:

```
PROCEDURE GETLINE; (*VAR THISLINE : LINE*)
```

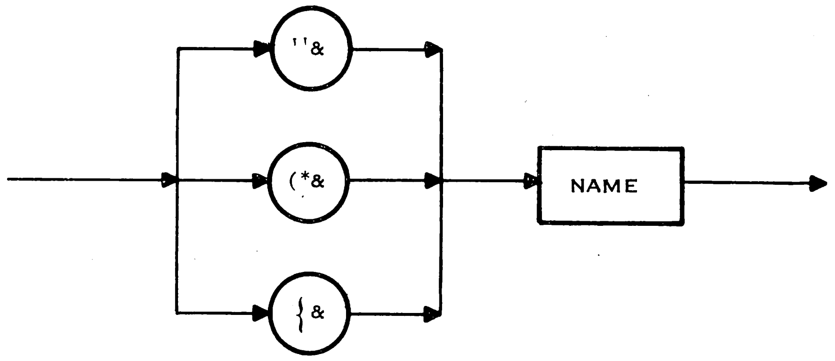
Adding forward declarations for functions is similar. The keyword FORWARD follows the parameter list and the function result type. The routine heading includes only the function name, not the parameter list and function type.

**12.4.1 SPLIT PROGRAM COMMAND.** The Split Program Command specifies the node name for a source module at the beginning of each module. The syntax of the command is as follows:

<split program command> ::= "& <name> | (*& <name>|"{ " & <name>

The syntax diagram is as follows:

Split program command:



The name parameter is the name that SPLITPGM supplies in the command file as the node name. Unless the CHECK flag for each node is turned off, CONFIG checks that the node name is identical to the IDT of the source module. Therefore the name parameter in the command should be the IDT of the module. All name parameters for the modules of a program should be unique in the first six characters because SPLITPGM uses the names as the names of members of LIBRARY. The split program command for a module must precede the heading (PROGRAM, PROCEDURE, or FUNCTION heading) of the module and any comments or option comments that precede the heading.

Example:

```
"& GETLINE
```

This command should be inserted between the forward declaration and the heading of procedure GETLINE.

**12.4.2 INPUT EXAMPLE.** The result of changing procedure headings to forward declarations and adding procedure headings and split program commands in the example program is shown in figure 12-10.

**12.4.3 LIBRARY AND FILES.** SPLITPGM catalogs the souce modules defined by the split program commands as members of a library file identified by library synonym LIBRARY. The user must assign the pathname of the desired library to synonym LIBRARY.

The files used by SPLITPGM are:

| | | |
|---|---|---|
| INPUT | I | Contains TIP program to be split. |
| OUTPUT | O | Contains CONFIG commands to build process configuration. |
| SYSMSG | O | Contains system messages. |

**12.4.4 EXECUTION.** The XPT procedure may be used for executing the split program utility SPLITPGM. Enter XPT when DX10 requests a command and enter the following information in response to the requests of XPT:

```
     PROGRAM FILE: .TIP.PROGRAM
  TASK NAME OR ID: SPLITPGM
            INPUT: <input source file>
           OUTPUT: <output file>
         MESSAGES: ME
             MODE: FOREGROUND
           MEMORY: 2,2
```

The input source file is the access name of the file to be split. It consists of the source file with commands inserted at the proper places (paragraph 12.4). The output file is the access name for the file to which SPLITPGM writes CONFIG commands to build the process configuration for the program to be split. The modules are cataloged as members of the library the pathname of which has been assigned to synonym LIBRARY.

SPLITPGM divides the source program into modules as defined by the commands and the BEGIN and END keywords that are the limits of the statement portions of the modules. The contents of the resulting module may be listed by submitting a *LIST ALL command to the configuration processor. Figure 12-8 shows the contents of each source module of the example program shown in figure 12-11.

The following batch stream executes SPLITPGM and CONFIG to build the process configuration fot the program being split:

```
BATCH
.SYN LIBRARY = "<pathname>"
XPT PF=".TIP.PROGRAM", T="SPLITPGM", I="<input source file>",
O="<output file>", MEM="2,2"
XCONFIG COMMANDS="<output file>", LISTING="<listing file>"
EBATCH
```

The pathname is the pathname of the directory of the library to which the source library is to be written. The listing file is the access name of a device or file for the CONFIG listing. The input source file and the output file are as defined previously.

**12.5 SPLIT OBJECT UTILITY**

The split object utility SPLIT divides the object modules in an object file into members of an object library. This function is performed automatically by CONFIG when the object file is input to the deferred processing run of CONFIG. SPLIT may be executed to perform the same function on an object file written by CODEGEN and not submitted to a deferred processing run of CONFIG.

The library synonym that SPLIT uses for the object library is OBJLIB.

SPLIT uses the following files:

| | | |
|---|---|---|
| OBJECT | I | Contains object file to be split. |
| OUTPUT | O | Contains error messages. |
| SYSMSG | O | Contains system messages. |

TIP software includes an SCI procedure, XSPLIT, for executing the split object program, SPLIT. Enter XSPLIT at any time DX10 requests a command. DX10 requests the following information:

```
     OBJECT FILE
OBJECT DIRECTORY
   ERROR LISTING
        MESSAGES
            MODE
```

Three of the items require access names of devices or files, as follows:

- OBJECT FILE — The access name of the object file as written by the compiler.
- ERROR LISTING — The access name of the device or file for the error listing.
- MESSAGES — The access name of the device or file for the system messages.

```
"&LABELS
PROGRAM LABELS;                                                          000010
(*-----------------------------------------------------------------------000020
   PROGRAM LABELS.                                                       000030
   PURPOSE :     THIS PROGRAM READS AN ADDRESS LABEL AND PRINTS MULTIPLE 000040
                 COPIES OF THAT LABEL.                                   000050
   FILES USED : INPUT   - FOR USER-SUPPLIED PARAMETERS AND THE LABEL     000060
                CRTFIL  - USED FOR PROMPTING INPUT                       000070
                OUTPUT  - MULTIPLE COPIES OF THE LABEL                   000080
   PROCEDURES CALLED : INTERACT, READANDPRINT                            000090
-----------------------------------------------------------------------*)000100
VAR CRTFIL : TEXT ;                              (*USED TO PROMPT INPUT*) 000110
    CHARSPERLINE : INTEGER;           (*NUMBER OF CHARACTERS PER LINE*)  000120
    LINESPERLABEL : INTEGER;              (*NUMBER OF LINES PER LABEL*)  000130
    COPYCOUNT : INTEGER;                  (*NUMBER OF COPIES TO PRINT*)  000140
PROCEDURE INTERACT; FORWARD;
"&INTERACT
PROCEDURE INTERACT;                                                      000150
(*-----------------------------------------------------------------------000160
  PROCEDURE INTERACT;                                                    000170
  PURPOSE : INTERACT PROMPTS THE USER, REQUESTING CERTAIN INPUTS.        000180
  OUTPUTS : CHARSPERLINE - NUMBER OF CHARACTERS PER LINE                 000190
            LINESPERLABEL - NUMBER OF LINES PER LABEL                    000200
            COPYCOUNT - NUMBER OF LABELS TO PRINT                        000210
-----------------------------------------------------------------------*)000220
BEGIN                                                      (*INTERACT*)  000230
  REWRITE( CRTFIL );                                                     000240
  WRITELN( CRTFIL, 'HOW MANY CHARACTERS PER LINE?' );                    000250
  RESET(INPUT); READ( CHARSPERLINE );                                    000260
  WRITELN( CRTFIL, 'HOW MANY LINES PER LABEL?' );                        000270
  READLN; READ( LINESPERLABEL );                                         000280
  WRITELN( CRTFIL, 'HOW MANY LABELS?' );                                 000290
  READLN; READ( COPYCOUNT ); WRITELN(CRTFIL, 'NOW INPUT THE LABEL');     000300
END;                                                       (*INTERACT*)  000310
PROCEDURE READANDPRINT; FORWARD;                                         000320
"&READANDPRINT
PROCEDURE READANDPRINT;                                                  000320
(*-----------------------------------------------------------------------000330
  PROCEDURE READANDPRINT;                                                000340
  PURPOSE : READANDPRINT READS A LABEL AND PRINTS MULTIPLE COPIES OF IT.000350
  PROCEDURES CALLED : GETLINE, PRINTLABEL                                000360
-----------------------------------------------------------------------*)000370
TYPE                                                                     000380
   LINE = PACKED ARRAY (.1..CHARSPERLINE.) OF CHAR;                      000390
VAR                                                                      000400
   LABELIMAGE : ARRAY (.1..LINESPERLABEL.) OF LINE;                      000410
PROCEDURE GETLINE(VAR THISLINE : LINE): FORWARD;                         000420
"&GETLINE
PROCEDURE GETLINE(*VAR THISLINE : LINE*);                                000420
(*-----------------------------------------------------------------------000430
  PROCEDURE GETLINE(VAR THISLINE : LINE);                                000440
  PURPOSE : GETLINE READS A SINGLE LINE OF A LABEL.                      000450
  INPUTS :  CHARSPERLINE - NUMBER OF CHARACTERS PER LINE                 000460
  OUTPUTS : THISLINE - THE LINE THAT WAS READ.                           000470
```

**Figure 12-11. Example of Input to SPLITPGM (Sheet 1 of 2)**

```
------------------------------------------------------------------------*)
VAR CH : INTEGER;
BEGIN                                                      (*GETLINE*)
  READLN; CH := 1;
  WHILE CH <= CHARSPERLINE AND NOT EOLN(INPUT) DO BEGIN
    READ( THISLINE(.CH.) ); CH := CH + 1;
    END;                          (*FILL IN REST OF LINE WITH BLANKS*)
  FOR J := CH TO CHARSPERLINE DO THISLINE(.J.) := ' ';
END;                                                       (*GETLINE*)
PROCEDURE PRINTLABEL; FORWARD;
"&PRINTLABEL
PROCEDURE PRINTLABEL;
(*-----------------------------------------------------------------------
  PROCEDURE PRINTLABEL;
  PURPOSE : PRINTLABEL PRINTS ONE COPY OF THE LABEL.
  INPUTS  : LINESPERLABEL - NUMBER OF LINES PER LABEL
            CHARSPERLINE - NUMBER OF CHARACTERS PER LINE
            LABELIMAGE - THE LABEL TO BE PRINTED
------------------------------------------------------------------------*)
BEGIN                                                   (*PRINTLABEL*)
  FOR L := 1 TO LINESPERLABEL DO BEGIN
    FOR CH := 1 TO CHARSPERLINE DO WRITE( LABELIMAGE(.L.)(.CH.) );
    WRITELN; END;
END;                                                    (*PRINTLABEL*)
BEGIN                                                 (*READANDPRINT*)
  FOR L := 1 TO LINESPERLABEL DO GETLINE( LABELIMAGE(.L.) );
  FOR K := 1 TO COPYCOUNT DO PRINTLABEL;
END;                                                  (*READANDPRINT*)
BEGIN                                                       (*LABELS*)
  INTERACT; READANDPRINT;
END.                                                        (*LABELS*)
```

**Figure 12-11. Example of Input to SPLITPGM (Sheet 2 of 2)**

The OBJECT DIRECTORY is the pathname of the directory for the object library. The MODE item is either FOREGROUND or BACKGROUND, as described for the compiler, paragraph 11.5.

# SECTION XIII

# REVERSE ASSEMBLER

## 13.1 GENERAL

The reverse assembler (RASS) provides a Model 990 Computer Assembly Language source program that corresponds to an object module written by the TIP compiler. The output of RASS may be directly assembled. Submitting an object module to RASS to obtain the assembly language source code allows the user to perform manual optimization when appropriate. The assembly language source code allows debugging at the machine language level.

## 13.2 REQUIRED FILES

RASS requires the following files:

| | | |
|---|---|---|
| OBJECT | I | Contains the object file to be processed. |
| OUTPUT | O | Contains the assembly language listing. |
| SYSMSG | O | Contains system messages. |

## 13.3 EXECUTING THE REVERSE ASSEMBLER

TIP software includes an SCI procedure XRASS for executing the reverse assembler, RASS. Enter XRASS at any time DX10 requests a command. DX10 requests the following information:

```
  OBJECT
 LISTING
MESSAGES
    MODE
  MEMORY
```

The first three items require access names of devices or files, as follows:

- OBJECT — The access name of the object file (input).
- LISTING — The access name of the device or file for the listing (both errors and assembly language source).
- MESSAGES — The access name of the device or file for the system messages.

The MODE may be either FOREGROUND or BACKGROUND, as described for the compiler (paragraph 11.5). The MEMORY item is an ordered pair specifying stack and heap for the execution of RASS.

## 13.4 EXAMPLE LISTING

Figure 13-1 is an example of the assembly listing produced by RASS. The program listed consists of three modules. The listing for each module begins with the IDT directive that contains the module name. This is followed by a comment line that provides a heading for the columns at the right.

Following the heading are the PSEG directive, the data directives, the DEF directive, and the REF directives. The location counter value is shown for each data word and the contents also are shown, both in hexadecimal. When the data word represents a relocatable address, the hexadecimal value is followed by a plus sign (+). The representation of the word as a pair of ASCII characters is shown at the end of each line. Each unprintable character is shown as a period (.).

Following the directives, another comment provides a heading for the instructions that follow. For each instruction the listing shows the location counter value and the value of the word or words of the machine language instruction in hexadecimal format. The file that contains the listing (OUTPUT) may be used as a source code file for the assembler. The values that appear at the right end of most lines are treated as comments by the assembler.

RASS is specifically designed to process the object files written by CODEGEN and has limited application for processing other object code files. It recognizes only the object output tag characters and operation codes supplied by CODEGEN. The directives are supplied in accordance with TIP format, since object code seldom contains enough information to explicitly identify the directives. When processing object code that contains a tag character or operation code that RASS does not recognize, an error termination occurs. Even if the object code does not contain unrecognizable data, only those words that RASS correctly interprets as instructions will be correctly listed.

The first eight bytes of each module contain the module name (program, procedure, or function name). The next two bytes contain the nesting level: 1 for the main program, 2 for procedures and routines declared in the main program, etc. The TIP source program corresponding to the example in figure 13-1 is shown in figure 9-1. It consists of main program DIGIO that declares procedures CCHAR and CINT. Therefore the three modules in figure 13-1 are CCHAR and CINT at level 2 and mainprogram DIGIO at level 1.

Careful comparison of the source code in figures 9-1 and 13-1 shows how the TIP compiler implements Pascal statements in assembly language. Manual optimization of the assembly language source code may be performed by omitting any redundant source lines and reassembling the object module.

```
       IDT   'CCHAR   '
*                                            LC      HEX     CHAR
       PSEG
D0000  DATA >4343                            0000    4343    CC
       DATA >4841                            0002    4841    HA
       DATA >5220                            0004    5220    R
       DATA >2020                            0006    2020
       DATA >0002                            0008    0002    ..
D000A  DATA >0030                            000A    0030    .0
D000C  DATA >0039                            000C    0039    .9
       DATA L0072                            000E    0072+   ..
       DATA D0000                            0010    0000+   ..
       DEF  CCHAR
       REF  ENT$S
       REF  RET$S
*                                            LC      WORD(S)
CCHAR  EQU  $
       BL   @ENT$S                           0012    06A0    0000
       DATA >003A                            0016    003A
       MOV  @>0034(R9),R5                    0018    C169    0034
       CLR  *R5                              001C    04D5
       LI   R6,>0001                         001E    0206    0001
       MOV  @>0036(R9),@>0038(R9)            0022    CA69    0036    0038
L0028  EQU  $
       C    R6,@>0038(R9)                    0028    8A46    0038
       JGT  L0072                            002C    1522
       MOV  R6,R5                            002E    C146
       SLA  R5,1                             0030    0A15
       AI   R5,>0026                         0032    0225    0026
       A    R9,R5                            0036    A149
       C    *R5,@D000A                       0038    8815    000A+
       JLT  L006E                            003C    1118
       MOV  R6,R5                            003E    C146
       SLA  R5,1                             0040    0A15
       AI   R5,>0026                         0042    0225    0026
       A    R9,R5                            0046    A149
       C    *R5,@D000C                       0048    8815    000C+
       JGT  L006E                            004C    1510
       MOV  @>0034(R9),R5                    004E    C169    0034
       LI   R3,>000A                         0052    0203    000A
       MPY  *R5,R3                           0056    38D5
       MOV  R6,R5                            0058    C146
       SLA  R5,1                             005A    0A15
       AI   R5,>0026                         005C    0225    0026
       A    R9,R5                            0060    A149
       A    *R5,R4                           0062    A115
       AI   R4,>FFD0                         0064    0224    FFD0
       MOV  @>0034(R9),R5                    0068    C169    0034
       MOV  R4,*R5                           006C    C544
L006E  EQU  $
       INC  R6                               006E    0586
       JMP  L0028                            0070    10DB
L0072  EQU  $
       B    @RET$S                           0072    0460    0000
       END
       IDT  'CINT    '
*                                            LC      HEX     CHAR
       PSEG
D0000  DATA >4349                            0000    4349    CI
       DATA >4E54                            0002    4E54    NT
       DATA >2020                            0004    2020
       DATA >2020                            0006    2020
```

Figure 13-1. RASS Listing Example (Sheet 1 of 4)

```
          DATA >0002                                  0008    0002      ..
 D000A    DATA >0000                                  000A    0000      ..
          DATA L0066                                  000C    0066+     ..
          DATA D0000                                  000E    0000+     ..
          DEF   CINT
          REF   ENT$M
          REF   DIV$
          REF   CINT
          REF   PUTCH$
          REF   RET$M
*                                                      LC     WORD(S)
CINT      EQU   $
          BL    @ENT$M                                0010    06A0      0000
          DATA >002C                                  0014    002C
          MOV   @>0002(R12),R15                       0016    C3EC      0002
          MOV   @>0028(R9),R8                         001A    C229      0028
          LI    R12,>000A                             001E    020C      000A
          BL    @DIV$                                 0022    06A0      0000
          MOV   R7,@>002A(R9)                         0026    CA47      002A
          C     @>002A(R9),@D000A                     002A    8829      002A      000A+
          JEQ   L003E                                 0030    1306
          MOV   @>002A(R9),@>0028(R10)                0032    CAA9      002A      0028
          LI    R11,CINT                              0038    020B      0000
          BLWP  R10                                   003C    040A
L003E     EQU   $
          MOV   @>0028(R9),R8                         003E    C229      0028
          LI    R12,>000A                             0042    020C      000A
          BL    @DIV$                                 0046    06A0      0024
          MOV   R8,R6                                 004A    C188
          AI    R6,>0030                              004C    0226      0030
          MOV   R6,R8                                 0050    C206
          LI    R12,>0100                             0052    020C      0100
          BL    @DIV$                                 0056    06A0      0048
          MOV   R8,R5                                 005A    C148
          MOV   @>0040(R15),R12                       005C    C32F      0040
          MOV   R5,R7                                 0060    C1C5
          BL    @PUTCH$                               0062    06A0      0000
L0066     EQU   $
          B     @RET$M                                0066    0460      0000
          END
          IDT   'DIGIO    '
*                                                      LC      HEX     CHAR
          PSEG
D0000     DATA >4449                                  0000    4449      DI
          DATA >4749                                  0002    4749      GI
          DATA >4F20                                  0004    4F20      O
          DATA >2020                                  0006    2020
D0008     DATA >0001                                  0008    0001      ..
D000A     DATA >0060                                  000A    0060      ..
D000C     DATA >494E                                  000C    494E      IN
          DATA >5055                                  000E    5055      PU
          DATA >5420                                  0010    5420      T
          DATA >2020                                  0012    2020
D0014     DATA D000C                                  0014    000C+     ..
D0016     DATA >0002                                  0016    0002      ..
D0018     DATA >0013                                  0018    0013      ..
D001A     DATA >454E                                  001A    454E      EN
          DATA >5445                                  001C    5445      TE
          DATA >5220                                  001E    5220      R
          DATA >3120                                  0020    3120      1
          DATA >544F                                  0022    544F      TO
          DATA >2035                                  0024    2035       5
```

Figure 13-1. RASS Listing Example (Sheet 2 of 4)

```
         DATA >2044                             0026   2044    D
         DATA >4947                             0028   4947    IG
         DATA >4954                             002A   4954    IT
         DATA >5300                             002C   5300    S.
D002E    DATA D001A                             002E   001A+   ..
D0030    DATA >008E                             0030   008E    ..
D0032    DATA >0019                             0032   0019    ..
         DATA L0112                             0034   0112+   ..
         DATA D0000                             0036   0000+   ..
         DEF  PSCL$$
         REF  ENT$1
         REF  MOV$4
         REF  FL$INI
         REF  WRS$T
         REF  WRLN$
         REF  REST$T
         REF  EOLN$
         REF  GET$CH
         REF  MOV$6
         REF  CCHAR
         REF  CINT
         REF  CLS$
         REF  RET$1
*                                                LC    WORD(S)
PSCL$$   EQU  $
         BL   @ENT$1                            0038   06A0    0000
         DATA >0090                             003C   0090
         MOV  R9,@>0028(R10)                    003E   CA89    0028
         A    @D000A,@>0028(R10)                0042   AAA0    000A+   0028
         MOV  @D0014,R7                         0048   C1E0    0014+
         MOV  R10,R8                            004C   C20A
         AI   R8,>002A                          004E   0228    002A
         BL   @MOV$4                            0052   06A0    0000
         MOV  @D0008,@>0032(R10)                0056   CAA0    0008+   0032
         MOV  @D0016,@>0034(R10)                005C   CAA0    0016+   0034
         LI   R11,FL$INI                        0062   020B    0000
         BLWP R10                               0066   040A
         MOV  @>0040(R9),@>0028(R10)            0068   CAA9    0040    0028
         MOV  @D0018,@>002A(R10)                006E   CAA0    0018+   002A
         MOV  @D0018,@>002C(R10)                0074   CAA0    0018+   002C
         MOV  @D002E,@>002E(R10)                007A   CAA0    002E+   002E
         LI   R11,WRS$T                         0080   020B    0000
         BLWP R10                               0084   040A
         MOV  @>0040(R9),@>0028(R10)            0086   CAA9    0040    0028
         LI   R11,WRLN$                         008C   020B    0000
         BLWP R10                               0090   040A
         MOV  @>0060(R9),@>0028(R10)            0092   CAA9    0060    0028
         LI   R11,REST$T                        0098   020B    0000
         BLWP R10                               009C   040A
         INC  @>008C(R9)                        009E   05A9    008C
L00A2    EQU  $
         MOV  @>0060(R9),R12                    00A2   C329    0060
         BL   @EOLN$                            00A6   06A0    0000
         JGT  L00CA                             00AA   150F
         MOV  @>008C(R9),R15                    00AC   C3E9    008C
         SLA  R15,1                             00B0   0A1F
         AI   R15,>007E                         00B2   022F    007E
         A    R9,R15                            00B6   A3C9
         MOV  @>0060(R9),R12                    00B8   C329    0060
         BL   @GET$CH                           00BC   06A0    0000
         SWPB R7                                00C0   06C7
         MOV  R7,*R15                           00C2   C7C7
```

**Figure 13-1. RASS Listing Example (Sheet 3 of 4)**

```
         INC   @>008C(R9)                     00C4   05A9   008C
         JMP   L00A2                          00C8   10EC
L00CA    EQU   $
         DEC   @>008C(R9)                     00CA   0629   008C
         MOV   R9,R7                          00CE   C1C9
         AI    R7,>0080                       00D0   0227   0080
         MOV   R10,R8                         00D4   C20A
         AI    R8,>0028                       00D6   0228   0028
         BL    @MOV$6                         00DA   06A0   0000
         MOV   R9,@>0034(R10)                 00DE   CA89   0034
         A     @D0030,@>0034(R10)             00E2   AAA0   0030+   0034
         MOV   @>008C(R9),@>0036(R10)         00E8   CAA9   008C    0036
         LI    R11,CCHAR                      00EE   020B   0000
         BLWP  R10                            00F2   040A
         A     @D0032,@>008E(R9)              00F4   AA60   0032+   008E
         MOV   @>008E(R9),@>0028(R10)         00FA   CAA9   008E    0028
         LI    R11,CINT                       0100   020B   0000
         BLWP  R10                            0104   040A
         MOV   @>0040(R9),@>0028(R10)         0106   CAA9   0040    0028
         LI    R11,WRLN$                      010C   020B   008E
         BLWP  R10                            0110   040A
L0112    EQU   $
         MOV   R9,@>0028(R10)                 0112   CA89   0028
         A     @D000A,@>0028(R10)             0116   AAA0   000A+   0028
         LI    R11,CLS$                       011C   020B   0000
         BLWP  R10                            0120   040A
         B     @RET$1                         0122   0460   0000
         END
```

**Figure 13-1. RASS Listing Example (Sheet 4 of 4)**

# SECTION XIV

# LINKING AND EXECUTING PASCAL TASKS

## 14.1 GENERAL

This section describes the runtime options provided for TIP and the linking procedures required for these options. This section also describes the procedures for executing Pascal tasks under DX10, TX990, and RX990, including the debug mode. Finally, the section discusses runtime errors and describes the abnormal termination dump provided to assist in debugging the errors.

## 14.2 TIP RUNTIME OPTIONS

The runtime options closely relate to the operating system under which the TIP task is to execute, as follows:

- Execution under DX10
- LUNO I/O capability under DX10
- Multitask capability under DX10
- Execution under TX990
- Multitask capability under TX990
- Execution under RX990
- Minimal runtime capability
- Stand-alone execution
- Shared procedure capability

Three runtime libraries are linked in various ways with the object code produced by the TIP compiler to provide the options. The libraries are .TIP.OBJ, .TIP.LUNOBJ, and .TIP.MINOBJ. Table 14-1 lists the characteristics of each library.

The runtime sizes shown in the descriptions of the options are minimum sizes; when the programs incorporate additional features of the language, additional runtime code is required. Table C-1 lists additional features and the amount of additional code they require. The user can estimate the size of the total runtime code by adding the sizes from table C-1 to the minimum sizes shown for each option. When library .TIP.MINOBJ is not used, those sizes marked with an asterisk are included in the minimum runtime code size shown.

**Table 14-1. TIP Runtime Libraries**

| Option | .TIP.OBJ Library | .TIP.LUNOBJ Library | .TIP.MINOBJ Library |
|---|---|---|---|
| I/O access | SCI synonyms | LUNOs | LUNOs |
| Files INPUT and OUTPUT | Predeclared | Predeclared | Optional |
| Post-mortem dump | Linking option | Linking option | Not available |
| Error diagnostics | English messages | English messages | Code numbers |
| Heap allocation | Allocated at startup | Allocated at startup | Optional; static allocation |
| Probe displays | Optional | Optional | Not available |
| Execute under DX10 | Yes | Yes | Yes |
| Execute under TX990 | No | Yes | Yes |
| Execute under RX990 | No | Yes | Yes |
| Bid using SCI or OCP | Yes | Yes | Yes |
| Bid by another task | No | Yes | Yes |
| Memory-resident task | No | Except under DX10 | Yes |

**14.2.1 EXECUTION UNDER DX10.** Execution of a TIP task under DX10 with full runtime support using SCI synonyms to define the I/O files and devices requires linking the Pascal runtime library .TIP.OBJ with the object code provided by the compiler. The required linking is described in paragraph 14.3.1, and the procedure for executing the task is described in paragraph 14.3.2. This procedure results in linking at least 19,700 bytes of runtime code with the task. There is an overhead of 224 bytes for each main program; this overhead includes stack space for the descriptors for predefined files INPUT and OUTPUT. Thus 20,000 bytes are required in addition to the object code for the task. Optionally, the dump routines may be omitted, reducing the size of the runtime code to 16,860 bytes.

**14.2.2 LUNO I/O.** Execution under DX10 with access to files and I/O devices by means of LUNOs requires linking the task with library .TIP.LUNOBJ. The linking procedure is described in paragraph 14.3.3, and the resulting linked object code is executed using the procedure described in paragraph 14.3.4. The result is that the overhead is somewhat less than with the .TIP.OBJ library. Specifically, the size of the minimum runtime code is approximately 16,200 bytes, or 13,360 bytes if the optional dump routines are not included. Programming considerations for this option are described in paragraph 15.5.

**14.2.3 MULTITASK CAPABILITY UNDER DX10.** Multitask capability under DX10 is supported by a set of task routines that are available on either library, .TIP.OBJ or .TIP.LUNOBJ. However, any task that is bid by another task using a task control routine must have been linked using library .TIP.LUNOBJ. Linking for a called task must be done as described in paragraph 14.3.3. When the initial task is linked using library .TIP.OBJ, the task is executed as described in paragraph 14.3.2; when the initial task is linked using library .TIP.LUNOBJ, the task is executed as described in paragraph 14.3.4. Similarly, the sizes of the runtime code depend on the linking procedure followed. The size information in either of the two preceding paragraphs applies, depending on the runtime library linked. Programming considerations for these tasks are described in paragraph 15.6.

**14.2.4 EXECUTION UNDER TX990.** TIP tasks compiled and linked under DX10 may be executed under TX990 (Release 2.3 or later). The runtime library .TIP.LUNOBJ must be linked with the object code as described in paragraph 14.3.5. The task is executed as described in paragraph 14.3.6. The size of the minimum runtime code is approximately 16,200 bytes, or 13,360 bytes if the optional dump routines are not included.

**14.2.5 MULTITASK CAPABILITY UNDER TX990.** Multitask applications compiled and linked under DX10 for execution under TX990 are also supported, with the limitation that only one task may use predeclared files INPUT and OUTPUT. The procedure for linking is described in paragraph 14.3.7, and the tasks are executed as described in paragraph 14.3.8. All tasks that are linked with the system use the same runtime code.

**14.2.6 EXECUTION UNDER RX990.** TIP tasks compiled and linked under DX10 may also be executed under RX990. The runtime library .TIP.LUNOBJ must be linked with the object code as described in paragraph 14.3.9. The resulting task is executed as described in paragraph 14.3.10. The size of the minimum runtime code is approximately 16,200 bytes, or 13,360 bytes if the optional dump routines are not included.

**14.2.7 MINIMAL RUNTIME CAPABILITY.** A third runtime library, .TIP.MINOBJ, allows the user to include only the runtime code actually required, resulting in the minimal size of runtime code to be linked to the object code compiled by the TIP compiler. Specifically, the minimal runtime library may be used for TIP tasks that do not need the following:

- Memory dump on abnormal termination
- End-action error handling
- Linkage to FORTRAN routines that perform I/O
- Stack and heap allocation at program initiation
- Access to SCI synonyms
- Error messages in English

The procedure for linking with library .TIP.MINOBJ is described in paragraph 14.3.11. A further reduction is possible when the predeclared files INPUT and OUTPUT are not required. This can be accomplished by omitting the main program routine, beginning execution in a procedure instead. Paragraph 15.4 describes the procedure for omitting the main program routine and designating a procedure for initial execution.

The runtime code resulting from use of library .TIP.MINOBJ is at least 3930 bytes in size. The reduction obtained by omitting the routines required for predeclared files INPUT and OUTPUT results in a minimum of 1000 bytes of runtime code. Of this, approximately 800 bytes may be shared by multiple tasks; each requires 200 bytes that are not shared with other tasks.

**14.2.8 STAND-ALONE EXECUTION.** The minimal runtime library is also used for tasks that execute without operating system support (stand-alone). These tasks are loaded by the ROM loader of the computer. Linking is described in paragraph 14.3.13, and execution is described in paragraph 14.3.14. Programming considerations are described in paragraph 15.7. The size of the runtime code for stand-alone programs is approximately 660 bytes. Use of arithmetic, set, runtime check, and heap management routines increases the size of runtime code to a larger value. The size of the stack region must also be added to get the total size of runtime code.

**14.2.9 SHARED PROCEDURE CAPABILITY.** The sharing of a procedure segment by several tasks is supported under the DX10 and RX990 operating systems. (In this context, the procedure segment is a reentrant module of executable code as defined for DX10 or RX990 rather than a Pascal routine.) The linking procedure for execution under DX10 or RX990 is described in paragraph 14.3.15. The size of the runtime code is not affected by the sharing of procedure segments; however, sharing procedure segments saves memory space that might otherwise be required for additional copies of the segment.

## 14.3 LINKING AND EXECUTING

This paragraph describes linking procedures for runtime libraries .TIP.OBJ, .TIP.LUNOBJ, and .TIP.MINOBJ, and for execution procedures for tasks linked by the procedures. Linking is performed by the link editor of DX10, which is executed using the SCI procedure XLE described in the *Model 990 Computer DX10 Operating System Release 3 Reference Manual, Volume IV*. The procedure requests the following information:

```
       CONTROL ACCESS NAME:
 LINKED OUTPUT ACCESS NAME:
       LISTING ACCESS NAME:
               PRINT WIDTH:
```

The CONTROL ACCESS NAME is the pathname of the link edit control file. The contents of this file differ according to the linking option required. Typical contents of the control file for each option are shown in the following paragraphs. The LINKED OUTPUT ACCESS NAME is the pathname of the file on which the linked output is to be written. The LISTING ACCESS NAME is the pathname of the file to which the link map listing is to be written. The PRINT WIDTH is the line length of the lines of the listing; the default value (80) is normally used.

**14.3.1 LINKING FOR DX10 EXECUTION USING SCI SYNONYMS.** The procedure in this paragraph applies to a TIP task to be executed under DX10. This task performs I/O operations using SCI synonyms for access to files and/or devices. The resulting task cannot be executed under TX990 or RX990, nor can it be bid by another task (a cooperating task).

The contents of the link edit control file for this type of linking operation are listed, with optional linking commands enclosed in brackets ([ ]). The file should contain the following:

```
[FORMAT <object format>;]      Optional format of linked object code
[LIBRARY <user library>;]      Optional user library
LIBRARY .TIP.OBJ;              Runtime library
PHASE 0, <task name>;          Associates task with name
INCLUDE (MAIN);                Specifies required portion of runtime code
INCLUDE <user object>;         TIP compiler output
END
```

The first command is optional. When included, it specifies the format of the linked object file produced by the link editor. If it is entered with operands IMAGE, REPLACE, the link editor installs the task on a program file. When the FORMAT command is omitted, the linked object code is written on a sequential file in ASCII format. This code must be installed on a program file before execution.

The second command is also optional. It is included when there is a user library that contains one or more modules required for the linking operation.

The remaining commands are required. The task name operand of the PHASE command becomes the task name on the linked object file. The object file operand of the INCLUDE command is the pathname of the object file written by CODEGEN.

Additional commands are required in the file in the following cases:

- Program calls FORTRAN routines.
- Program was compiled with PROBES and/or PROBER option turned on.
- Program calls EXTEND procedure for file OUTPUT or SYSMSG.

When FORTRAN routines are called by the program, the FORTRAN library or libraries must be available for the linking operation. This requires one or more LIBRARY commands in the control file following the LIBRARY .TIP.OBJ command. The following commands access the two standard FORTRAN libraries:

```
LIBRARY .FORTRN.OSLOBJ
LIBRARY .FORTRN.STLOBJ
```

Other FORTRAN routines are on nonstandard libraries. Ideally, only the LIBRARY commands for the library or libraries that include the required routines should be included. When a FORTRAN routine uses FORTRAN I/O, the following command must be included:

```
INCLUDE (FTNIO)
```

If compiler option PROBER or PROBES is turned on during the compilation, the following commands must be included:

```
INCLUDE (PRB$INIT)
INCLUDE (PRB$TERM)
```

An additional command is required when option PROBER is turned on, as follows:

```
INCLUDE (PRB$PERF)
```

An additional command is required when option PROBES is turned on, as follows:

```
INCLUDE (PRB$COMP)
```

When the EXTEND procedure is called to access file OUTPUT, an additional command is required, as follows:

```
INCLUDE (EXTOUT)
```

When an EXTEND procedure is called to access file SYSMSG, an additional command is included, as follows:

```
    INCLUDE (EXTMSG)
```

Two additional options allow a modest reduction in the size of the runtime code by omitting the printing of the abnormal termination dump described in paragraph 14.7.4. One option saves 2340 bytes and provides optional printing of an unformatted dump when an abnormal termination occurs. The other option saves 2840 bytes but provides no dump when an abnormal termination occurs. For either option, substitute an INCLUDE (P$MAIN) command for the INCLUDE (MAIN) command (fifth command) in the control file.

For the optional dump, add an INCLUDE command as follows:

```
    INCLUDE (SPRSDUMP)
```

The optional dump is printed if byte 7 of the task segment contains a nonzero value. No dump is printed if this byte contains zero (the initial value). The contents of the byte may be altered by using the Modify Memory command.

For the option that omits the dump entirely, add an INCLUDE command as follows:

```
    INCLUDE (NODUMP)
```

**14.3.2 EXECUTING UNDER DX10 USING SCI SYNONYMS.** An SCI procedure, XPT, is used to execute a TIP program in which I/O access uses SCI synonyms. The user enters the procedure name XPT at any time DX10 requests a command. DX10 requests the following information:

```
        PROGRAM FILE
    TASK NAME OR ID
               INPUT
              OUTPUT
            MESSAGES
                MODE
              MEMORY
```

The first, third, fourth, and fifth items require access names of devices or files, as follows:

- PROGRAM FILE — The access name of the program file on which the link editor wrote the linked object.
- INPUT — The access name of the file or device for predeclared textfile INPUT.
- OUTPUT — The access name of the file or device for predeclared textfile OUTPUT.
- MESSAGES — The access name of the file or device for system messages.

The second item, TASK NAME OR ID, is the task name supplied to the link editor, or the numeric ID associated with that name on the program file. The sixth item, MODE, is BACKGROUND, FOREGROUND, or DEBUG, specifying the mode of execution of the program. The background and foreground modes are as described for the compiler, paragraph 11.5. When the user enters DEBUG, DX10 places the program in the suspended state. This allows the user to enter any of the debug commands of SCI or any of the Pascal debug commands described in paragraph 14.6.1. The MEMORY item requires an ordered pair specifying stack and heap requirements for the program. The default value is 1, 1 initially. When a value is specified, it becomes the default value until another value is specified.

When a program accesses files other than the predeclared files INPUT and OUTPUT, the file name used in the source program is a synonym of the actual file pathname. The user may assign a pathname as the value of the synonym using an AS command prior to entering the XPT command. When no value has been assigned to the synonym prior to executing XPT, the runtime system assigns a default value. The default value is a pathname consisting of the file name with the station number appended. When the file name consists of more than six characters, the pathname consists of the first six characters of the file name and the station number. The resulting name is cataloged on the volume directory of the system disk.

For example, if the program accessed files FILEA and DISPLAY and the program were executed from terminal ST03, the default pathnames would be:

```
.FILEA03
.DISPLA03
```

The runtime system displays the following message on the system message file when the execution of the program is initiated:

```
<progname> EXECUTION BEGINS
```

When the program terminates normally, the runtime system displays:

```
NORMAL TERMINATION
```

Otherwise, the runtime system displays an error message. Following either termination message, the runtime system displays a summary of the approximate amount of stack and heap space used.

A user-written procedure may be used instead of XPT to execute a TIP task, provided the procedure contains SCI primitive .BID, .QBID, or .DBID to bid the task.

**14.3.3 LINKING FOR DX10 EXECUTION USING LUNOS.** The procedure in this paragraph applies to a TIP task to be executed under DX10 that performs I/O operations using LUNOs for access to files and/or devices. The resulting task can be bid either by an SCI command or by another task (a cooperating task).

The contents of the link edit control file for this type of linking operation are listed, with optional linking commands enclosed in brackets ([ ]). The file should contain the following:

```
[FORMAT <object format>;]       Optional format of linked object code
[LIBRARY <user library>;]       Optional user library
LIBRARY .TIP.LUNOBJ;            Runtime library
LIBRARY .TIP.OBJ;               Runtime library
PHASE 0, <task name>;           Associates task with name
INCLUDE (MAIN);                 Specifies required portion of runtime code
INCLUDE <user object>;          TIP compiler output
END
```

When FORTRAN routines are called by the task, the FORTRAN library or libraries must be available for the linking operation. This requires the following LIBRARY commands in the control file following the LIBRARY .TIP.OBJ command:

```
LIBRARY  .FORTRN.DXLOBJ
LIBRARY  .FORTRN.OSLOBJ
LIBRARY  .FORTRN.STLOBJ
```

The other additional commands listed in paragraph 14.3.1 may be added to the control file also. When included, they provide the same capabilities. The same options for the abnormal termination dump apply.

**14.3.4 EXECUTING UNDER DX10 USING LUNOS.** Tasks linked as described in the previous paragraph are executed using the XT, XTS, or XHT commands. The main difference between using these commands as described in the *Model 990 Computer DX10 Operating System Release 3 Reference Manual, Volume IV* for assembly language tasks and using them for TIP tasks is that TIP tasks use the parameters as values that specify stack and heap requirements.

The LUNOs used in the task must be assigned prior to executing the task. Use the AL command or the AGL command described in the *Model 990 Computer DX10 Operating System Release 3 Reference Manual, Volume II.*

All three commands request the same information, as follows:

```
PROGRAM FILE OR LUNO:
        TASK NAME OR ID:
                 PARM1:
                 PARM2:
            STATION ID:
```

The PROGRAM FILE OR LUNO is either the pathname of the program file on which the task was installed or the LUNO assigned to the program file. If the FORMAT IMAGE, REPLACE command was placed in the link edit file when the task was linked, the pathname entered as the LINKED OUTPUT ACCESS NAME (XLE command) may be used as the first item.

The TASK NAME OR ID is either the task name or the installed ID of the task. The task name was specified in the PHASE command of the link edit file; the installed ID is listed on the link edit map when the link edit operation also installs the task.

The PARM1 and PARM2 are integers that specify numbers of bytes of memory. PARM1 requests memory for stack space, and PARM2 requests memory for heap space. In each case, the item is interpreted as a number of bytes, not thousands of bytes as in the XPT procedure. When the default value of 0 is used, 1024 bytes are requested for stack and 1024 bytes are requested for heap.

The STATION ID is the number of the station (terminal) with which the task is to be associated.

SCI command XT is appropriate for a task that does not interact with the terminal. Command XTS is appropriate for an interactive task. XHT is appropriate for a task that is being debugged; the system places the task in the suspended state after bidding the task. The user may write an SCI procedure to assign the LUNOs and execute the task. Writing SCI procedures is described in the *Model 990 Computer DX10 Operating System Release 3 Reference Manual, Volume V.*

**14.3.5 LINKING FOR TX990 EXECUTION.** The procedure in this paragraph applies to a TIP task to be executed under TX990 using OCP commands.

The contents of the link edit control file for this type of linking operation are listed, with optional linking commands enclosed in brackets ([ ]). The file should contain the following:

| | |
|---|---|
| [FORMAT <object format>;] | Optional format of linked object code |
| [LIBRARY <user library>;] | Optional user library |
| LIBRARY .TIP.LUNOBJ; | Runtime library |
| LIBRARY .TIP.OBJ; | Runtime library |
| PHASE 0, <task name>; | Associates task with name |
| INCLUDE (TXMAIN); | Specifies required portion of runtime code |
| INCLUDE <user object>; | TIP compiler output |
| END | |

When the FORMAT command is used and specifies IMAGE format, the link editor must be called using SCI command TXXLE. The response to the request LINKED OUTPUT ACCESS NAME must be the pathname of a relative record file with 256-byte records. TXXLE executes the link editor to produce a TX990 program file. If XLE is used and a sequential file is written, the response to the request LINKED OUTPUT ACCESS NAME should be the device name of a cassette unit or a sequential file pathname. The SCI command DXTX is used to copy a file to a floppy disk accessible with a TX990 pathname.

When FORTRAN routines are called by the task, the FORTRAN library or libraries must be available for the linking operation. This requires the following LIBRARY commands in the control file following the LIBRARY .TIP.OBJ command:

```
LIBRARY  .FORTRN.TXLOBJ
LIBRARY  .FORTRN.OSLOBJ
LIBRARY  .FORTRN.STLOBJ
```

An additional option is available for TX990 execution. Normally, the TIP task terminates with an End of Task supervisor call. The option substitutes an End of Program supervisor call, which passes control to a rebid task, normally the control program. The option applies when the following command is included following the INCLUDE (TXMAIN) command:

```
INCLUDE (ENDPROG)
```

The other additional commands listed in paragraph 14.3.1 may be added to the control file to provide the same capabilities as with DX10. When either the INCLUDE (SPRSDUMP) or the INCLUDE (NODUMP) command is included, the INCLUDE (TXMAIN) command must be replaced by the following:

```
INCLUDE (P$MAINTX)
```

**14.3.6 EXECUTING UNDER TX990.** The OCP commands used in executing tasks under TX990 are described in the *Model 990 Computer TX990 Operating System Programmer's Guide* (*Release 2*). This paragraph applies to tasks to be executed one at a time while resident in the dynamic task area. The commands that are required are the AL command, the LP command, and the EX command.

All LUNOs required for I/O operations must be assigned by the AL command. An AL command must be executed for each LUNO required by the program. The following example assigns LUNO $3C_{16}$, the SYSMSG file, to the system console:

```
AL,3C,LOG.
```

The task must be loaded into the dynamic task area by executing an LP command. The following command loads a program at pathname DSC2:TIPTSK/LOC into the area at priority level 3, unprivileged:

```
LP,DSC2:TIPTSK/LOC.
```

The amount of stack and heap space required for the task is specified in the parameter operands of the EX command. Both operands are hexadecimal numbers and specify memory size in bytes. The first operand specifies stack size, and the second specifies heap size. The following example executes the task in the dynamic task area (ID $10_{16}$) with 2048 bytes of stack space and 3072 bytes of heap space:

```
EX,10,800,C00.
```

**14.3.7 LINKING MULTIPLE TASKS FOR TX990 EXECUTION.** The procedures in this paragraph apply to TIP tasks to be linked with a TX990 system as memory-resident tasks. When an application requires several cooperating tasks and the tasks are written in Pascal, they must be linked with the system. One or more Pascal tasks may be linked with the system, as required for other reasons.

Only one of the Pascal tasks to be linked with the system may access predeclared files INPUT and OUTPUT. The other tasks must be compiled using a dummy main routine as described in paragraph 15.4, and they begin execution in a procedure. In this way, they can be compiled and linked without the runtime code required for these files.

Generation of a TX990 system, linking of the system, and linking of user tasks with the system are described in the *Model 990 Computer TX990 Operating System Programmer's Guide* (*Release 2*). The following modules must be assembled and linked:

- Configuration module
- TXDATA object module
- TASKDF object module

The configuration module identifies the resident TIP tasks. The source code for the module consists of an IDT directive that assigns a name to the module, a COPY directive that accesses a file of macro definitions, and appropriate macro instructions to generate the required source code. The following source statements provide the source code for the configuration module:

```
IDT '<name>'                  Assigns a name to the configuration module
COPY .TIP.MACROS              Accesses a macro library
TASK <task name>,<procedure>,<stack>,<heap>,<station>
TASK <task name>,<procedure>,<stack>,<heap>,<station>
  .
  .
  .
[MESG <luno>]                 Optional message file specification
[MESG NO]                     Alternative message file specification
[DUMP <task name>,<task number>,<luno>]
                              Optional dump routines
STOP                          Required final macro instruction
END
```

The TASK macro instruction is included for every TIP task. The operands are as follows:

| | |
|---|---|
| <task name> | — the name supplied to GENTX as the task's initial data label |
| <procedure> | — the name of the TIP procedure in which execution is to begin |
| <stack> | — the number of bytes required for the stack for the task |
| <heap> | — the number of bytes required for the heap for the task |
| <station> | — the number of the station to be associated with the task |

The numbers of bytes of stack and heap are the same numbers that are returned in the terminating message when the task is executed. The numbers will be incremented to provide the stack margin.

The MESG macro instruction specifies the file or device on which messages are written. Messages are written when the library routine MESSAGE is called by the user's task and when the task terminates abnormally (unless the DUMP task is included). The operand of the MESG macro instruction specifies the LUNO to which the messages are to be written. The alternate form of the macro instruction, MESG NO, causes the library routine to return control to the task without printing a message.

The DUMP macro instruction causes a task to be included that prints a dump of the process record, stack region, and heap region of the task when it terminates abnormally. The operands are as follows:

| | |
|---|---|
| <task name> | — assigns an initial task data label for the dump task |
| <task number> | — task identifier for the dump task |
| <luno> | — LUNO for the device or file to which the dump is written |

When the macro instruction is omitted, the dump task is not included. The task requires approximately 700 bytes and requires routines that occupy 2700 bytes from the runtime library.

The STOP macro instruction is required. It has no operands and generates miscellaneous declarations.

Source code for the TXDATA and TASKDF modules is written by GENTX. In the task definition entries of GENTX, define each task specified in the configuration module by either a TASK or DUMP macro instruction. The initial data labels named in the macro instructions are supplied to GENTX. The task identifier in the DUMP macro instruction is also supplied to GENTX.

Next, assemble the configuration module, the TXDATA module, and the TASKDF module. When the TIP tasks have been compiled, the system may be linked. The link edit control file (wilth optional commands enclosed in brackets) contains the following:

```
[FORMAT COMPRESSED;]          Optional format of linked object code
NOSYMT;                       Omit symbol tables
LIBRARY .TIP.MINOBJ;          Runtime library
LIBRARY .TIP.LUNOBJ;          Runtime library
LIBRARY .TIP.OBJ;             Runtime library
PHASE 0, TX990;               Identifies TX990 system
INCLUDE <file name>;          File name of TXDATA object
INCLUDE <file name>;          File name of TASKDF object
INCLUDE <file name>;          File name of configuration module object
INCLUDE <file name>;          File name of TIP object module
[INCLUDE <file name>;]        File name of additional TIP object module
   .
   .
   .
ALLOCATE;                     Allocate DSEGs and COMMONs here
SEARCH;                       Search libraries here
INCLUDE <file name>;          File name of TX990 parts
END
```

An INCLUDE must be provided for the object module of each TIP task. The sequence of the ALLOCATE and SEARCH commands is not the logical sequence of these commands; it is the sequence that the current version of the link editor requires. If the search of the libraries introduces a module or modules that have DSEG or COMMON blocks, this sequence will fail. This could occur only if the user enters a LIBRARY command for a user library; the libraries accessed by the LIBRARY commands specified do not specify DSEGs or COMMONs.

**14.3.8 EXECUTING TASKS LINKED WITH TX990.** The same OCP commands referred to in paragraph 14.3.6 are used in executing tasks linked with TX990. When the linked tasks are cooperating tasks that use task control routines to call each other, the procedure in this paragraph applies only to tasks initiated by the user or operator. The EX command specifies the task identifier defined to GENTX for the task.

All LUNOs required for I/O operations (including those in tasks not initiated by the user) must be assigned by the AL command. An AL command must be executed for each LUNO. The following example assigns LUNO $3C_{16}$ to the system console:

```
AL,3C,LOG.
```

No LP commands are required; the tasks are memory resident. The amount of stack and heap space required is specified in the configuration module. The following is an example of the EX command to execute task $3F_{16}$:

```
EX,3F.
```

Since execution of a task that has a dummy main routine begins in a TIP procedure, the declaration of that procedure may declare parameters. When parameters are specified in the EX command, they are passed to the procedure and become the values of the parameters declared for the procedure. The following is an example of a command to execute a task that has declared two parameters for the procedure in which execution begins:

```
EX,3F,200,FF.
```

**14.3.9 LINKING FOR RX990 EXECUTION.** The procedure in this paragraph applies to a TIP task to be executed under RX990 using OCP commands.

The link edit control file (with optional commands enclosed in brackets) contains the following commands:

```
[FORMAT COMPRESSED;]          Optional format of linked object code
[LIBRARY <user library>;]     Optional user library
LIBRARY .TIP.LUNOBJ;          Runtime library
LIBRARY .TIP.OBJ;             Runtime library
PHASE 0, <task name>;         Associates name with task
INCLUDE (TXMAIN);             Specifies required portion of runtime code
INCLUDE <user object>;        TIP compiler output
END
```

The FORMAT command shown is the only optional format available for RX990. The output is a sequential file, whether or not the FORMAT command is entered. The response to the request LINKED OUTPUT ACCESS NAME (XLE command) should be a device that is available on the RX990 system.

The additional commands available for other options are the same as those listed for TX990 execution (paragraph 14.3.5) except for the INCLUDE (ENDPROG) command, which applies only to TX990.

**14.3.10 EXECUTING UNDER RX990.** The OCP commands used in executing tasks under RX990 are described in the *Model 990/10 Computer RX990 Operating System Programmer's Guide.* The first step in executing the task is to install the task. An AL command is required to assign the LUNO for reading the linked object code for the task, and an IT command is required to read and install the code. The following commands are required:

```
AL,2,<device>                 Assign LUNO for reading task
IT,<task id>,<priority>       Install the task
```

Next, all LUNOs required by the task must be assigned with AL commands. The following commands assign the LUNOs for the predeclared files:

```
AL,3C,<device>                Assign LUNO for SYSMSG file
AL,3D,<device>                Assign LUNO for INPUT file
AL,3E,<device>                Assign LUNO for OUTPUT file
```

RX990 LUNOs are global; tasks that execute concurrently must use different LUNOs. TIP tasks that execute concurrently must either change the LUNOs for the predeclared files in all but one task or use the predeclared files in no more than one task. The procedure for changing the LUNO for a predeclared file is shown in paragraph 15.5.

The XT command specifies the stack and heap requirements as hexadecimal numbers of bytes. The command is as follows:

```
XT,<task id>,<stack>,<heap>      Execute the task
```

**14.3.11 LINKING MINIMAL RUNTIME.** To achieve minimal object code, the source code may be prepared in accordance with programming considerations described in paragraph 15.8.

The size of the required stack and heap space is specified in an assembly language module that must be assembled and linked with the object code from the compiler. A default module is provided that specifies 1024 bytes each for stack and heap space. When the default size is not appropriate, prepare an assembly language source module as follows:

```
        IDT    '<name>'                    Module name
STACK   EQU    <stack size>                Number of bytes for stack
HEAP    EQU    <heap size>                 Number of bytes for heap
        DEF    ST$BOT,ST$TOP,HP$BOT,HP$TOP
        DSEG
        EVEN
ST$BOT  BSS    STACK                       Bottom of stack region
        BSS    64                          Stack margin area
ST$TOP  EVEN                               Top of stack region
HP$BOT  BSS    HEAP                        Bottom of heap region
HP$TOP  EVEN                               Top of heap region
        DEND
        END
```

The stack and heap sizes for the EQU statements may be obtained by executing the program under DX10 with full runtime, noting the sizes returned in the termination message. Another method is to first add the stack frame sizes of individual routines and then add two bytes to the total for overhead. The heap size is zero if routine NEW is not called in the task and the Pascal I/O routines are not used. If the default heap manager is used, add the sizes of heap requested. When the heap manager with space recovery is used, add the sizes of memory packets requested in NEW routine calls and not released by calls to the DISPOSE routine at the time when the highest number of packets are active. If only one packet is active at a time, use the size of the largest packet. Add an overhead of six bytes plus two bytes per active packet.

Assemble the module, storing the module on a file to be accessed in the link edit control file during the linking operation. The link edit control file contains the following commands (with optional commands enclosed in brackets):

```
[FORMAT <object format>;]         Optional format of linked object code
[LIBRARY <user library>;]         Optional user library
LIBRARY .TIP.MINOBJ;              Runtime library
LIBRARY .TIP.LUNOBJ:              Runtime library
LIBRARY .TIP.OBJ;                 Runtime library
PHASE 0,<name>;                   Associates name with task
[INCLUDE (MAIN);]                 Specifies portion of runtime required for DX10
[INCLUDE (TXMAIN);]               Specifies portion of runtime required for TX990
                                  or RX990
[INCLUDE <file name>;]            Required for object module specifying stack and
                                  heap size
INCLUDE <user object>;            TIP compiler output
END
```

Another optional command provides a heap manager routine that recovers heap space when the DISPOSE routine is called. The command is INCLUDE (DISPOSE). It is placed in the file following the INCLUDE (MAIN) or the INCLUDE (TXMAIN) command. When the optional command is omitted, the default heap manager is included. The default heap manager includes a limited DISPOSE routine that only disposes of the most recently allocated packet. A call to DISPOSE that specifies any other packet does not release the packet's heap space. When the optional command is included in the link edit control file, a larger heap manager is included. This heap manager recovers heap space by releasing the space allocated to any packet specified in a DISPOSE call.

**14.3.12 EXECUTING MINIMAL RUNTIME.** Executing TIP tasks that have the minimal runtime code linked with them uses the same SCI commands (DX10) or OCP commands (TX990 or RX990) as those used to execute tasks that access I/O devices and files with LUNOs (paragraph 14.3.4, 14.3.6, or 14.3.10). The parameter operands of the XT, XTS, or XHT SCI commands or of the EX or XT OCP commands are not used to specify stack and heap space, which are specified during the linking operation. When the tasks are compiled with a dummy main routine and execution begins in a procedure, parameters placed in the appropriate execute task command are passed to the initial procedure. The declaration of that procedure must specify the desired parameters.

**14.3.13 LINKING FOR STAND-ALONE EXECUTION.** A task that is linked for minimal runtime code and that complies with the limitations described in paragraph 15.7 may be executed stand-alone. The link edit control file for the linking operation is as follows:

```
FORMAT ASCII;                 Format of linked object code required by loader
NOSYMT
LIBRARY .TIP.MINOBJ;          Runtime library
LIBRARY .TIP.LUNOBJ;          Runtime library
LIBRARY .TIP.OBJ;             Runtime library
PHASE 0, <name>;              Associates name with task
INCLUDE (P$MAINSA);           Specifies portion of runtime required for stand-
                              alone
INCLUDE <user program>;       TIP compiler output
END
```

Stand-alone tasks may reside in a combination of ROM and RAM, which requires the use of the PROGRAM, DATA, and COMMON link editor commands described in the *Model 990 Computer Link Editor Reference Manual.*

**14.3.14 STAND-ALONE EXECUTION.** Execution procedures for stand-alone tasks vary considerably. Typically, the linked object module is loaded by a ROM loader and begins execution when the loader passes control to the entry point at completion of the loading operation.

A stand-alone task terminates with an IDLE instruction followed by a JMP $ instruction (a single instruction loop). When the task has executed and the computer is executing either of these instructions, workspace register 0 contains the termination code and workspace register 1 contains the address of the process record. The termination code is one of the codes listed and defined in paragraph E.3; it is zero for normal termination. Paragraph 14.7.4 lists the contents of the process record. The contents of the WP at relative bytes $22_{16}$ and $23_{16}$ of the process record are the address of the stack frame of the routine that executed last. The contents of the PC, bytes $24_{16}$ and $25_{16}$, are the address following the last instruction executed.

**14.3.15 LINKING FOR SHARED PROCEDURES.** The DX10 and RX990 operating systems support the sharing of procedure segments by several tasks. This requires that the code in the shared procedure segment be reentrant and that the linking of each task that shares the procedure allows the same amount of memory for the shared procedure. The object code compiled by the TIP compiler is reentrant, and most of the modules in the runtime libraries are reentrant. This paragraph describes the linking of the shared procedures and of the tasks that share the procedures to provide the required memory-size compatibility.

The first step is to determine which modules may be shared by performing a linking operation for each of the tasks as if there were to be no shared procedure segment. For DX10, the tasks are linked as shown in paragraph 14.3.1 or 14.3.3 except that the INCLUDE (MAIN) command is replaced by an INCLUDE (P$MAIN) command. For RX990, the tasks are linked as shown in paragraph 14.3.9 except that the INCLUDE (TXMAIN) command is replaced by an INCLUDE (P$MAINTX) command. For minimal runtime, the linking is as shown in paragraph 14.3.11, with the appropriate substitution.

Following the linking of each task, examine the link map listings of the tasks. In the module definition portion of the map, the modules that were actually linked are listed. Compare the module lists of each listing to identify the modules that can be placed in a shared procedure segment. The names of these modules (or the first six characters of the module names) are used to select the modules for the shared procedure.

The next step is to link the shared procedure. The link edit control file is as follows:

```
[FORMAT COMPRESSED;]             Optional format of linked object code
[LIBRARY <user library>;]        Optional user library
[LIBRARY .TIP.MINOBJ;]           Optional runtime library
[LIBRARY .TIP.LUNOBJ;]           Optional runtime library (required for RX990)
LIBRARY .TIP.OBJ;                Runtime library
NOAUTO
PROCEDURE <procedure name>;      Associates name with procedure segment
INCLUDE <module name>;           Shared module
  .
  .
  .
SEARCH
TASK <task name>;                Associates name with task segment
DUMMY                            Dummy task segment
[INCLUDE (P$MAIN);]              Must follow DUMMY command (DX10)
[INCLUDE (P$MAINTX);]            Must follow DUMMY command (RX990)
ALLOCATE
END
```

The file must contain an INCLUDE command for each shared module in the procedure segment. This linking operation results in some unresolved references from module P$MAIN because the task segment is a dummy segment; there should be no unresolved references from the procedure segment. The linked object module produced by this operation is the linked object for the shared procedure segment.

The library contains a partial link of most of the routines that are shared by most applications. The linked object of this partial link is selected by the following command:

```
INCLUDE (SHRPROC)
```

This command replaces the INCLUDE commands for the modules included in the partial link. The shared procedure may be linked with the INCLUDE (SHRPROC) command; any unresolved references in the procedure segment indicate that INCLUDE commands for the modules that define those references must be added to the link control file.

The last step is to link the tasks. The LIBRARY commands and INCLUDE commands used to link the procedure must also be used to link the tasks, even though the procedure segment is a dummy segment. These commands result in the correct linking within the task segments. The link edit control file for the task contains the following commands:

```
[FORMAT COMPRESSED;]             Optional format of linked object code
[LIBRARY <user library>;]        Optional user library
[LIBRARY .TIP.MINOBJ;]           Optional runtime library
[LIBRARY .TIP.LUNOBJ;]           Optional runtime library (required for RX990)
LIBRARY .TIP.OBJ;                Runtime library
PROCEDURE <procedure name>;      Associates name with procedure segment
DUMMY                            Dummy procedure segment
INCLUDE <module name>;           Shared module
  .
  .
  .
SEARCH
TASK <task name>;                Associates name with task segment
[INCLUDE (P$MAIN);]              Must follow TASK command (DX10)
[INCLUDE (P$MAINTX);]            Must follow TASK command (RX990)
ALLOCATE
INCLUDE <user object>;           TIP compiler output for main program
INCLUDE <user object>;           Nonshared module
  .
  .
  .
END
```

The file must contain an INCLUDE command for each shared module in the procedure segment. The resulting linked object module is the linked object module for the task.

Install the procedure with the appropriate install procedure command, and install the task with the appropriate install task command, specifying the procedure as the attached procedure for the task. Alternatively, the FORMAT commands shown may be replaced with FORMAT IMAGE commands, which cause the link editor to install the task and procedure after linking. When this is done, the procedure segment must not reference anything in any module specified in an INCLUDE command that follows the ALLOCATE command.

A TIP task that executes under DX10 and uses LUNOs for I/O access or one that executes under RX990 may have two attached procedure segments that contain shared modules. When a task having two attached procedure segments includes FORTRAN routines, the FORTRAN routines must be in either the second procedure segment or in the task segment. A task that executes under DX10, accessing I/O devices and files with SCI synonyms, is limited to one attached procedure segment.

COMMON blocks shared by several tasks may be placed in a procedure segment attached to the tasks. The resulting procedure is no longer reentrant; i.e., the data in the block may change. The procedure may be referred to as a *dirty procedure* because it is no longer pure code. The technique for doing this requires a special $BLOCK module in the linking operation. The special module is provided to support the BLOCK DATA statement of FORTRAN. To place one or more COMMON blocks in a procedure segment, write a TIP procedure named $BLOCK that contains an ACCESS declaration of variables in the shared COMMON block(s). Then, include this module in the procedure segment.

There is an exception to the general rule that the LIBRARY commands used for linking tasks that share a procedure segment must be identical. When it is desired to share a procedure segment with tasks that execute under DX10 and not all of the tasks access I/O in the same way (some use LUNOs, others SCI synonyms), the procedure may be successfully shared if the following holds true: the shared library routines are not fundamentally different in the two libraries, .TIP.LUNOBJ and .TIP.OBJ, and the .TIP.LUNOBJ version is used. When the partial link SHRPROC is used, these requirements are met; i.e., the modules linked in this partial link are essentially the same in both libraries. The routines that cannot be included in the shared procedure segment are listed in table 14-2. Since an open file operation calls some of these routines, the procedure segment cannot contain a user routine that opens a file. Another requirement is that modules ENT$ and ENT$3 through ENT$16 be those in library .TIP.LUNOBJ. This can be done by using explicit INCLUDE commands for these modules in the link edit control file for linking the procedure segment. This is required even though the routines in these modules may be used only by routines in the task segment.

**Table 14-2. Nonsharable Runtime Library Routines**

| | |
|---|---|
| EXTMSG | EXTND$ |
| FIND$S | EXTOUT |
| INIT$ | GO$ |
| MESAG$ | INIT$1 |
| P$TERM | OPEN$ |
| P$UC | REST$R |
| SET$NA | REST$S |
| TIP$TC | REST$T |
| | REWRT$ |

Note: The modules in the left column are not sharable because they are significantly different in libraries .TIP.OBJ and .TIP.LUNOBJ. The modules in the right column are not sharable because they use routines in the left column.

The proper way to link the segments for a procedure attached to tasks using both types of I/O access is to link the procedure segment using a PARTIAL command and libraries .TIP.LUNOBJ and .TIP.OBJ. There should be no unresolved references except references to labels defined in module P$MAIN. If the procedure segment is to be the first of two attached procedures, there should be no unresolved references.

The link edit control files for linking the tasks should contain procedure segments that consist of only an INCLUDE command that specifies the module containing the partially linked object code for the procedure segment. Each task should be linked with or without library .TIP.LUNOBJ, as required.

## 14.4 PASCAL TASK EXECUTION CONSIDERATIONS

There are two considerations that apply to execution of any Pascal task: failures caused by synonym table overflow, and the user condition code.

**14.4.1 SYNONYM TABLE OVERFLOW.** The TIP SCI command procedures use many synonyms, including those that begin with a dollar sign ($) and are transparent to the user. Because of the heavy use of synonyms, the synonym table may overflow. The problems caused by synonym table overflow are not always obvious or straightforward. When entry of a TIP SCI procedure produces unanticipated results, the user should reduce the number of synonyms and try again. Besides deleting any unused synonyms assigned by the user, the user should execute procedure Q$SYN to delete any nonessential synonyms assigned by the operating system and procedure P$SYN to delete any nonessential synonyms assigned by the TIP SCI procedures.

**14.4.2 CONDITION CODE.** Synonym $$CC is used to store system and user condition codes. Procedure P$UC may be called by a user task to set $$CC to one of the following *values:*

| | | |
|---|---|---|
| 0000 | — | Procedure called with a parameter of 0 or not called, and task terminated normally |
| $4000_{16}$ | — | Procedure called with a parameter of 1, and task terminated normally |
| $6000_{16}$ | — | Procedure called with a parameter of 2, and task terminated normally |
| $8000_{16}$ | — | Procedure called with a parameter of 3, and task terminated normally |
| $A000_{16}$ | — | Procedure called with a parameter of 4, and task terminated normally |
| $C000_{16}$ | — | Task terminated abnormally or procedure HALT called |

Synonym $$CC may be tested to detect whether the task terminated normally or whether procedure P$UC was called by something other than a parameter of zero during execution of the task. The synonym must be tested or stored prior to execution of any of the DX10 utilities that set the synonym. Another synonym may be set to the value of $$CC with an .SYN SCI primitive and a value of @$$CC to store the value. Specifically, the LS SCI command always lists the value of $$CC as zero because the command utility sets $$CC to zero.

A task that calls P$UC must declare the procedure externally, as follows:

```
PROCEDURE P$UC (CODE: INTEGER); EXTERNAL;
```

The parameter is a value in the range of 0 through 4. The parameter value determines the value to which $$CC is set, as previously described.

## 14.5 COMPILING, LINKING, AND EXECUTING WITH A BATCH STREAM

The compiler, link editor, and resulting TIP program may be executed using a batch stream. Figure 14-1 shows the SCI commands for a batch stream to compile, link, and execute a program. The example assumes the existence of directory .USER, which contains the source code in a member .USER.SOURCE. The procedure creates files .USER.OBJECT, .USER.LISTING, .USER.MESSAGE, .USER.LINK, .USER.PROGRAM, .USER.LINKLIST, and .USER.OUTPUT. The input data for the program must be on file .USER.INPUT.

## 14.6 DEBUG MODE

The debug mode allows debugging with the SCI debug commands (described in the *Model 990 Computer DX10 Operating System Release 3 Reference Manual, Volume IV*) and with the Pascal debug commands (described in the following paragraphs). The following limitations apply to a task that is to be debugged using the Pascal debug commands:

- The stack display commands SPS and LPS cannot be used while the task is executing in a FORTRAN routine or in any routine with nonstandard linkage. (Appendix F describes the standard interface for assembly language routines.)

- To reference a Pascal routine by name or to have the name displayed, the routine must have been compiled with the TRACEBACK option (a default option) in effect.

```
 BATCH      ! Compile, Link, and Execute a TIP Program
*
*  Execute the TI Pascal Compiler
*
 XTIP  SOURCE =  .USER.SOURCE,  OBJECT =   .USER.OBJECT,
       LISTING = .USER.LISTING, MESSAGES = .USER.MESSAGE
*
 .IF @$$CC, GT, 04000    !  If there were compile errors,
       EBATCH            !     then stop.
 .ENDIF
*
*  Generate a link control file as input to the Link Editor
*
 .DATA .USER.LINK        ; Install directly into user program file,
 FORMAT  IMAGE, REPLACE  ;   replacing existing task by same name--
 LIBRARY .TIP.OBJ        ;   or at first available location if task
 PHASE 0,SAMPLE          ;   does not already exist.
 INCLUDE (MAIN)
 INCLUDE .USER.OBJECT
 END
 .EOD
*
*  Execute the 990 Linkage Editor
*
 XLE   CONTROL ACNM =  .USER.LINK,
       LINKED OUTPUT = .USER.PROGRAM,
       LISTING ACNM =  .USER.LINKLIST
*
 .IF @$$CC, GT, 04000    !  If errors in link,
       EBATCH            !     then stop.
 .ENDIF
*
*  Execute the TIP program just installed
*
 XPT   PGM FILE = .USER.PROGRAM, TASK NAME = SAMPLE,
       INPUT =    .USER.INPUT,   OUTPUT =  .USER.OUTPUT,
       MESSAGE =  .USER.MESSAGE
*
 EBATCH     !  End of batch stream.
```

**Figure 14-1. Batch Stream for Compiling, Linking, and Executing a TIP Program**

The Pascal debug commands may be used to debug tasks linked in any of the ways described in paragraph 14.3 except tasks linked with TX990 (described in paragraph 14.3.7). Tasks that are linked to be executed under either TX990 or RX990 may be debugged under DX10 to make use of the capabilities of the Pascal debug commands.

The Pascal debug commands provide breakpoint capability and the capability of displaying the Pascal stack. Symbolic debugging is not supported in Pascal tasks.

Both the SCI and Pascal debug commands require that the task be unconditionally suspended either before or during the execution of the command. The unconditionally suspended state (task state 6) is the state in which the task is suspended until activated by a command. There are several ways in which a task may be unconditionally suspended:

- The Pascal task is executed with the debug mode selected, which causes the task to be loaded for execution but suspended before execution actually begins.
- The task suspends itself.
- The task executes a breakpoint (XOP 15, 15).
- The task is suspended by the SCI debug commands.

Once the task is in a suspended state, debug commands may be used to assign breakpoints, simulate execution, display memory, and perform other debug functions. When the debugging has been completed, the task may be terminated by entering a Kill Background Task (KBT) command.

The task may be placed in the controlled mode by executing the Debug command (XD) as for any task being debugged under DX10. Placing a Pascal task in the controlled mode enables displays associated with breakpoints and allows the user to enter the controlled mode debug commands. Otherwise, the user must monitor the status of the task using the Show Pascal Stack (SPS) command or the Show Internal Register (SIR) command.

When the terminal that is executing the Pascal task is a VDT, a breakpoint causes a display of data. When the terminal is a TTY type device (or the VDT is in the TTY mode), no display occurs.

**14.6.1 PASCAL DEBUG COMMANDS.** Seven commands are provided with the SCI to support the debugging of Pascal tasks. These commands provide a means of examining the stack area of a Pascal task and of setting breakpoints in a manner that is independent of the code generated by the Pascal compiler. The commands are:

- Assign Breakpoint — Pascal (ABP)
- Delete Breakpoint — Pascal (DBP)
- Delete and Proceed from Breakpoint — Pascal (DPBP)
- Proceed from Breakpoint — Pascal (PBP)
- List Breakpoints — Pascal (LBP)
- Show Pascal Stack (SPS)
- List Pascal Stack (LPS)

**NOTE**

The Pascal debug commands are available only with DX10 operating systems Release 3.2 and subsequent releases.

The commands are entered when the SCI command prompt ([]) is displayed. The syntax shown in the descriptions of the commands represents the actual display. The following conventions are used:

previous — Indicates the last value entered for the prompt.

acnm -- Indicates an access name (either a device name, or a file pathname).

string — Indicates a character string.

constant exp — Indicates a decimal or hexadecimal integer or an expression composed of decimal or hexadecimal integers and the operators +, -, *, and /.

Underscore (_) — Indicates the default value.

Braces ({ }) — Enclose alternative entries, one of which must be entered.

Angle brackets (<>) — Enclose a user entry.

Brackets ([]) — Enclose an optional entry.

Uppercase — Indicates system displayed data.

SCI writes error messages when it detects errors in the Pascal debug commands. These messages are listed in Appendix E.

**14.6.1.1 Assign Breakpoint — Pascal (ABP).** The Assign Breakpoint — Pascal (ABP) command assigns a breakpoint at entry to, return from, or both entry to and return from a Pascal routine. The syntax of the ABP command is:

```
ASSIGN BREAKPOINT — PASCAL

                       RUN ID:  { <constant exp> }
                                {   previous     }

              NAME OF ROUTINE:  <string>

WHERE(ENTRY/RETURN/BOTH):       <string>
```

The RUN ID is the DX10-assigned runtime ID of the Pascal task to be debugged. The NAME OF ROUTINE is the procedure or function identifier of any Pascal or assembly language routine called by the Pascal task. Pascal routines must be compiled with the Traceback option. Assembly language routines must have the standard interface described in Appendix F. Valid responses to the WHERE prompt are ENTRY, RETURN, BOTH, or single-character abbreviations of any of these. ENTRY (or E) specifies a breakpoint at entry to the named routine; RETURN (or R) specifies a breakpoint at return from the routine; and BOTH (or B) specifies breakpoints at entry and return.

The first instruction in a routine compiled by the TIP compiler (or coded with the standard interface) is a BL instruction to a runtime routine for entry handling. When an ABP command that specifies either ENTRY or BOTH is executed, the command replaces the instruction following the BL instruction with a breakpoint (XOP 15,15). The last instruction in the routine is a B instruction to a runtime return handler. When an ABP command that specifies either RETURN or BOTH is executed, the command replaces the B instruction with a breakpoint (XOP 15,15). This is the implementation of a breakpoint in a Pascal task, and these are the only points at which a breakpoint may be placed.

Execution of a breakpoint suspends the task, placing it in state 6. When execution has been initiated by a Delete and Proceed from Breakpoint — Pascal command or a Proceed from Breakpoint — Pascal command, the terminal is a VDT, and debugging is in the controlled mode, a display appears on the VDT screen when the breakpoint is executed.

Execution can be continued following a breakpoint by entering one of the following:

- A Proceed from Breakpoint — Pascal command
- A Delete and Proceed from Breakpoint — Pascal command
- A Delete Breakpoint — Pascal command followed by a Resume Task command

A task need not be in memory when an ABP command sets a breakpoint in the task. When the task is not suspended, an ABP command suspends the task while it sets the breakpoint and then restores the original state of the command.

Example:

```
ASSIGN BREAKPOINT — PASCAL

                         RUN ID:  >04

              NAME OF ROUTINE:  CCHAR

WHERE(ENTRY/RETURN/BOTH):  RETURN
```

The example command sets a breakpoint at the return from routine CCHAR of a task with the runtime ID of $04_{16}$.

**14.6.1.2 Delete Breakpoint — Pascal (DBP).** The Delete Breakpoint — Pascal (DBP) command deletes a breakpoint at entry to, return from, or both entry to and return from a Pascal routine. The syntax of the DBP command is:

```
DELETE BREAKPOINT — PASCAL

                         RUN ID:  { <constant exp> }
                                  {   previous     }

              NAME OF ROUTINE:  { <string> }
                                {   ALL    }

WHERE(ENTRY/RETURN/BOTH):  <string>
```

The RUN ID is the DX10-assigned runtime ID of the Pascal task to be debugged. The NAME OF ROUTINE is either the procedure or function identifier of a routine or keyword ALL. When keyword ALL is entered, all breakpoints in the task are deleted. Valid responses to the WHERE prompt are ENTRY, RETURN, BOTH, or single-character abbreviations of any of these. The response specifies deleting a breakpoint at entry or return from the routine, or breakpoints at both entry and return from the routine.

When a breakpoint has been set at the specified location, the DBP command replaces the breakpoint (XOP 15,15) with the original contents of the location. When the location does not contain a breakpoint, the command writes a warning message. If the task is suspended when the command is entered, the command does not cause the task to resume execution. A Resume Task command should be entered to resume execution.

Example:

```
DELETE BREAKPOINT — PASCAL

                     RUN ID:  >04

         NAME OF ROUTINE:  CCHAR

WHERE(ENTRY/RETURN/BOTH):  E
```

The example command deletes a breakpoint at entry to routine CCHAR of a task with the runtime ID of $04_{16}$.

**14.6.1.3 Delete And Proceed From Breakpoint — Pascal (DPBP).** The Delete and Proceed from Breakpoint — Pascal (DPBP) command deletes the breakpoint that suspended the task, optionally assigns another breakpoint, and resumes execution of the task. The syntax of the DPBP command is:

```
DELETE AND PROCEED FROM BREAKPOINT — PASCAL

                     RUN ID: {<constant exp>}
                             {  previous    }

     DESTINATION ROUTINE:  [<string>]

WHERE(ENTRY/RETURN/BOTH):  [<string>]
```

The RUN ID is the DX10-assigned runtime ID of the Pascal task being debugged. The other two parameters may be omitted; when omitted, no breakpoint is assigned. The DESTINATION ROUTINE is the procedure or function identifier of a routine. Valid responses to the WHERE prompt are ENTRY, RETURN, BOTH, or single-character abbreviations of any of these. The response specifies assigning a breakpoint at the entry or return of the specified routine, or breakpoints at the entry and return of the routine.

When the task has not been suspended by a breakpoint, the command assigns the specified breakpoint, if any, and writes a warning message. When the task is in the controlled mode, all activity at the terminal is suspended until the task executes a breakpoint. Data similar to the display of an SPS command (paragraph 14.6.1.6) is displayed at a breakpoint (when the terminal is a VDT). When the task is not in the controlled mode, the user must monitor task status with an SP or SIR command (described in the *Model 990 Computer DX10 Operating System Release 3 Reference Manual, Volume IV*) to determine that a breakpoint has executed.

Examples:

```
DELETE AND PROCEED FROM BREAKPOINT — PASCAL

                              RUN ID:  >04

                 DESTINATION ROUTINE:

            WHERE(ENTRY/RETURN/BOTH):
```

The example command deletes the breakpoint that suspended task $04_{16}$ and resumes execution of the task (if task $04_{16}$ was suspended by a breakpoint). If the task was not suspended by a breakpoint, the command displays a warning message.

```
DELETE AND PROCEED FROM BREAKPOINT — PASCAL

                              RUN ID:  >04

                 DESTINATION ROUTINE:  CINT

            WHERE(ENTRY/RETURN/BOTH):  BOTH
```

The example command deletes the breakpoint, assigns breakpoints at the entry and return of routine CINT of task $04_{16}$, and resumes execution of the task. If task $04_{16}$ had not been suspended by a breakpoint, the command would have assigned the breakpoint and then displayed a warning message.

**14.6.1.4 Proceed From Breakpoint — Pascal (PBP).** The Proceed from Breakpoint — Pascal (PBP) command optionally assigns a breakpoint and resumes execution of the task. The syntax of the PBP command is:

```
PROCEED FROM BREAKPOINT — PASCAL

                              RUN ID: { <constant exp> }
                                      {   previous     }

                 DESTINATION ROUTINE:  [<string>]

            WHERE(ENTRY/RETURN/BOTH):  [<string>]
```

The RUN ID is the DX10-assigned runtime ID of the Pascal task being debugged. The other two parameters may be omitted; when omitted, no breakpoint is assigned. The DESTINATION ROUTINE is the procedure or function identifier of a routine. Valid responses to the WHERE prompt are ENTRY, RETURN, BOTH, or single-character abbreviations of any of these. The response specifies assigning a breakpoint at the entry or return of the specified routine, or breakpoints at the entry and return of the routine.

When the task is in the controlled mode, all activity at the terminal is suspended until the task executes a breakpoint. Data similar to the display of an SPS command (paragraph 14.6.1.6) is displayed at a breakpoint (when the terminal is a VDT). When the task is not in the controlled mode, the user must monitor task status with an SP or SIR command to determine that a breakpoint has executed.

Examples:

```
PROCEED FROM BREAKPOINT — PASCAL

                   RUN ID:  >04

      DESTINATION ROUTINE:

WHERE(ENTRY/RETURN/BOTH):
```

The example command resumes execution of task $04_{16}$ without altering the number of active breakpoints.

```
PROCEED FROM BREAKPOINT — PASCAL

                   RUN ID:  >04

      DESTINATION ROUTINE:  CINT

WHERE(ENTRY/RETURN/BOTH):  E
```

The example command assigns a breakpoint at the entry to routine CINT of task $04_{16}$ and resumes execution of the task.

**14.6.1.5 List Breakpoints — Pascal (LBP).** The List Breakpoints — Pascal (LBP) command writes a list of the breakpoints assigned for a task. The syntax of the LBP command is:

```
LIST BREAKPOINTS — PASCAL

                 RUN ID: { <constant exp> }
                         {   previous     }
```

The RUN ID is the DX10-assigned runtime ID of the Pascal task for which breakpoints are to be listed.

Each entry of the listing describes the breakpoint or breakpoints for a routine of the task. The format of each entry of the listing is as follows:

```
NNNNNNNN L EEEE RRRR
```

The routine name is represented by NNNNNNNN. The letter represented by L is E, R, or B, for entry breakpoint, return breakpoint, or both entry and return breakpoints. Field EEEE contains the hexadecimal address within the task of the entry breakpoint when L is either E or B. Field RRRR contains the hexadecimal address within the task of the return breakpoint when L is either R or B. Up to three entries may be displayed on a line of the listing.

Example:

```
LIST BREAKPOINTS — PASCAL

                 RUN ID:  >04
```

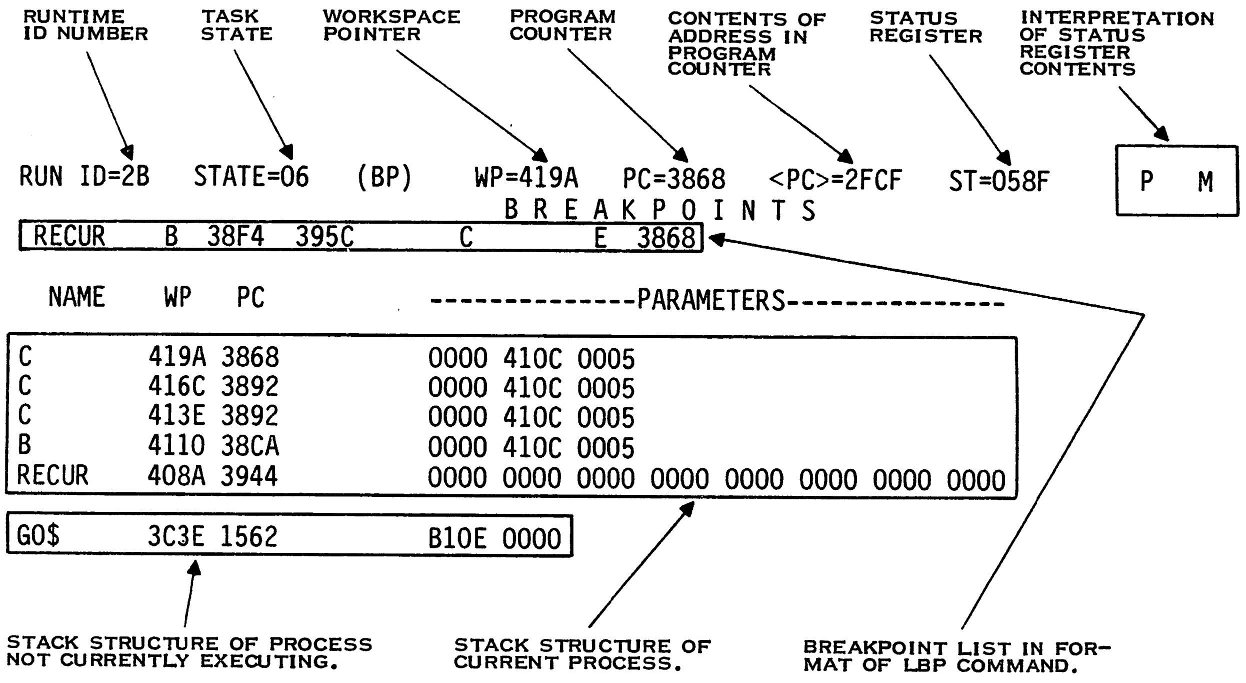
The example command specifies listing all breakpoints assigned for task $04_{16}$.

**14.6.1.6 Show Pascal Stack (SPS).** The Show Pascal Stack (SPS) command displays data from the stack structure of the currently executing process and other data pertaining to the specified task. The syntax of the SPS command is:

```
SHOW PASCAL STACK

                        RUN ID: { <constant exp> }
                                {   previous     }
```

The RUN ID is the DX10-assigned runtime ID of the Pascal task for which the stack is to be displayed.

The displayed data includes the contents of the internal registers, the breakpoints, and the stack structure of all processes beginning with the current process. An example of the display is as follows:


RUNTIME ID NUMBER
TASK STATE
WORKSPACE POINTER
PROGRAM COUNTER
CONTENTS OF ADDRESS IN PROGRAM COUNTER
STATUS REGISTER
INTERPRETATION OF STATUS REGISTER CONTENTS
STACK STRUCTURE OF PROCESS NOT CURRENTLY EXECUTING.
STACK STRUCTURE OF CURRENT PROCESS.
BREAKPOINT LIST IN FORMAT OF LBP COMMAND.

```
RUN ID=2B   STATE=06   (BP)   WP=419A   PC=3868   <PC>=2FCF   ST=058F     P   M
                                  B R E A K P O I N T S
RECUR      B  38F4  395C         C          E  3868

 NAME      WP    PC            --------------PARAMETERS---------------

C         419A 3868            0000 410C 0005
C         416C 3892            0000 410C 0005
C         413E 3892            0000 410C 0005
B         4110 38CA            0000 410C 0005
RECUR     408A 3944            0000 0000 0000 0000 0000 0000 0000 0000

GO$       3C3E 1562            B10E 0000
```

The first line of the display shows the runtime ID, the task state, the contents of the Workspace Pointer (WP) and the Program Counter (PC), the contents of the address in the PC, the contents of the Status Register, and the interpretation of the Status Register contents. In the example, (BP) follows the task state to indicate that the task is in unconditional suspension due to execution of a breakpoint. The address in the PC is the address of the instruction that is to be executed next when execution resumes. The contents of the address is the machine instruction that is to be executed next. The letters P and M indicate that the Parity and Map file bits of the Status Register have been set to one. The letters that may be displayed in this field are listed in the description of the SIR command in the *Model 990 Computer DX10 Operating System Release 3 Reference Manual, Volume 4.*

The next portion of the display lists the assigned breakpoints (in the format defined for the LBP command). Three breakpoints are listed in the example: two in program module RECUR and one in routine C. The contents of the PC and the address of the breakpoint at the entry to routine C are the same; the breakpoint in routine C is the breakpoint that resulted in this display.

The remainder of the display consists of the stack structures of the processes involved in executing the task. For more information on processes, refer to the description of the abnormal termination dump (paragraphs 14.7.5, 14.7.6, and 14.7.7). The stack structure of the currently executing process is displayed first. Each line of the stack structure portion of the display corresponds to the stack frame of an active routine, listed in the reverse of the order in which they were activated. In the example above, program module RECUR called routine B, which called recursive routine C. The display shows the stack structure at entry of the second time routine C called itself. The first line corresponds to the second recursive call of routine C; the second line to the first recursive call; the third line to the call of routine C by routine B; the fourth line to the call of routine B; and the last line to the program module RECUR.

Each line of the stack structure lists the contents of the WP and PC and parameters for the routine. The address in the WP is also address of the bottom (lowest address) of the stack frame for the routine. The address in the PC is the address at which execution of the routine resumes when control returns to the routine. The parameters listing shows the contents of up to eight words of the stack frame, beginning at address $28_{16}$ relative to the bottom of the stack frame. When the stack frame contains fewer than $30_{16}$ words, only the words from relative address $28_{16}$ to the top of the frame are displayed. When a routine has eight or more parameters, eight (or the first eight) words of the parameters are displayed. When the routine has fewer than eight parameters, all of the parameters are displayed, followed by local data if the routine has any local data. The word that corresponds to a value parameter contains the value of the parameter; the word that corresponds to a reference parameter contains the address of the parameter. In the example, routine C has three parameters and no local data. Parameter 1 is a value parameter with a value of zero; parameter 2 is a reference parameter at location $410C_{16}$; and parameter 3 is a value parameter with a value of 5.

A blank line separates the stack structure for the currently executing process from that of the first process that is not currently executing. The format of the display of a process that is not currently executing is the same as the display format described previously. The information is not as useful for debugging purposes because the user is not concerned with debugging the routines in these processes. In the example, the stack structure of the non-executing process contains a single stack frame. When there is more than one non-executing process, more than one additional stack structure is displayed; a blank line separates each stack structure display from that of the next stack structure.

Example:

```
    SHOW PASCAL STACK

                            RUN ID:  >2B
```

The example command specifies displaying the stack structure for task $2B_{16}$.

**14.6.1.7 List Pascal Stack (LPS).** The List Pascal Stack (LPS) command lists a specified portion of the stack frame for a specified routine. The listing is written to a specified device. The syntax of the LPS command is:

LIST PASCAL STACK

$$\text{RUN ID:}\begin{Bmatrix}\text{<constant exp>}\\ \underline{\text{previous}}\end{Bmatrix}$$

NAME OF ROUTINE: [<string>]

$$\text{STARTING OFFSET:}\begin{Bmatrix}\text{<constant exp>}\\ \underline{0}\end{Bmatrix}$$

NUMBER OF BYTES: [<constant exp>]

LISTING ACCESS NAME: [<acnm>]

The RUN ID is the DX10-assigned runtime ID of the Pascal task for which the stack is to be listed. The NAME OF ROUTINE is optional. When a routine name is entered, stack frames for the named routine in all processes are listed. When the name is omitted, all stack frames of all processes are listed. The STARTING OFFSET is the relative address within each stack frame of the first word to be listed. The NUMBER OF BYTES is optional. When a number of bytes is entered, the specified number of bytes of each stack frame are listed. When the number is omitted or when the number entered is greater than the number the stack frame contains, the command lists the contents from the starting offset to the top of the frame. The LISTING ACCESS NAME is a DX10 pathname of a file or device to which the listing is to be written. The parameter is optional; when it is omitted, the list is written to the Terminal Local File of the terminal and is then displayed.

If the task specified in the command is not unconditionally suspended, the task is suspended while the listing is written. The listing consists of a heading and a variable number of lines for each stack frame. The format of the heading is as follows:

```
LIST STACK FRAME OF NNNNNNNN. FRAME BEGINS AT AAAA.
```

NNNNNNNN represents the name of the routine. AAAA represents the address of the bottom of the stack frame.

The format of the lines that follow the heading is:

```
RRRR DDDD DDDD DDDD DDDD DDDD DDDD DDDD DDDD CCCCCCCCCCCCCCCC
```

RRRR represents the relative address within the stack frame of the first word listed. Each DDDD represents the contents of a word of the stack frame listed as four hexadecimal digits. Each C represents the contents of a byte represented as an ASCII character, or as a period (.) if the ASCII character is unprintable.

The number of lines listed for each stack frame depends upon the number of bytes to be listed. One line is listed when 16 or fewer bytes are to be listed. When more than 16 bytes are to be listed, they are listed 16 bytes per line with any remaining bytes on an additional line.

The stack frames for the currently executing process are listed first, followed by those for processes that are not executing. When the specified routine is executing or suspended or when all routines are being listed, the first stack frame listed is that of the current routine. Otherwise, the stack frame that corresponds to the most recent call of the routine is listed first. Additional stack frames are listed in the reverse of the order in which they were called. A blank line separates the listings of stack frames; two blank lines separate the listings of stack frames of a process from those of another process.

When listing of stack frames for a task or routine that is not active is requested, the command writes an error message.

Examples:

```
LIST PASCAL STACK

                RUN ID: >2B
```

```
          NAME OF ROUTINE:
          STARTING OFFSET:  >28
          NUMBER OF BYTES:
      LISTING ACCESS NAME:  MYVOL.PASCAL.STACKS.TASK1
```

The example command lists the stack frames of all routines of task $2B_{16}$ on a file. The entire contents of the stack frames starting with byte $28_{16}$ relative to the bottom of the stack are listed. The access name of the file is MYVOL.PASCAL.STACKS.TASK1.

```
   LIST PASCAL STACK

                   RUN ID:  >2B
          NAME OF ROUTINE:  B
          STARTING OFFSET:  >28
          NUMBER OF BYTES:  10
      LISTING ACCESS NAME:
```

The example command lists the stack frames for routine B of task $2B_{16}$ on the terminal. Ten bytes starting at byte $28_{16}$ of each stack frame are requested. As many as 16 bytes may be listed per stack frame; the requested number of bytes requires one line. If there are 16 or more bytes from the starting byte to the top of the stack, 16 bytes are listed, filling the line. Otherwise, the bytes from the starting byte to the top of the stack are listed.

**14.6.2 DEBUGGING UNDER TX990.** The Pascal debug commands are not available under TX990. Although a task linked for execution under TX990 may be debugged under DX10, it is also possible to debug the task under TX990 using the debug commands of OCP. These commands are described in the *Model 990 Computer TX990 Operating System Programmer's Guide* (*Release 2*).

The Task Status (ST) command may be used to determine the contents of the WP, which is also the pointer to the stack frame. The Dump Memory (DM) command displays the contents of the stack frame, which are described in paragraph 14.7.4. The contents of the word at address $26_{16}$ relative to the stack frame pointer (WP) is the address of the process record. The DM command can be used to display the contents of the process record. The DISPLAY array and the other contents of the process record, described in paragraph 14.7.4, are helpful in determining the progress of the task's execution.

The contents of the PC can be used to determine the routine that is in execution. An example of a link map is shown in figure 14-3. Using the link map for the task in execution, identify the module that is in execution by comparing the PC value minus the task address shown by the ST command to the origin addresses of the modules listed in the link map. The Reverse Assembler listing, an example of which is shown in figure 13-1, lists the contents of the module.

The data for the routine in execution is in the stack frame. When the task is compiled with the MAP option, the addresses of the data relative to the stack frame pointer are listed. An example of a compiler listing with MAP option is shown in figure 9-1.

## 14.7 RUNTIME ERRORS

There are several categories of errors that are detected at runtime and cause abnormal termination of a TIP program. The error categories are:

- Failures of runtime checks specified as compiler options
- I/O errors
- Depletion of available memory space
- Task errors
- Mathematics package errors

In each case, an error message is displayed and an abnormal termination occurs. An abnormal termination dump is displayed along with the error message. The following paragraphs describe the error categories and the abnormal termination dump. Runtime error messages are listed in Appendix E.

**14.7.1 RUNTIME CHECKS.** Seven compiler options provide runtime checks by including additional code in the object module to perform the checks and issue the error messages. The options and the values checked are:

| | |
|---|---|
| CKINDEX | Array indexes within range. |
| CKOVER | Arithmetic overflow. |
| CKPREC | Loss of precision. |
| CKPTR | NIL pointer values. |
| CKSET | Set elements within range. |
| CKSUB | Subrange assignments. |
| CKTAG | Identifiers consistent with tag fields. |

When any of these options has been specified for compilation of a program and the check fails at runtime, abnormal termination occurs and the error message and abnormal termination dump are displayed.

**14.7.2 I/O ERRORS.** When an I/O error occurs and the I/O termination flag of the file or device is true, an abnormal termination occurs. The user program may call procedure IOTERM and set the flag for a file or device false; otherwise, the flag is true. The error message contains an I/O error code that is either an operating system I/O error code or a TIP I/O error code, depending on the type of operation and file. An EXTEND, RESET, or REWRITE operation on a random or sequential file returns a code provided by the open operation of the operating system. Operations on textfiles return one of the following error codes:

| | |
|---|---|
| 1 | Bad parameter passed to I/O routine. |
| 2 | Field width too large for logical record. |

3 Incomplete data (READ operation).

4 Invalid character in field (READ only).

5 Data value too large (READ only).

6 Attempt to read past end-of-file.

7 Field larger than logical record size.

The error message with an error code and the abnormal termination dump are displayed when an I/O error abnormal termination occurs.

**14.7.3 MEMORY SPACE ERRORS.** The variables for a TIP program are organized in a stack (last in, first out). The variables for the main program are organized in a stack frame, and those of each routine are similarly organized in a stack frame. When executing a program, the user specifies an amount of memory space for the stack. When execution begins, the stack frame of the main program is placed in the stack. When a routine is called, its stack frame is added to the stack. If the routine calls a nested routine, the stack frame of that routine is also added, so that at any time the stack contains the stack frame of the currently executing routine and those of all its ancestors, including the main program. Should a call for a routine occur when the stack does not contain space for the stack frame of the routine, an error termination occurs, displaying the message:

```
STACK OVERFLOW
```

Memory space for dynamic variables is assigned in an area of memory called the heap when procedure NEW is called in a TIP program. The user specifies an amount of memory space for the heap when executing a program. If procedure NEW requests more memory for a variable (or record of variables) than remains in the heap and the runtime system is unable to obtain more memory space for the heap, an error termination occurs, displaying the following message:

```
HEAP FULL
```

**14.7.4 ERROR TERMINATION.** Figure 14-2 shows the error message written on the system message file and the abnormal termination dump written on file OUTPUT for an I/O error. The error message indicates that the error occurred when an attempt was made to open file INPUT, access name .INPUT04. The status shown, 27, is the completion code returned by DX10 Release 3.2, which means NO FILE DEFINED BY NAME SPECIFIED. No further analysis of this error is necessary; the user assigns the access name of an existing file when executing the program.

The abnormal termination dump includes a dump of each process involved in the run and a dump of the heap. The TIP runtime support system has been designed to permit multiple sites of execution within the executable object code for a program. Each such site is called a process and has its own stack region within which its local variables are allocated. Each TIP program that executes under DX10 has at least two processes. The example program has two: DIGIO (the user program) and system process GO$. GO$ is an initialization/termination process that is given control when a TIP load module is placed in execution. GO$ determines the user's stack and heap parameters, allocates and initializes both the stack region for the user program and the heap, opens the file OUTPUT, and transfers control to the user program. Upon termination, GO$ again receives control and performs a controlled termination, either normal or abnormal.

```
DIGIO     EXECUTION BEGINS
 FILE INPUT    ACCESS NAME .INPUT04
 OPEN ERROR - STATUS = 27
HALT CALLED
STACK USED =    660  HEAP USED =   324

       *** DUMP OF PROCESS ***

 PROCESS RECORD FOR  DIGIO
 58AC (0000) 470E 542A 5592 FF00 FF00 FF00 FF00 FF00  (G.T*U...........)
 58BC (0010) FF00 FF00 FF00 FF00 FF00 FF00 FF00 FF00  (................)
 58CC (0020) FF00 5592 1592 C18F 470E 8000 0000 5428  (..U.....G.....T()
 58DC (0030) 56BC 58AA 0000 0000 4FD4 4FBC 0300       (V.X.....O.O...  )

 TOP OF STACK
 55BA (0000) 0300 0019 0000 0000 0000 0000 0000 4B64  (..............K.)
 55CA (0010) 4FBC 55BA 55E4 00B2 58AC 4B64 275E C18F  (O.U.U...X.K.'^..)
 55DA (0020) 0000 0000 0000 0000 0000 0000 0000 0000  (................)
 55EA-5609  SAME AS LAST LINE
 560A (0050) 0549 4E50 5554 0000 0000 0019 0001 0000  (.INPUT..........)
 561A (0060) 0000 0000 0000 0000 0000 0000 5614 5640  (............V.V@)
 562A (0070) 25F8 58AC 5592 3BD4 318F 0000 0000 FFFF  (%.X.U.;.1.......)

 DATA AREA FOR  HALT$       LEVEL=2
 5592 (0000) 0008 0001 565C 1BB4 0100 5548 284A 4FE4  (....V\....UH(JO.)
 55A2 (0010) 0001 5592 55BA 00CC 58AC 5516 1A44 D18F  (..U.U...X.U..D..)
 55B2 (0020) 477E 5516 1A44 0000                      (G.U..D..        )

 DATA AREA FOR  IO$ERR      LEVEL=2
 5516 (0000) 1A3E 0C00 3030 3732 0002 5542 5542 5375  (.>..0072..UBUBS.)
 5526 (0010) 548A 5516 5592 1582 5378 54E4 200C C18F  (T.U.U...S.T. ...)
 5536 (0020) D18F 0000 FFFF 0000 548A 0000 204F 5045  (........T... OPE)
 5546 (0030) 4E20 4552 524F 5220 2D20 5354 4154 5553  (N ERROR - STATUS)
 5556 (0040) 203D 2032 3745 202E 494E 5055 5430 347A  ( = 27E .INPUT04.)
 5566 (0050) 2A74 58AC 5516 219E C58F 5378 494E 0000  (*.X.U.!...S.IN..)
 5576 (0060) 5420 5378 0000 0000 536C 0000 0007 3400  (T S.....S.....4.)
 5586 (0070) 5374 5373 0000 0000 0000 0000            (S.S.........    )

 DATA AREA FOR  OPEN$       LEVEL=2
 54E4 (0000) 0000 0001 0000 54E4 0014 0014 54EA 54A0  (......T.....T.T.)
 54F4 (0010) 5546 54E4 5516 1886 58AC 54BA 443E 548A  (UFT.U...X.T.D>T.)
 5504 (0020) 318F 58AC FFFF FFFF 548A 0000 53A0 0027  (1.X.....T...S..')
 5514 (0030) 0050                                     (.P              )

 DATA AREA FOR  REST$T      LEVEL=2
 54BA (0000) 0013 0001 0000 0000 0000 0000 0000 54EA  (..............T.)
 54CA (0010) 54A0 54BA 54E4 1F44 58AC 542A 3A60 548A  (T.T.T..DX.T*:.T.)
 54DA (0020) 0000 0000 FFFF 0000 548A                 (........T.      )

 DATA AREA FOR  DIGIO       LEVEL=1
 542A (0000) 582A 0000 0000 0000 0000 0000 0000 39D4  (X*............9.)
 543A (0010) 54EA 542A 54BA 4424 58AC 5412 03C8 018F  (T.T*T.D$X.T.....)
 544A (0020) 0000 5412 03C8 0000 0000 0000 0000 0000  (..T.............)
 545A (0030) 0000 0000 0000 0000 0000 0000 0000 0000  (................)
 546A (0040) 546A 0011 004F 53A0 0141 0001 0050 5400  (T..4.OS..A...PT.)
 547A (0050) 4F55 5450 5554 2020 0001 0100 0000 0000  (OUTPUT  ........)
 548A (0060) 548A 0000 0000 0000 0041 0002 0000 5378  (T........A....S.)
 549A (0070) 494E 5055 5420 2020 0001 0100 0027 0000  (INPUT   .....'..)
 54AA (0080) 0000 0000 0000 0000 0000 0000 0000 0000  (................)
```

**Figure 14-2. Abnormal Termination Dump Example (Sheet 1 of 2)**

```
          *** DUMP OF PROCESS ***

PROCESS RECORD FOR   GO$
470E (0000) 58AC 542A 4814 FF00 FF00 FF00 FF00 FF00  (X.T*H...........)
471E (0010) FF00 FF00 FF00 FF00 FF00 FF00 FF00 FF00  (................)
472E (0020) FF00 477E 3004 FFFF 58AC 8000 0000 4750  (..G.0...X.....GP)
473E (0030) 4A2E 4B5A 0000 0000 4FD4 4FBC FFFF       (J.KZ....O.O...  )

DATA AREA FOR  TERM$          LEVEL=2
477E (0000) 39F8 0001 0000 477A 0000 0000 0000 0000  (9.....G.........)
478E (0010) 2E56 477E 47DA 062E 470E 58AC 155E 542A  (.VG.G...G.X..^T*)
479E (0020) FF00 4752 155E 0000 58AC 39C2 4449 4749  (..GR.^..X.9.DIGI)
47AE (0030) 4F20 2020 2020 4558 4543 5554 494F 4E20  (O     EXECUTION )
47BE (0040) 4245 4749 4E53 2E93 0404 0000 0000 0000  (BEGINS..........)
47CE (0050) 0001 477A 470E 0003 0000 542A            (..G.G.....T*    )

DATA AREA FOR  GO$            LEVEL=2
4752 (0000) 4ADA 0000 0000 0000 0000 0000 0000 470E  (J.............G.)
4762 (0010) 0000 4752 477E 2C80 470E 473A 03C8 018F  (..GRG.,.G.G:....)
4772 (0020) 0000 0000 FFFF 0000 58AC 58AC            (........X.X.    )
*** END OF PROCESS DUMP

            *** DUMP OF HEAP ***

4686 (0000) 4841 4C54 2043 414C 4C45 4420 2D20 5354  (HALT CALLED - ST)
4696 (0010) 4154 5553 203D 2032 3720 202E 494E 5055  (ATUS = 27  .INPU)
46A6 (0020) 5430 3420 0000 0000 0000 0000 0000 0000  (T04 ............)
46B6 (0030) 0000 0000 0000 0000 0000 0000 0000 0000  (................)
46C6 (0040) 0000 0000 0000 0000 0000 0000 0000 0000  (................)

46D8 (0000) 0A2E 5041 554C 412E 4D53 4700            (..PAULA.MSG.    )

46E6 (0000) 0000 0B04 0009 4686 0050 000C 0000 0000  (......F..P......)
46F6 (0010) 068D 0000 0000 46D8 0000 0000 0000 0000  (......F.........)
4706 (0020) 0000 0000 01FF                           (......          )

536C (0000) 082E 494E 5055 5430 3420                 (..INPUT04       )

5378 (0000) 0027 9106 4018 0000 0050 0000 0000 0000  (.´..@....P......)
5388 (0010) 0409 0000 0000 536C 0000 0000 0000 0000  (......S.........)
5398 (0020) 0000 0000 0000                           (......          )

53A0 (0000) 2035 3341 3020 2830 3030 3029 2032 3033  ( 53A0 (0000) 203)
53B0 (0010) 3520 3333 3333 2033 3333 3320 3230 3333  (0 3333 3333 2033)
53C0 (0020) 2033 3333 3320 3333 3230 2033 3233 3320  ( 3230 2033 3233 )
53D0 (0030) 3333 3230 2020 2820 3332 3330 2032 3033  (3033  ( 3230 203)
53E0 (0040) 3233 3020 3230 3329 0000 0000 0000 0000  (230 203)........)

53F2 (0000) 0B2E 5041 554C 412E 5445 4D50            (..PAULA.TEMP    )

5400 (0000) 0000 0B05 0009 486C 0050 0002 0000 0000  (......H..P......)
5410 (0010) 068D 0000 0000 53F2 0000 0000 0000 0000  (......S.........)
5420 (0020) 0000 0000 01FF                           (......          )
```

Figure 14-2. Abnormal Termination Dump Example (Sheet 2 of 2)

The process dump consists of a dump of the process record and of the stack. The process record is the structure that describes the resources associated with each process. The fields of the process record and the hexadecimal relative byte numbers of the fields are:

- PROCESSLIST, bytes 0 and 1. Link to next process. Processes form a circular linked list.

- DISPLAY, 16 element array, bytes 2 through 21. Each element is a pointer to the stack frame of the most recently called routine at a static nesting level, 1 through 16. Only updated by routines that contain nested routines.

- WP, bytes 22 and 23. Contents of WP, saved when process is not executing. Address of workspace applicable to error.

- PC, bytes 24 and 25. Contents of program counter, saved when process is not executing. Address of instruction applicable to error.

- ST, bytes 26 and 27. Contents of status register, saved when process is not executing.

- RS, bytes 28 and 29. Pointer to the process that caused execution of this process to be resumed.

- FLAGS, bytes 2A and 2B. When bit 0 is set to one and the heap space has been filled, the task terminates abnormally. When bit 0 is set to zero, the task continues to execute when the heap space is full. Bits 1 through 15 are not currently used.

- NONSTD, bytes 2C and 2D. Contains zero except when process is executing in a routine with neither TIP nor FORTRAN linkage. Contains the most recent workspace pointer of a routine with TIP or FORTRAN linkage when process is executing in a routine with neither TIP nor FORTRAN linkage. Dump of such a routine is not possible; dump begins with workspace pointed to by this address.

- STKBLOCK, bytes 2E and 2F. Pointer to the stack region of the process.

- SBNDRY, bytes 30 and 31. Stack boundary; largest value reached for top-of-stack. Used for stack overflow checks and stack usage calculations.

- STKMAX, bytes 32 and 33. First word beyond the stack region of the process.

- FTOP, bytes 34 and 35. Top-of-stack for FORTRAN; otherwise, zero. Used to implement reentrant FORTRAN and to call TIP procedures from FORTRAN.

- FILL, bytes 36 and 37. Not used.

- PHEAP, bytes 38 and 39. Address of a data block used by the heap manager.

- PTASK, bytes 3A and 3B. Address of a data block that contains task information.

- TERMCD, bytes 3C and 3D. Process termination code, as follows:

  $FFFF_{16}$ — Active process

  0000 — Normal termination

  Other values — Error codes as listed in paragraph E.3.

The dump of the stack follows the dump of the process record and consists of a variable number of stack frames, each of varying length. The 128 bytes at the top of the stack (above the topmost stack frame) are displayed under the heading:

TOP OF STACK

This area may contain pertinent information when another stack frame was being built as the error occurred, or when the stack frame for a routine that completed remains in this area of memory. Each stack frame begins with a heading, as follows:

DATA AREA FOR HALT$     LEVEL=2

A stack frame is referred to in the dump as a data area. HALT$ was executing when the dump was displayed. The level, 2, is the static nesting level. The stack frame contains five fixed fields. The fields of the stack frame and the hexadecimal relative byte numbers of the fields are:

- REGS, bytes 0 through 1F. The workspace (registers) used by the routine.
- SAVED, bytes 20 and 21. Saved display pointer. The routine being entered saves the old display pointer at its level in this field prior to storing its bottom-of-stack pointer in the display. Used only by routines that contain nested routines.
- CALLER, bytes 22 and 23. Workspace pointer of calling routine. Used only by routines that contain nested routines.
- RETADDR, bytes 24 and 25. Address to which routine will return. Used only by routines that contain nested routines.
- PRP, bytes 26 and 27. Process record pointer. Current process address (used with LUNO I/O). Not used with SCI I/O.

For functions, the next eight or ten bytes are reserved for the function result. Eight bytes are used for all types except DECIMAL, which requires ten bytes. Results of types that require fewer than eight bytes are stored left-justified in an eight-byte field. Beginning at the next byte (byte $28_{16}$ for procedures, byte $30_{16}$ or $32_{16}$ for functions) is a variable-length section that contains compiler-generated temporaries, local variables, parameters (arguments), and dynamic arrays.

The stack frames of the program and the active routines are displayed, followed by the displays of process GO$ and any other system processes. The displays of other processes are similar to that of the user process, except that the area of the stack above the top stack frame is shown only for the user process.

The stack frame for the main program contains two examples of the TIP File Descriptor. The TIP File Descriptor is built by TIP software and describes the file to the runtime support system. The fields of the descriptor and the hexadecimal relative byte numbers of the fields are:

- FLADR, bytes 0 and 1. The address of this file descriptor.
- NEXT, bytes 2 and 3. The index (based on zero) of the next character in the buffer to be read or written (Textfiles only).
- LAST, bytes 4 and 5. The index of the last nonblank character in the buffer (Textfiles only).

- BUFF, bytes 6 and 7. The address of the buffer containing the next record (Textfiles and sequential files only).

- FS, byte 8. The file state code. The file state codes are:

  — 00 Closed.
  — 01 Open for write.
  — 02 Open for read.
  — 03 End of file read.
  — 04 End of medium read (applies to SKIPFILES procedure).
  — 05 Beginning of medium read (applies to SKIPFILES procedure).

- F, byte 9. Flags. Bits 0 and 1 of the byte specify the file type as follows:

  — 00 Sequential file (byte value is $00_{16}$ through $3F_{16}$).
  — 01 Text file (byte value is $40_{16}$ through $7F_{16}$).
  — 10 Random file (byte value is $80_{16}$ through $BF_{16}$).

  Bit 7 specifies error termination. Set to 1 to terminate on error; set to 0 to return to user program on I/O open error. (Odd byte values terminate on all errors.)

- ELSIZE, bytes A and B. Element size in bytes.

- RECSIZE, bytes C and D. Logical record length.

- SCB, bytes E and F. Address of supervisor call block for I/O operations.

- FILE NAME, bytes 10 through 17. Logical file name as declared in TIP program.

- STATUS, bytes 1C and 1D. Status of last I/O operation (returned by operating system).

The TIP File Descriptor for the predefined file OUTPUT is at relative address $40_{16}$ of the main program stack frame. That of the predefined file INPUT is at relative address $60_{16}$ of the main program stack frame.

Following the process dumps is the dump of the heap. This contains the dynamic variables and records for the program. The runtime system uses dynamic variables and records that are in the heap. Dynamic variables and records of the user program, if any, are in the heap also.

**14.7.5 USE OF THE ABNORMAL TERMINATION DUMP.**The abnormal termination dump is very useful in analyzing errors, especially when the contents of variables show what caused the error. The stack frames of the program and active routines show the progress of the program up to the point at which the error occurred. The source listing in figure 9-1 is the listing of the program that terminated in the error of the example dump. It consists of program module DIGIO and procedure modules CCHAR and CINT. The dump of the stack includes a stack frame for DIGIO but none for CCHAR or CINT. Therefore, only DIGIO was active when the error occurred; the error occurred when one of the instructions of the main program module was executed.

The information provided in the source listing (figure 9-1) as a result of specifying the MAP option is essential for examining the contents of variables and arrays. The map of identifiers for a module shows the stack frame displacement of each variable or parameter of the module. The map of identifiers for DIGIO shows that array BUFF consists of 12 bytes starting at relative address $80_{16}$. In figure 14-2, the left column contains the hexadecimal address of the first byte displayed on each line.

The number in parentheses is the hexadecimal displacement of that byte from the start of the stack frame. The 12 bytes in relative bytes $28_{16}$ through $33_{16}$ of the stack frame of DIGIO contain zeros. The map of identifiers shows that variable I occupies 2 bytes at relative byte $36_{16}$. This location also contains zero. Variable NUM occupies 2 bytes at relative byte $34_{16}$, which also contain zero. If the error involved the data in any of these variables, locating the values of the variables in the stack frame enables the user to determine what went wrong.

The variables of routines may be parameters passed by value or parameters passed by reference. In the case of parameters passed by value, the stack frame contains the values of these parameters when the routine was called. In the case of parameters passed by reference, the stack frame contains the address of the parameter.

Information in the stack frames is useful in determining the point at which the error occurred. To trace execution back to the instruction that failed in the user program requires the source listing (figure 9-1), the reverse assembly listing (figure 13-1), the linking map (figure 14-3), and the abnormal termination dump (figure 14-2). Relative address $24_{16}$ (PC) of the process record contains the contents of the program counter when process GO$ received control for termination. In the example, this address contains $1592_{16}$. From the linking map, figure 14-3, the code at that address is found to be part of module MAIN, loaded at address 0000 and extending through address $3832_{16}$. Since the top stack frame in the stack is that of HALT$, the address in PC of the process record should be in that routine. The definitions section of the linking map shows the entry point of routine HALT$ to be $1582_{16}$, so the PC value stored is from routine HALT$.

The stack frame of HALT$ identifies the routine from which HALT$ received control. Relative address $24_{16}$ (RETADDR) contains either the address in the program counter when HALT$ received control, zero, or $FFFF_{16}$. In the example, address $24_{16}$ contains $1A44_{16}$, indicating that the standard category linkage was used; the address from the program counter is $1A44_{16}$. This address is also in module MAIN and should be in routine IO$ERR. The definitions section of the linking map shows the entry point of routine IO$ERR to be $1886_{16}$ and the entry point of the routine above IO$ERR to be $1A66_{16}$ (MAP$). The program counter address is in routine IO$ERR.

The stack frames of OPEN$ and REST$T can be checked in a similar manner to verify the path of control back through these routines. Relative address $1C_{16}$ contains $3A60_{16}$, indicating that control passed to routine REST$T from that point. The module list of the linking map shows that the main module of the user program, DIGIO, was loaded at address $39C2_{16}$ and occupies $126_{16}$ bytes. Address $3A60_{16}$ is address $9E_{16}$ relative to the load point $39C2_{16}$. The reverse assembly listing shows that relative address $9E_{16}$ is the address of an INC instruction that follows a BLWP instruction. That BLWP instruction transferred control to routine REST$T.

To reconstruct the events resulting in the termination, DIGIO called REST$T, implementing the RESET procedure call in the source program. REST$T called routine OPEN$ to open file INPUT. The open operation failed because the file assigned for input had not been created. OPEN$ called routine IO$ERR, which called HALT$. HALT$ called process GO$, which terminated the program and printed the dump.

```
 SDSLNK         3.2.2  78.275              11/06/78  13:56:15              PAGE    1
COMMAND LIST

FORMAT      IMAGE,REPLACE
LIBRARY     .TIP.OBJ
PHASE       0,DIGIO
INCLUDE     (MAIN)
INCLUDE     .PAULA.OBJ.DIGIO
END


 SDSLNK         3.2.2  78.275              11/06/78  13:56:15              PAGE    2
LINK MAP

CONTROL FILE = .PAULA.CTL.DIGIO

LINKED OUTPUT FILE = .PAULA.PGM

LIST FILE = .PAULA.LLST.DIGIO

NUMBER OF OUTPUT RECORDS = 76

OUTPUT FORMAT = IMAGE

LIBRARIES

NO   ORGANIZATION   PATHNAME

1    RANDOM          .TIP.OBJ
 SDSLNK         3.2.2  78.275              11/06/78  13:56:15              PAGE    3


PHASE 0, DIGIO    ORIGIN = 0000  LENGTH = 500E   (TASK ID = 4)   ENTRY=0020

MODULE    NO   ORIGIN    LENGTH      TYPE        DATE       TIME      CREATOR

MAIN      1     0000      38E2    INCLUDE,1   10/31/78   21:37:27   SDSLNK
$DATA     1     463E      055A
CCHAR     2     38E2      0076    INCLUDE     11/06/78   13:55:12   DXPSCL
CINT      3     3958      006A    INCLUDE     11/06/78   13:55:19   DXPSCL
DIGIO     4     39C2      0126    INCLUDE     11/06/78   13:55:34   DXPSCL
MESAG$DX  5     3AE8      00F6    LIBRARY,1   10/31/78   15:31:48   DXPSCL
P$TERM    6     3BDE      0050    LIBRARY,1   10/31/78   19:43:37   SDSMAC
SCIRTNS   7     3C2E      0784    LIBRARY,1   10/31/78   21:35:39   SDSLNK
$DATA     7     4B98      0424
INIT$1    8     43B2      0060    LIBRARY,1   10/31/78   15:31:33   DXPSCL
P$INIT    9     4412      0002    LIBRARY,1   10/31/78   17:45:21   SDSMAC
PB$INIT   10    4414      0002    LIBRARY,1   10/31/78   17:45:46   SDSMAC
REST$T    11    4416      0050    LIBRARY,1   10/31/78   15:50:34   DXPSCL
GET$CH    12    4466      002A    LIBRARY,1   10/31/78   17:41:20   SDSMAC
RDLN$     13    4490      00E4    LIBRARY,1   10/31/78   15:49:50   DXPSCL
GET$RCOR  14    4574      00CA    LIBRARY,1   10/31/78   15:48:18   DXPSCL
```

Figure 14-3. Linking Map of Example Program (Sheet 1 of 2)

```
COMMON      NO    ORIGIN     LENGTH

CUR$        1      4FBC       0018
HEAP$       1      4FD4       0010
SYS$MS      5      4FE4       0020
PARM$       1      5004       000A



                                             D E F I N I T I O N S

 NAME       VALUE NO      NAME       VALUE NO      NAME       VALUE NO      NAME       VALUE NO

*ABEND$     00AC   1     *ABND$0     00D4   1     *ABND$1     00CC   1     *ABND$2     009E   1
*BID$SV     2B00*  1      CCHAR      38F4   2      CINT       3968   3     *CLOSE$     0118   1
 CLS$       0172   1     *CLS$FI     01DA   1     *CMP$ST     020C   1     *CREAT$     022E   1
*CUR$       0380   1     *CUR$$      0F94   1      DIV$       038E   1     *DMPP$H     0416   1
*DSTR$$     03C8   1     *ENC$T      0BF2   1     *ENI$T      0CB2   1     *ENS$T      0E10   1
*ENT$       0EFE   1      ENT$1      0072   1      ENT$2      0EEE   1      ENT$M      0F08   1
 ENT$S      0F84   1     *ENX$T      0FCA   1     *EOF$WR     1178   1      EOLN$      1198   1
*FIND$S     31AA   1      FL$INI     133C   1     *FREE$      11F6   1      GET$CH     4466  12
 GET$RC     458C  14     *HALT$      1582   1     *HEAP$T     15CC   1      INIT$1     43CE   8
 IO$ERR     1886   1     *MAP$       1A66   1     *MARG$N     000E   1      MESAG$     3AFE   5
 MOV$4      1C3A   1     *MOV$5      1C38   1      MOV$6      1C36   1     *MOV$7      1C34   1
*MOV$8      1C32   1     *MOV$N      1C2C   1     *MSG$       0010   1     *NEW$       1C5A   1
 OPEN$      1F44   1     *OPN$FI     20B0   1      P$INIT     4412   9      P$TERM     3BF2   6
*PATCH$     0F1C   1      PB$INI     4414  10      PB$TER     4414  10     *PRT$ME     22C8   1
 PSCL$$     39FA   4     *PUT$RC     25F8   1      PUTCH$     25B4   1      RDLN$      44AA  13
 REST$T     4424  11     *RESUM$     275A   1      RET$1      0080   1      RET$2      0F9A   1
 RET$M      0FAC   1      RET$S      0FAE   1      REWND$     26C8   1      REWRT$     2702   1
*RSUMR$     2796   1      S$GTCA     42D8   7     *S$IADD     3C2E   7     *S$IASC     3CA2   7
*S$IDIV     3D44   7     *S$IMUL     3D84   7      S$INT      3E3E   7     *S$ISUB     3C56   7
 S$MAPS     3FD6   7     *S$NEW      4068   7      S$PARM     40F4   7      S$PTCA     4326   7

SDSLNK              3.2.2  78.275              11/06/78  13:56:15                PAGE   4
NAME       VALUE NO      NAME       VALUE NO      NAME       VALUE NO      NAME       VALUE NO

*S$RTCA    436A   7     *S$SCPY     411E   7      S$SETS     419C   7      S$STOP     4286   7
 SET$AC    295E   1     *SET$NA     2A20   1     *STACK$     2A4A   1     *STORE$     3182   1
 SVC$      2A6E   1      T$CC       000A*  1      T$EC       000C*  1     *T$INID     0004*  1
 T$MSG     000E*  1     *T$RNID     0003*  1     *T$STN      0005*  1     *T$TIB      4FBC   1
*TX$ERR    329A   1     *WRC$T      3462   1     *WREOF$     3548   1     *WRI$T      35A8   1
 WRLN$     3694   1      WRS$T      3716   1     *WRX$T      3806   1



****  LINKING COMPLETED
```

Figure 14-3. Linking Map of Example Program (Sheet 2 of 2)

**14.7.6 UNFORMATTED ABNORMAL TERMINATION DUMP.** The linking procedures describe the linking of an alternate abnormal termination dump routine. Figure 14-4 shows an example of the unformatted dump written by this routine.

The dump contains approximately the same information, printed in the order of memory addresses rather than in the process record, stack region, heap region order of the formatted abnormal termination dump. The four characters printed in the upper left corner of the unformatted dump are the address of the process record. The left column contains the hexadecimal addresses of the first words displayed on the lines of the dump. The numbers in parentheses are relative addresses within the blocks of data displayed in the dump. The next eight columns show the contents of eight words represented as hexadecimal numbers. The right column contains the contents of the same eight words as ASCII characters, with unprintable characters represented by periods (.). A group of three periods at the left on a line represents one or more lines identical to the preceding line (normally containing zeros).

With the process record address, the user can find the process record in the dump. The contents of the process record are listed in paragraph 14.7.4. The more useful items of information in the process record are:

- Address of the stack frame of the main program — relative address $0002_{16}$
- Contents of WP — relative address $0022_{16}$
- Contents of PC — relative address $0024_{16}$
- Address of the beginning of the stack region — relative address $002E_{16}$
- Termination code (listed in paragraph E.3) — relative address $003C_{16}$

To use the unformatted dump, note the address of the process record and locate the block that contains the process record. The DISPLAY array shows the addresses of the stack frames for the most recently called routines at each static nesting level. The contents of each stack frame are listed in paragraph 14.7.4. The compiler listing (when written with the MAP option) shows the stack frame relative address of each variable. Pointers in the stack frame identify stack frames of other routines that were called at each static nesting level. File descriptors that describe the files defined for the task are also located in the stack frame, as described in paragraph 14.7.4.

**14.7.7 TX990 ABNORMAL TERMINATION DUMP.** Tasks that are linked with TX990 as described in paragraph 14.3.7 may include a task that writes an abnormal termination dump. Figure 14-5 shows the abnormal termination dump written by this task. The dump consists of a message line, the process record block, the stack region, and the heap region.

The message line displays the time of day, the task identifier, the termination code, the program counter contents, the number of bytes of stack used, and the number of bytes of heap used. The termination code is one of the error codes listed in paragraph E.3. The code shown in the example represents an open error resulting from a REWRITE or EXTEND procedure. The status returned by TX990 is 01, indicating that the LUNO for the operation is illegal.

```
48AC

 3694(0000) 4841 4C54 2043 414C 4C45 4420 4543 5554     HALT CALLED ECUT
 36A4(0010) 494F 4E20 4245 4749 4E53 0000 0000 0000     ION BEGINS......
 36B4(0020) 0000 0000 0000 0000 0000 0000 0000 0000     ................
...
 36D4(0040) 0000 0000 0000 0000 0000 0000 0000 0000     ................

 36E6(0000) 092E 5359 534D 5347 3231                    ..SYSMSG21

 36F2(0000) 0000 0B04 0001 3694 0050 000C 0000 0000     ......6..P......
 3702(0010) 068D 0000 0000 36E6 0000 0000 0000 0000     ......6.........
 3712(0020) 0000 0000 01FF                              ......

 371A(0000) 48AC 442A 3816 FF00 FF00 FF00 FF00 FF00     H.D*8...........
 372A(0010) FF00 FF00 FF00 FF00 FF00 FF00 FF00 FF00     ................
 373A(0020) FF00 375E 040C D18F 48AC 8000 0000 375C     ..7^....H.....7\
 374A(0030) 3A38 3B66 0000 0000 3FE0 3FC8 FFFF 0000     :8;.....?.?.....

 375C(0000) 03DC 3AE6 0000 0000 0000 0000 0000 0000     ..:.............
 376C(0010) 371A 0000 375E 378A 0C94 371A 3746 02E6     7...7^7...7.7F..
 377C(0020) 018F 0000 0000 FFFF 0000 48AC 48AC 029A     ..........H.H...
 378C(0030) 0001 0000 3786 0000 0000 48AC 0000 0E6A     ....7.....H.....
 379C(0040) 378A 37E6 00CE 371A 375E 0424 442A FF00     7.7...7.7^.$D*..
 37AC(0050) 375E 0424 0000 48AC 0280 4455 4D50 4445     7^.$..H...DUMPDE
 37BC(0060) 4D4F 2020 4558 4543 5554 494F 4E20 4245     MO  EXECUTION BE
 37CC(0070) 4749 4E53 2E9C 0515 0000 02E6 0001 3786     GINS..........7.
 37DC(0080) 371A 0000 0003 0000 3FF0 0001 364A 3B68     7.......?...6J;.
 37EC(0090) 375C 3B66 040F 3B66 364A 3FE0 37E6 3816     7\;...;.6J?.7.8.
 37FC(00A0) 0158 446A 378A 1024 D18F 378A 378A 1024     .XD.7..$..7.7..$
 380C(00B0) 382E 48AC 0001 3828 2D3E 2D3E 381A 3B68     8.H...8(->;.8.;.
 381C(00C0) 382C 381C 00C4 0000 2000 43C6 3816 3846     8,8 ......C.8.8F
 382C(00D0) 01AC 446A 37E6 0000 01B2 37E6 37E6 0144     ..D.7.....7.7..D
 383C(00E0) 3852 375C 3B66 37FE 0000 3840 0001 0000     8R7\;.7...8@....
 384C(00F0) FFFF 386C 0000 0001 3842 3874 3846 3872     ..8.....8B8.8F8.
 385C(0100) 1B08 371A 3816 01EE 318F FFFF 000B FFFF     ..7.8...1.......
 386C(0110) 3872 446A 0000 2020 0000 3870 C58F 3868     8.D...  ..8...8.
 387C(0120) 3894 1B08 0001 37E6 3872 38A4 2238 371A     8.....7.8.8."87.
 388C(0130) 3846 05BC 318F C18F 000B FFFF 0000 4400     8F..1.........D.
 389C(0140) 43A2 0048 3870 0001 2EC2 4400 38C6 2238     C..H8.....D.8."8
 38AC(0150) 371A 3868 05BC 318F 2EC2 38A4 38CE 223E     7.8...1...8.8.">
 38BC(0160) 371A 3872 1BBA D18F 0000 3FF0 0000 000B     7.8.......?.....
 38CC(0170) 4400 3898 000B 0B86 0000 000B 38C6 38F0     D.8.........8.8.
 38DC(0180) 223E 371A 3894 1BBA D18F 3AE6 38D6 0000     ">7.8.....:.8...
 38EC(0190) 223E 36F2 5450 5554 2020 38C2 38E6 0009     ">6.TPUT  8.8...
 38FC(01A0) 3484 00F2 38EE 3918 223E 371A 38B4 32B2     4...8.9.">7.8.2.
 390C(01B0) D18F 36F2 5359 0000 5347 36F2 0000 0000     ..6.SY..SG6.....
 391C(01C0) 36E6 0000 0008 3100 36EF 36EE 0000 0000     6.....1.6.6.....
 392C(01D0) 0000 0000 0009 3920 39FA 35D2 0100 38E6     ......9 9.5...8.
 393C(01E0) 3458 C58F 0000 0000 FFFF 0000 5359 534D     4X..........SYSM
 394C(01F0) 5347 064F 5554 5055 5409 0000 0000 0000     SG.OUTPUT.......
 395C(0200) 0000 0000 0000 0000 0000 0000 0000 0000     ................
...
 399C(0240) 0000 0000 43F4 38C2 0057 0000 0653 5953     ....C.8..W...SYS
 39AC(0250) 4D53 0001 0000 39A0 39DE 2262 371A 38C6     MS....9.9.".7.8.
 39BC(0260) 35AE C18F 378A 38C6 FFFF 0000 38C2 000C     5...7.8.....8...
 39CC(0270) 43F2 43FE 4026 0000 0000 03CC 0000 0000     C.C.@&..........
 39DC(0280) 0001 0000 0000 0000 0000 0000 0000 0000     ................
 39EC(0290) 0000 0000 0000 0000 0000 0000 0000 36E6     ..............6.
 39FC(02A0) 391C 0014 0000 0000 0000 0000 0001 0000     9...............
 3A0C(02B0) 39FA 3A38 2262 371A 3920 35AE C18F 37D6     9.:8".7.9 5...7.
```

**Figure 14-4. Unformatted Abnormal Termination Dump Listing (Sheet 1 of 2)**

```
 3A1C(02C0) 3920 FFFF 0000 391C 000C 36E4 36F0 3650    9 ....9...6.6.6P
 3A2C(02D0) 0000 0000 0094 0000 0000 0001 0000 0000    ................
 3A3C(02E0) 0000 0000 0000 0000 0000 0000 0000 0000    ................
...
 3B5C(0400) 0000 0000 0000 0000 0000 0000              ............

 437A(0000) 0000 0000 0000 0000 0000 0000 0000 0000    ................
 438A(0010) 0405 0000 0000 0000 0000 0000 0000 0000    ................
 439A(0020) 0000 0000 0000                             ......

 43A2(0000) 2034 3341 3228 3030 3030 2920 3230 3334     43A2(0000) 2034
 43B2(0010) 2033 3334 3420 3334 3230 2033 3333 3420     3334 3420 3334
 43C2(0020) 3332 3330 2032 3033 3320 3333 3333 2033    3320 3333 3333 2
 43D2(0030) 3033 3320 2020 2033 3332 3020 3333 3333    333    333    33
 43E2(0040) 3320 2020 2033 3320 0000 0000 0000 0000    3    33 ........

 43F4(0000) 092E 4F55 5450 5554 3231                   ..OUTPUT21

 4400(0000) 0000 0B05 0001 3872 0050 0002 0000 0000    ......8..P......
 4410(0010) 068D 0000 0000 43F4 0000 0000 0000 0000    ......C.........
 4420(0020) 0000 0000 01FF                             ......

 4428(0000) 029C 482A 0000 0000 0000 0000 0000 0000    ..H*............
 4438(0010) 0292 44DA 442A 44AA 0720 48AC 4412 02E6    ..D.D*D.. H.D...
 4448(0020) 018F 0000 4412 02E6 0000 0000 0000 0000    ....D...........
 4458(0030) 0000 0000 0000 0000 0000 0000 0000 0000    ................
 4468(0040) 0000 446A 0016 004F 43A2 0141 0001 0050    ..D..<.OC..A...P
 4478(0050) 4400 4F55 5450 5554 2020 0001 0100 0000    D.OUTPUT  ......
 4488(0060) 0000 448A 0000 0000 0000 0041 0002 0000    ..D.........A....
 4498(0070) 437A 494E 5055 5420 2020 0001 0100 0000    C.INPUT   ......
 44A8(0080) 0000 0003 0001 0000 0000 0000 0000 0000    ................
 44B8(0090) 44DA 44A0 44AA 44D2 07FE 48AC 442A 02D2    D.D.D.D...H.D*..
 44C8(00A0) C58F 37D6 442A 02D2 0000 0300 494E 5055    ..7.D*......INPU
 44D8(00B0) 5420 2020 0001 0002 3B70 3FC8 44D2 44FC    T   ....;.?.D.D.
 44E8(00C0) 07E4 48AC 3B70 0A48 C58F 0000 0000 0000    ..H.;..H........
 44F8(00D0) 44E6 0000 225C 48AC 44AA 06F6 C58F 0000    D..."\H.D.......
 4508(00E0) 0000 FFFF 0000 44E0 0000 437A 0000 0000    ......D...C.....
 4518(00F0) 437A 4512 0086 0000 0000 0000 0000 0001    C.E.............
 4528(0100) 0000 4518 4556 2262 48AC 44E6 1E4C C58F    ..E.EV".H.D..L..
 4538(0110) 0000 0000 FFFF 0000 4512 0028 4378 43A0    ........E..(C.C.
 4548(0120) 4026 0000 0000 0352 0000 0000 0001 0000    @&.....R........
 4558(0130) 0000 0000 0000 0000 0000 0000 0000 0000    ................
...
 48A8(0480) 0000                                       ..

 48AC(0000) 371A 442A 44AA FF00 FF00 FF00 FF00 FF00    7.D*D...........
 48BC(0010) FF00 FF00 FF00 FF00 FF00 FF00 FF00 FF00    ................
 48CC(0020) FF00 44AA 0730 C58F 371A 8000 0000 4428    ..D..0..7.....D(
 48DC(0030) 4556 48AA 0000 0000 3FE0 3FC8 0300         EVH.....?.?...
```

**Figure 14-4. Unformatted Abnormal Termination Dump Listing (Sheet 2 of 2)**

```
                TASK        TERMINATION      PC       STACK USED     HEAP USED
TIME OF DAY  IDENTIFIER        CODE       CONTENTS  (HEXADECIMAL)  (HEXADECIMAL)

 00:03:55    TASK 0021     CODE=F601      PC=1C76     STK=011C       HEAP=0026
     75F4(0000) 75F4 7644 0000 0000 0000 0000 0000 0000   ...D............ \
     7604(0010) 0000 0000 0000 0000 0000 0000 0000 0000   ................  > PROCESS
     7614(0020) 0000 76F4 1C76 C00F 0000 8000 0000 7642   ...............B  | RECORD
     7624(0030) 775E 7B82 0000 0000 7632 75EE 0601        .^.......2....   /
                         WP AT TERMINATION
     7644(0000) 0000 0000 117A 0000 0000 7702 7684 0000   ................ \
     7654(0010) 7B42 7644 76CA 17D4 75F4 762C 0E6E D10F   .B.D.........,.. |
     7664(0020) 0000 0000 0000 75F4 0000 0000 0000 0000   ................ |
     7674(0030) 0000 0000 0000 0000 0000 0000 0000 0000   ................ |
     7684(0040) 7684 0000 0000 0000 0041 0002 0000 7BA2   .........A...... |
     7694(0050) 4F55 5450 5554 2020 0001 0100 0001 0000   OUTPUT  ........ |
     76A4(0060) 0000 0000 0000 0000 0000 0000 0000 0000   ................ |
   ...                                                                     |
     76C4(0080) 0000 0000 0000 0024 7BA2 7684 76A4 0000   .......$........ |
     76D4(0090) 0000 0000 76FA 769A 76CA 76F4 1BAE 75F4   ................ |
     76E4(00A0) 7644 11C4 7684 0000 7644 FFFF 75F4 7684   .D.......D...... |
     76F4(00B0) 7B42 0001 5554 2020 0001 0002 0000 769A   .B..UT  ........  > STACK REGION
     7704(00C0) 7756 76F4 7726 18E8 75F4 76CA 17F0 7684   .V...&.......... |
     7714(00D0) 7BC8 7704 FFFF 75F4 7684 0001 1454 0001   .............T.. |
     7724(00E0) 0050 0504 FA00 75F4 0001 0384 0200 8CA0   .P.............. |
     7734(00F0) 0000 2120 7726 7752 18EE 75F4 76F4 1C76   ..! .&.R........ |
     7744(0100) C00F 0000 76F4 0000 75F4 7684 0000 7724   ...............$ |
     7754(0110) 7722 0000 0000 0000 0000 0000 7BA2 0000   ."..............  |
     7764(0120) 0000 0000 0000 0000 0000 0000 775E 7788   ..............^.. |
     7774(0130) 1B08 75F4 7726 1E58 C00F 0000 0000 0000   .....&.X........ |
     7784(0140) 75F4 7BA2 0000 0000 0000 0000 0000 0000   ................ |
     7794(0150) 0000 0000 0000 0000 0000 0000 0000 0000   ................ |
   ...                                                                     |
     7B74(0530) 0000 0000 0000 0000 0000 0000 0000        ..............   /

     7BA2(0000) 0001 003E 4008 0000 0050 0000 0000 0000   ...>@....P...... \
     7BB2(0010) 0005 0000 0000 0000 0000 0000 0000 0000   ................ |
     7BC2(0020) 0000 0000 0000 0000 0000 0000 0000 0000   ................  > HEAP REGION
   ...                                                                     |
     7DE2(0240) 0000 0000 0000 0000 0000 0000 0000 0000   ................ /
(A) 140780
```

Figure 14-5. TX990 Abnormal Termination Dump Listing

In the three blocks of memory listed, the left column contains a four-digit hexadecimal number followed by another four-digit hexadecimal number in parentheses. The numbers are the address of the first word of data displayed on the line. The first number is the absolute address; the number in parentheses is a relative address, relative to the first word of the block. The lines that consist of a group of three periods (.) represent one or more lines of data identical to the data on the preceding line (normally lines of words that contain zeros). To the right of the addresses are eight sets of four hexadecimal digits, each representing the contents of a memory word. The right column of the block shows the contents of the eight words of memory represented as ASCII characters. Unprintable characters are represented by periods.

The dump displays the contents of the process record, the stack region, and the heap region. The contents of the process record are listed in paragraph 14.7.4. Because some of the pertinent information in the process record is printed in the message, the main item of interest is the contents of the WP in relative location $0022_{16}$.

The task used in printing the example shown in figure 14-5 is one that has a main routine. When execution begins in a procedure, the first word of array DISPLAY (relative address $0002_{16}$ in the process record) contains zero.

The contents of each stack frame are listed in paragraph 14.7.4. Using the contents of the WP, locate the stack frame for the routine that was executing when the abnormal termination failed. Using the links in that stack frame, identify the stack frames that were active. Stack frames may contain file descriptors that describe the files used by the task. The file descriptor is described in paragraph 14.7.4.

# SECTION XV

# COMPUTER-DEPENDENT SOURCE CODE PREPARATION TECHNIQUES

## 15.1 GENERAL

The source code preparation information in this section is not part of the TIP language but is computer dependent. It includes descriptions of the routines provided for direct CRU I/O and for interface with the 990 operating systems, and the guidelines for preparing source code for the optional linking methods described in Section XIV. Paragraphs in this section describe:

- Computer-dependent direct CRU I/O routines
- Operating system interface routines
- Preparing source code for a program with a dummy main routine
- Specifying and changing LUNOs for LUNO I/O
- Writing multiple task source code
- Writing source code for stand-alone tasks
- Writing source code for minimal runtime code

## 15.2 DIRECT CRU I/O ROUTINES

The direct CRU I/O routines provide I/O operations with CRU devices. These routines may be used for I/O operations with CRU devices that are not supported by the operating system or by stand-alone TIP tasks. The CRU routines correspond to the CRU instructions of assembly language, as follows:

- Procedure $LDCR, corresponding to an LDCR instruction
- Procedure $SBO, corresponding to an SBO instruction
- Procedure $SBZ, corresponding to an SBZ instruction
- Procedure $STCR, corresponding to an STCR instruction
- Function $TB, corresponding to a TB instruction

**NOTE**

The CRU base address for each of these routines is the CRU (hardware) address. Each routine multiplies the value by 2 and places the product in workspace register 12 for the operation.

**15.2.1 PROCEDURE $LDCR.** A routine that calls procedure $LDCR to output data to a CRU device must declare the procedure among the declarations for the routine. The declaration for procedure $LDCR is as follows:

```
PROCEDURE $LDCR(BASE,WIDTH,VALUE:INTEGER); EXTERNAL;
```

Assuming that B, W, and V have been defined as integer variables, the following example shows a call to $LDCR:

```
B:= #200;                    (*SET BASE TO #200*)
W:= 8;                       (*WRITE A CHARACTER*)
V:= #41;                     (*CHARACTER IS #41 (A)*)
$LDCR(B,W,V);                (*OUTPUT CHARACTER*)
```

**15.2.2 PROCEDURE $SBO.** A routine that calls procedure $SBO to set a CRU bit to one must declare the procedure among the declarations for the routine. The declaration for procedure $SBO is as follows:

```
PROCEDURE $SBO(BASE:INTEGER); EXTERNAL;
```

Assuming that ADDR has been declared as an integer variable, the following example shows a call to $SBO:

```
ADDR:= #400;                 (*SET BASE TO #400*)
ADDR:= ADDR + 4;             (*ADD DISPLACEMENT*)
$SBO(ADDR);                  (*SET BIT TO ONE*)
```

**15.2.3 PROCEDURE $SBZ.** A routine that calls procedure $SBZ to set a CRU bit to zero must declare the procedure among the declarations for the routine. The declaration for procedure $SBZ is as follows:

```
PROCEDURE $SBZ(BASE:INTEGER); EXTERNAL;
```

Assuming that BITOUT has been declared as an integer variable, the following example shows a call to $SBZ:

```
BITOUT:= #400;               (*SET BASE TO #400*)
BITOUT:= BITOUT + 2;         (*ADD DISPLACEMENT*)
$SBZ(BITOUT);                (*SET BIT TO ZERO*)
```

**15.2.4 PROCEDURE $STCR.** A routine that calls procedure $STCR to input data from a CRU device must declare the procedure among the declarations for the routine. The declaration for procedure $STCR is as follows:

```
PROCEDURE $STCR(BASE,WIDTH:INTEGER; VAR VALUE: INTEGER);
EXTERNAL;
```

The following is an example of the use of the $STCR procedure:

```
CONST BS1 = #200;
    SIZE = 8;
VAR INCHR: INTEGER;
```

```
        .
        .
        .
$STCR(BS1,SIZE,INCHR);          (*READ CHARACTER*)
```

**15.2.5 FUNCTION $TB.** A routine that calls function $TB to input a bit from a CRU device must declare the function among the declarations for the routine. The declaration for function $TB is as follows:

```
FUNCTION $TB(BASE: INTEGER):BOOLEAN; EXTERNAL;
```

Assuming that ADD has been declared as integer and BUSY has been declared as boolean, the following is an example of the use of function $TB:

```
ADD:= #200;                     (*SET BASE TO #200*)
ADD:= ADD + 8;                  (*ADD DISPLACEMENT *)
BUSY:= $TB(ADD);                (*SET BUSY TO VALUE OF CRU INPUT*)
```

## 15.3 OPERATING SYSTEM INTERFACE ROUTINES

The TIP software package includes routines that are not part of the language but that provide access to additional features of the 990 operating systems, as follows:

- I/O routines
- Identification functions
- Time and date procedures
- Task control procedures
- Message handling procedures
- System common access procedures
- Semaphore procedures
- Supervisor call procedure

**15.3.1 I/O ROUTINES.** Five I/O routines included with the TIP software provide the following capabilities:

- Obtaining the value of a synonym
- Assigning a synonym and its value
- Specifying the access name of a file
- Defining the LUNO for a file
- Obtaining the device type for a textfile

**15.3.1.1 Procedure FIND$SYN.** Procedure FIND$SYN provides a means for the program to obtain the value of a synonym. The procedure is a Pascal interface to system routine S$MAPS, described in the *Model 990 Computer DX10 Operating System Release 3 Reference Manual, Volume V.* To use FIND$SYN, the user must declare type STRING and must declare the procedure externally. The user must place the maximum length of the value string (as limited by the size of the array into which it is placed) into element 0 of the value string. The user then calls the procedure.

The declaration of type STRING is as follows:

```
TYPE STRING = PACKED ARRAY [0. .N] OF CHAR;
```

where N represents an integer constant chosen by the user.

The declaration for procedure FIND$SYN is as follows:

```
PROCEDURE FIND$SYN (VAR SYNONYM, VALUE: STRING); EXTERNAL;
```

The procedure has two parameters: a string that contains the synonym for which the value is to be obtained, and a string into which FIND$SYN places the value. FIND$SYN accesses the Terminal Communications Area (TCA), searches the synonym table for the specified synonym, and returns the value. If the synonym is found, the number of characters in the value is placed in element 0 of the value string, and the characters of the value are placed in elements 1 through N of the value string. If the synonym is not found, zero is placed in element 0 of the value string. When a routine that uses LUNO I/O calls procedure FIND$SYN, zero is placed in element 0 of the value string.

An example of the use of procedure FIND$SYN is as follows:

```
CONST  SL = 80;                   (*MAXIMUM LENGTH OF STRING*)
       SYNA = 'ONE';              (*SYNONYM AS A CONSTANT*)
       N = 3;                     (*SIZE OF SYNONYM*)
TYPE   STRING = PACKED ARRAY [0. .SL] OF CHAR;
                                  (*REQUIRED TYPE*)
VAR    SYN: STRING;               (*FOR SYNONYM PARAMETER*)
       ACCESS_NAME: STRING: (*FOR VALUE PARAMETER*)
              .
              .
              .
PROCEDURE FIND$SYN (VAR SYNONYM, VALUE: STRING); EXTERNAL;
                                  (*EXTERNAL DECLARATION*)
              .
              .
              .
BEGIN
              .
              .
              .
  FOR K:= 1 TO N DO
    SYN[K]:= SYNA[K];             (*SET UP SYNONYM NAME STRING*)
  SYN [0] := CHR(N);              (*SET LENGTH OF STRING*)
  ACCESS_NAME [0]:= CHR(SL);  (*SET MAXIMUM LENGTH*)
  FIND$SYN(SYN,ACCESS_NAME);
                                  (*CALL FIND$SYN*)
    IF ORD(ACCESS_NAME [0]) = 0 THEN
      HALT;                            (*TERMINATE ABNORMALLY IF NO
                                         SYNONYM*)
                    .
                    .
                    .
```

**15.3.1.2 Procedure STORE$SYN.** Procedure STORE$SYN provides a means for the program to assign a value to a DX10 synonym. The procedure is a Pascal interface to system routines S$GTCA, S$SETS, and S$PTCA, described in the *Model 990 Computer DX10 Operating System Release 3 Reference Manual, Volume V.* To use STORE$SYN, the user must declare type STRING as described in paragraph 15.3.1.1 and must declare the procedure externally. The user then calls the procedure.

The declaration for procedure STORE$SYN is as follows:

```
PROCEDURE STORE$SYN (VAR SYNONYM, VALUE: STRING); EXTERNAL;
```

The procedure has two parameters: a string that contains the synonym and a string that contains the value. STORE$SYN reads the TCA from disk, adds the synonym and value to the synonym table, and writes the updated data to the disk. This procedure applies only to programs executing under DX10.

An example of the use of procedure STORE$SYN is as follows:

```
CONST  SL = 80;                   (*MAXIMUM LENGTH OF STRING*)
       SYNA = 'ONE';              (*SYNONYM AS A CONSTANT*)
       ACNM = '.INPUT.TEST';      (*PATHNAME AS A CONSTANT*)
       N = 3;                     (*SIZE OF SYNONYM*)
       NP = 11;                   (*SIZE OF PATHNAME*)
TYPE   STRING = PACKED ARRAY [0. .SL] OF CHAR;
                                  (*REQUIRED TYPE*)
VAR    SYN: STRING;               (*FOR SYNONYM PARAMETER*)
       ACCESS_NAME: STRING;       (*FOR VALUE PARAMETER*)
                 .
                 .
                 .
PROCEDURE STORE$SYN (VAR SYNONYM, VALUE: STRING); EXTERNAL;
                                  (*EXTERNAL DECLARATION*)
                 .
                 .
                 .
BEGIN
                 .
                 .
                 .
  FOR K:= 1 TO N DO
    SYN [K]:= SYNA[K];            (*SET UP SYNONYM NAME STRING*)
  SYN[0] := CHR(N);               (*SET LENGTH OF STRING*)
  FOR K:= 1 TO NP DO
      ACCESS_NAME[K]:= ACNM[K];
                                  (*SET UP VALUE STRING*)
```

```
ACCESS_NAME[0] := CHR (NP);
                                    (*SET LENGTH OF VALUE STRING*)
STORE$SYN(SYN,ACCESS_NAME);
                                    (*CALL STORE$SYN*)
                  .
                  .
                  .
```

**15.3.1.3 Procedure SET$ACNM.** Procedure SET$ACNM specifies the access name for a Pascal file. Procedure SET$ACNM is called prior to the RESET or REWRITE procedure that opens the file, to override the default synonym. This procedure is more general than the SETNAME procedure, which assigns a synonym of eight characters or less. To use SET$ACNM, the user must declare type STRING as described in paragraph 15.3.1.1 and must declare the procedure externally. The user then calls the procedure.

The declaration for procedure SET$ACNM is as follows:

```
PROCEDURE SET$ACNM (VAR F:<ft>; VAR ACNM: STRING); EXTERNAL;
```

The ft entry in the PROCEDURE declaration may be an identifier of a file type (FILE, RANDOM FILE, or TEXT) or a user-defined file type. The first parameter is a file name, and the second is a string that contains the access name. The access name parameter may not be a synonym. SET$ACNM associates the specified access name with the specified file.

When a routine that uses LUNO I/O calls procedure SET$ACNM, SET$ACNM associates the access name with the specified file and causes the RESET or REWRITE operation that opens the file to assign the LUNO to the access name. Unless a LUNO has been defined for the file using the SETLUNO procedure (described in paragraph 15.3.1.4), the default LUNO is used. It is the user's responsibility to define LUNOs so that two files that are open simultaneously in a program do not use the same LUNO. When the RESET or REWRITE operation assigns the LUNO, the CLOSE operation releases the LUNO. A LUNO assigned by the task is a task local LUNO when the task executes under DX10 or RX990; it is a global LUNO when the task executes under TX990.

Procedure FIND$SYN may be used to identify a synonym and obtain its value for use in a call to SET$ACNM. The following example shows the use of both procedures:

```
CONST  SL = 80;                 (*MAXIMUM STRING LENGTH*)
TYPE   STRING = PACKED ARRAY [0. .SL] OF CHAR;
                                (*REQUIRED TYPE*)
VAR    SYN: STRING;             (*FOR SYNONYM PARAMETER*)
       ACCESS_NAME: STRING; (*FOR VALUE PARAMETER*)
       DATA_FILE: TEXT;         (*FOR FILE PARAMETER*)
       N: INTEGER;              (*FOR SIZE OF SYNONYM*)
                  .
                  .
                  .
PROCEDURE FIND$SYN (VAR SYNONYM, VALUE: STRING); EXTERNAL;
                                (*EXTERNAL DECLARATIONS*)
PROCEDURE SET$ACNM (VAR F: TEXT; VAR ACNM: STRING); EXTERNAL;
                  .
                  .
                  .
```

```
BEGIN
                  .
                  .
                  .
RESET(INPUT);                    (*OPEN INPUT FILE*)
N:= 1;                           (*INITIALIZE INDEX*)
WHILE NOT EOLN DO BEGIN          (*READ SYNONYM*)
        READ(SYN[N]);
        N:= N+1
        END;
SYN[0]:= CHR(N-1);               (*SET LENGTH OF STRING*)
ACCESS_NAME[0]:= CHR(SL);        (*SET MAXIMUM LENGTH*)
FIND$SYN(SYN,ACCESS_NAME);       (*CALL FIND$SYN*)
IF ORD(ACCESS_NAME[0]) = 0 THEN
                                 (*IF NOT SYNONYM*)
    FOR K:= 0 TO N DO            (*COPY TO ACCESS NAME*)
        ACCESS_NAME[K]:= SYN[K];
SET$ACNM(DATA_FILE, ACCESS_NAME);
                                 (*SET FILE ACCESS NAME*)
RESET(DATA_FILE);                (*OPEN FILE*)
READ(DATA_FILE,.....             (*READ FILE*)
                  .
                  .
                  .
```

**15.3.1.4 Procedure SETLUNO.** Procedure SETLUNO defines a LUNO for a file in a task that uses LUNO I/O. LUNOs defined by procedure SETLUNO replace the default LUNOs, which are the ASCII code of the first letter of the file name ($41_{16}$ through $5A_{16}$ and $24_{16}$). The user must call SETLUNO prior to calling SET$ACNM, RESET, or REWRITE. The syntax for the procedure call is:

```
SETLUNO(<file>,<unit number>)
```

The parameters for the call are file, the file identifier, and unit number, an integer value in the range of 0 through 254. SETLUNO only defines the LUNO to be used to access the file; it does not assign the LUNO. A LUNO may be assigned automatically, as described in paragraph 15.3.1.3 or may be assigned using an SCI or OCP command prior to executing the task. When procedure SETLUNO is called by a task using SCI synonym I/O, no operation is performed.

The following example shows the use of procedure SETLUNO to define LUNO $6E_{16}$ for file FILE2:

```
SETLUNO (FILE2,#6E)
```

**15.3.1.5 Function DEV$TYPE.** Function DEV$TYPE returns the device type assigned to a file. Table 15-1 lists the device type codes and their meanings. To use DEV$TYPE, the user must declare the function externally, as follows:

```
FUNCTION DEV$TYPE (VAR F:TEXT): INTEGER; EXTERNAL;
```

The following example shows the use of function DEV$TYPE:

```
IF DEV$TYPE(OUTPUT)>0 THEN WRITELN(OUTPUT)
```

**Table 15-1. Device Type Codes**

| Hexadecimal Device Type | Device |
|---|---|
| 0 | Dummy device |
| 1 | Teleprinter |
| 2 | Line printer |
| 3 | Cassette |
| 4 | Card reader |
| 5 | Video display terminal |
| 6 | Disk |
| 7 | Communication device |
| 8 | Magnetic tape |
| FF | Disk file |

**15.3.2 IDENTIFICATION FUNCTIONS.** The identification functions return a requested identifier to the calling task. Function TASKID returns the runtime task identifier, and function STATIONID returns the identifier of the station (terminal) associated with a task.

**15.3.2.1 Function TASKID.** Function TASKID obtains the runtime identifier assigned to the task by DX10 or the installed task number when the task executes under TX990 or RX990. To use TASKID, the user must declare type BYTE and must declare function TASKID externally. The user may then call the function.

The declaration for type BYTE is as follows:

```
    TYPE     BYTE = 0. .#FF;
```

The declaration for function TASKID is as follows:

```
    FUNCTION TASKID ():BYTE; EXTERNAL;
```

There is no parameter for the function; the calling task is assumed to be the task for which the identifier is requested. The following is an example of the use of function TASKID:

```
    IDENT := TASKID;
```

**15.3.2.2 Function STATIONID.** Function STATIONID obtains the station identifier of the station associated with the task. To use STATIONID, the user must declare type BYTE as described in paragraph 15.3.2.1 and must declare function STATIONID externally. The user may then call the function.

The declaration for function STATIONID is as follows:

```
FUNCTION STATIONID (): BYTE; EXTERNAL;
```

There is no parameter for the function; the calling task is assumed to be the task for which the station identifier is requested. The following is an example of the use of function STATIONID:

```
TRMNL := STATIONID
```

**15.3.3 TIME AND DATE PROCEDURES.** The time and date procedures supply time and date information and time delay suspensions to the task. Procedure ITIME returns integer values representing the hour, minute, and second. Procedure IDATE returns integer values representing the year, month, and day of the month. Procedure IDATE returns integer values representing the year, month, and day of the month. Procedure DELAY places the calling task in a time delay suspension for a specified period.

**15.3.3.1 Procedure ITIME.** Procedure ITIME obtains the time of day in a record declared for the hour, minute, and second values. To use procedure ITIME, the user must declare record TIME_TYPE and must declare procedure ITIME externally. The user may then call the procedure.

The declaration for record TIME_TYPE is as follows:

```
TYPE  TIME_TYPE =RECORD
                   HOUR:     0. .24;
                   MINUTE:   0. .59;
                   SECOND:   0. .59
                  END;
```

The declaration for procedure ITIME is as follows:

```
PROCEDURE ITIME (VAR TIME: TIME_TYPE); EXTERNAL;
```

The parameter for the procedure is a record into which the procedure places integer values for the hour, minute, and second. The following is an example of a call to procedure ITIME, which assumes that variable NOW is a record of type TIME_TYPE:

```
ITIME (NOW);
```

**15.3.3.2 Procedure IDATE.** Procedure IDATE obtains the date in a record declared for the year, month, and day values. To use procedure IDATE, the user must declare record DATE_TYPE and must declare procedure IDATE externally. The user may then call the procedure.

The declaration for record DATE_TYPE is as follows:

```
TYPE  DATE_TYPE =RECORD
                   YEAR:     INTEGER;
                   MONTH:    1. .12;
                   DAY:      1. .31
                  END;
```

The declaration for procedure IDATE is as follows:

```
PROCEDURE IDATE (VAR DATE: DATE_TYPE);
```

The parameter for the procedure is a record into which the procedure places integer values for the year, month, and day of the month. The following is an example of a call to procedure IDATE, which assumes that variable TODAY is a record of type DATE_TYPE:

```
IDATE (TODAY);
```

**15.3.3.3 Procedure DELAY.** Procedure DELAY suspends the calling task for a period of time that is a multiple of 50 milliseconds. The interval is specified as a number of milliseconds, which is rounded off to the nearest 50-millisecond multiple. To use procedure DELAY, the user must declare the procedure externally. The user may then call the procedure.

The declaration for procedure DELAY is as follows:

```
PROCEDURE DELAY (MILLISECONDS: LONGINT); EXTERNAL;
```

The parameter for the procedure is an integer representing a number of milliseconds. The procedure rounds this value to the nearest number of 50-millisecond periods and suspends the calling task for that interval. The following is an example of a call to procedure DELAY that suspends the task for 100 milliseconds:

```
DELAY (85);
```

**15.3.4 TASK CONTROL PROCEDURES.** The task control procedures allow cooperating tasks by providing a means of bidding tasks, suspending the calling task, and terminating unconditional suspension of a task. Procedure BID initiates execution of another task. Procedure SUSPEND suspends the calling task unconditionally. Procedure ACTIVATE terminates the unconditional suspension of the specified task, causing the task to resume execution.

**15.3.4.1 Procedure BID.** Procedure BID initiates execution of a specified task and passes parameters to the task. To use procedure BID, the user must declare type BYTE and must declare procedure BID externally. The user may then call the procedure.

The declaration for type BYTE is as follows:

```
TYPE    BYTE = 0. .#FF;
```

The declaration for procedure BID is as follows:

```
PROCEDURE BID    (PROGRAM_FILE_LUNO: BYTE;
                 INSTALLED_TASK_ID: BYTE;
                 PARAMETERS: LONGINT;
                 VAR RUN_TIME_TASK_ID: BYTE;
                 VAR STATUS: INTEGER); EXTERNAL;
```

The five parameters for the procedure are:

- LUNO of the program file on which the task is installed (DX10 only; ignored by TX990 or RX990)
- Installed task identifier (DX10) or task identifier (RX990 or TX990)

- Parameters for the task (any type having a length of four bytes)
- A parameter into which procedure BID places the runtime ID of the task
- A parameter into which procedure BID places a status code

When the task being bid was linked without library .TIP.MINOBJ, the parameters specify stack and heap requirements. The first two bytes specify stack size in bytes, and the second two bytes specify heap size in bytes. When the task being bid is linked with library .TIP.MINOBJ and begins execution in a procedure (main routine omitted), the parameters are passed to the called procedure and are accessible to the procedure by the type declared in the procedure declaration.

The status code is set to zero to indicate successful bidding of the task. Otherwise, the status code indicates an error code that is defined for the operating system. The DX10 error codes range from $2B01_{16}$ through $2BFE_{16}$. The TX990 and RX990 error codes range from $0500_{16}$ through $05FF_{16}$.

The following is an example of a call to procedure BID to initiate task $1D_{16}$ on the program file assigned to LUNO $3F_{16}$. The task requires $800_{16}$ bytes of stack and $200_{16}$ bytes of heap. TSKID and TSTAT are variables into which the runtime identifier and status are to be stored:

```
BID (#3F,#1D,#08000200,TSKID,TSTAT);
```

**15.3.4.2 Procedure SUSPEND.** Procedure SUSPEND places the calling task in unconditional suspension. To use procedure SUSPEND, the user must declare the procedure externally. The user may then call the procedure.

The declaration for procedure SUSPEND is as follows:

```
PROCEDURE SUSPEND; EXTERNAL;
```

The procedure has no parameter; it assumes that the calling task is the task to be suspended. The following is an example of a call to procedure SUSPEND:

```
SUSPEND;
```

**15.3.4.3 Procedure ACTIVATE.** Procedure ACTIVATE terminates unconditional suspension of a specified suspended task. To use procedure ACTIVATE, the user must declare type BYTE as described in paragraph 15.2.5.1 and must declare procedure ACTIVATE externally. The user may then call the procedure.

The declaration for procedure ACTIVATE is as follows:

```
PROCEDURE ACTIVATE          (RUN_TIME_TASK_ID: BYTE;
                             VAR STATUS: INTEGER); EXTERNAL;
```

The parameters are the runtime task identifier and a parameter into which the procedure places a status code. The runtime task identifier is not necessarily the installed task identifier; it is returned by a call to procedure BID when the task is initiated by procedure BID. Each task can obtain its own runtime task identifier by calling function TASKID. When the activation of the task is successful, the procedure returns zero in the variable named as the second (status) parameter. Otherwise, the procedure returns a system error code ($0700_{16}$ through $07FF_{16}$).

The following is an example of a call to activate the task whose runtime identifier is $3C_{16}$, placing the status in variable TSTAT:

```
ACTIVATE (#3C,TSTAT);
```

**15.3.5 MESSAGE HANDLING PROCEDURES.** The message handling procedures send and receive messages between tasks and delete queued messages. Procedure PUTMSG places a message in a message queue. Procedure GETMSG obtains the next message from a queue. Procedure PRGMSG purges a message queue of any undelivered messages.

**15.3.5.1 Procedure PUTMSG.** Procedure PUTMSG places a specified message on a specified message queue. To use procedure PUTMSG, the user must declare type BYTE and must declare procedure PUTMSG externally. The user may then call procedure PUTMSG.

The declaration for type BYTE is as follows:

```
TYPE  BYTE = 0. .#FF;
```

The declaration for procedure PUTMSG is as follows:

```
PROCEDURE PUTMSG      (QUEUE: BYTE;
                       VAR MESSAGE: PACKED ARRAY [1. .?] OF CHAR;
                       VAR STATUS: BYTE); EXTERNAL;
```

The queue parameter is an arbitrarily assigned number that identifies a message queue. A convenient method of assigning queue numbers that ensures a unique number is to use the runtime identifier of the task to which the message is directed as the queue number. This technique is used by some existing software, particularly the Sort/Merge package; by using the method, the user avoids assigning queue numbers that are already in use. When a task initiates another task by calling the BID procedure, the runtime identifier of the initiated task is returned. A task can obtain its own runtime identifier by calling function TASKID.

The message parameter is the variable that contains the literal message to be placed in the queue. The status parameter is a variable into which the procedure places the status code. The status code is zero when the message is enqueued successfully. The status is a nonzero value when the queue is full.

The following is an example of a call to procedure PUTMSG; the queue is $3C_{16}$, the message is contained in variable MSG1, and the status is returned in variable MSTAT:

```
PUTMSG (#3C,MSG1,MSTAT);
```

**15.3.5.2 Procedure GETMSG.** Procedure GETMSG obtains the next message from the specified queue. To use procedure GETMSG, the user must declare type BYTE as described in paragraph 15.3.5.1 and must declare procedure GETMSG externally.

The declaration for procedure GETMSG is as follows:

```
PROCEDURE GETMSG      (QUEUE: BYTE;
                       VAR BUFFER: PACKED ARRAY [1. .?] OF CHAR;
                       VAR LENGTH, STATUS: BYTE); EXTERNAL;
```

The queue parameter is the queue number that other tasks use for messages directed to the calling task. The runtime task identifier is often used; it may be obtained by calling function TASKID. The buffer parameter is the variable into which the procedure places the received message. The length parameter is the variable into which the procedure places the number of bytes in the message. The status parameter is the variable into which the procedure places the status code: zero when the message is received, and a nonzero value when the queue is empty.

The following is an example of a call to procedure GETMSG; the queue is $3C_{16}$, the message buffer is variable MSG2, the variable for the length is MLGTH, and the status variable is MSTAT:

```
GETMSG (#3C,MSG2,MLGTH,MSTAT);
```

**15.3.5.3 Procedure PRGMSG.** Procedure PRGMSG purges a specified message queue by discarding any messages in the queue, leaving the queue empty. To use procedure PRGMSG, the user must declare type BYTE as described in paragraph 15.3.5.1 and must declare procedure PRGMSG externally. The user may then call procedure PRGMSG.

The declaration for procedure PRGMSG is as follows:

```
PROCEDURE PRGMSG (QUEUE: BYTE); EXTERNAL
```

The queue parameter is the queue number that identifies the queue to be purged. The following is an example of a call to procedure PRGMSG for message queue $3C_{16}$:

```
PRGMSG (#3C);
```

**15.3.6 SYSTEM COMMON ACCESS PROCEDURES.** The system common procedures allow access to the system common area of memory defined during generation of the operating system. Under either DX10 or RX990, system common is not available to tasks that use more than two memory segments for other purposes (e.g., a task segment and two procedure segments). Also, under DX10, when system common is accessed, the heap region cannot expand. Procedure SYSCOM obtains the address and size of the system common area. Procedure RLSCOM releases the system common area from task access.

**15.3.6.1 Procedure SYSCOM.** Procedure SYSCOM allows access to the system common area by obtaining the address and size of the area defined for the operating system. To use procedure SYSCOM, the user must declare the procedure externally. The user may then call the procedure.

The declaration of procedure SYSCOM is as follows:

```
PROCEDURE SYSCOM (VAR ADDRESS, LENGTH: INTEGER); EXTERNAL;
```

The two parameters are integer variables into which the procedure places the address and the length, respectively, of the system common area. The following is an example of a call to procedure SYSCOM, specifying variables ADDR and LEN for the address and length:

```
SYSCOM (ADDR,LEN);
```

**15.3.6.2 Procedure RLSCOM.** Procedure RLSCOM releases the task from access to system common. To use procedure RLSCOM, the user must declare the procedure externally. The user may then call the procedure.

The declaration of procedure RLSCOM is as follows:

```
    PROCEDURE RLSCOM (); EXTERNAL;
```

The procedure has no parameter. The following is an example of a call to procedure RLSCOM:

```
    RLSCOM;
```

**15.3.7 SEMAPHORE PROCEDURES.** The semaphore procedures support the use of semaphores to synchronize execution of cooperating tasks. A semaphore is an integer variable, typically a field of a COMMON block shared by several tasks. Initially, the variable is reset. When a section of code in a cooperating task is critical, the task tests the semaphore to determine if the critical code may be executed. When an activity of another task has set the semaphore to prevent any task from executing critical code, the task continues to test the semaphore before executing the critical code. When the semaphore is reset, the task sets the semaphore and executes the critical code. When execution of the critical code has completed, the task resets the semaphore, allowing another task that may be awaiting the resetting of the semaphore to continue. Procedure RESETSEMAPHORE resets a specified semaphore. Procedure TESTANDSET tests a semaphore, returns the state of the semaphore, and sets the semaphore if it was reset when tested.

**15.3.7.1 Procedure RESETSEMAPHORE.** Procedure RESETSEMAPHORE resets a specified semaphore. To use procedure RESETSEMAPHORE, the user must declare the procedure externally. Then the user may call the procedure.

The declaration of procedure RESETSEMAPHORE is as follows:

```
    PROCEDURE RESETSEMAPHORE (VAR SEMAPHORE: INTEGER); EXTERNAL;
```

The parameter is the semaphore variable to be reset. An example of the use of the procedure is included in the example that shows the use of a semaphore (shown in figure 15-1 and described in paragraph 15.3.7.2).

**15.3.7.2 Procedure TESTANDSET.** Procedure TESTANDSET tests a specified semaphore and sets a Boolean variable to the opposite state. When the semaphore is reset, the procedure also sets the semaphore. To use procedure TESTANDSET, the user must declare the procedure externally. Then the user may call the procedure.

The declaration of procedure TESTANDSET is as follows:

```
    PROCEDURE TESTANDSET  (VAR SEMAPHORE: INTEGER;
                           VAR SET_OK: BOOLEAN); EXTERNAL
```

The first parameter is an integer variable containing a value that is tested by the procedure. The second parameter is a Boolean variable that is set to false when the tested variable contains a value that is interpreted as the set state. The Boolean variable is set to true when the tested variable contains a value that is interpreted as the reset state. The tested variable (semaphore) is set to the set state at the same time that the Boolean variable is set to true.

```
COMMON FLAG: INTEGER;                  (*Semaphore Variable*)
   .
   .
   .
VAR OK: BOOLEAN;                       (*True/False Variable*)
   .
   .
   .
RESETSEMAPHORE(FLAG);                  (*Initialize Semaphore*)
   .
   .
   .
REPEAT                                 (*Await Permission*)
   TESTANDSET(FLAG,OK);                (*To Execute*)
   IF NOT OK THEN DELAY(100);          (*Critical Code*)
UNTIL OK;
   .
   .                                   (*Critical Code Here*)
   .
RESETSEMAPHORE(FLAG);                  (*Reset Semaphore After Executing Critical Code*)
```

**Figure 15-1. Semaphore Example**

Figure 15-1 shows the use of a semaphore to control execution of critical code in a task. Critical code is code that cannot be executed until another cooperating task has completed execution of part (or all) of its code, and/or code that must be executed before a cooperating task can begin execution of part (or all) of its code. In the example shown, the semaphore is integer variable FLAG in the COMMON block. A similar declaration in each of the cooperating tasks makes this variable available to each task. Boolean variable OK is local to the task and contains the result of the test of the semaphore by procedure TESTANDSET.

The task initializes the semaphore by executing a RESETSEMAPHORE procedure. At this point, any cooperating task that executes procedure TESTANDSET to test semaphore FLAG would set the semaphore and set the Boolean variable parameter to true.

The REPEAT statement encloses both a call to procedure TESTANDSET and an IF statement. Boolean variable OK is set false by procedure TESTANDSET as long as semaphore FLAG remains set. The IF statement causes a call to procedure DELAY that suspends the task for 100 milliseconds when Boolean variable OK is false. The calls to procedures TESTANDSET and DELAY are repeated until semaphore FLAG is reset. When the task that has set the semaphore completes its critical code and resets the semaphore, procedure TESTANDSET sets the semaphore and sets Boolean variable OK to true. Procedure DELAY is not called, and the code following the REPEAT statement is executed.

The critical code follows the REPEAT statement and ends with a call to procedure RESETSEMAPHORE. This code can only be executed while no other task that uses semaphore FLAG is executing its critical code. While this task is executing critical code, no other task that uses the semaphore can execute critical code.

**15.3.8 PROCEDURE SVC$.** Procedure SVC$ allows the user to execute any supervisor call of the operating system under which the task is executing. To use SVC$, the user must declare the structure for the supervisor call block and must declare procedure SVC$ externally. Next, the user must call procedure NEW to obtain an area for the supervisor call block and must build the supervisor call block. Then, the user may call procedure SVC$.

The following is an example of the declarations required for a supervisor call block and for a pointer to the supervisor call block:

```
TYPE POINT = @DANDT;
     SCB = RECORD
           CODE: INTEGER;
           TIME: POINT
         END;
     PT = @SCB;
     DANDT = ARRAY [1..5] OF INTEGER;
VAR BLOCK : PT;
```

The declaration for procedure SVC$ is as follows:

```
PROCEDURE SVC$(P:PT); EXTERNAL;
```

The supervisor call block in the example is the block for a date and time supervisor call. This supervisor call requires a four-byte block with the code in the first byte and zero in the second byte; the address of a five-word array for the date and time is in the third and fourth bytes. By declaring a record consisting of an integer and a pointer, the code and the zero can be initialized by an assignment. The array is obtained by a call to procedure NEW, and its address is placed in the record. The record SCB is also obtained by a call to procedure NEW, and its address is assigned to pointer PT. The following example shows the calls to procedure NEW and the call to procedure SVC$:

```
     NEW (BLOCK);                (*OBTAIN SUPERVISOR CALL BLOCK*)
     WITH BLOCK@ DO BEGIN
     NEW (TIME);                 (*OBTAIN ARRAY FOR DATE AND TIME*)
     CODE: =#0300;               (*ASSIGN CODE AND ZERO*)
     SVC$(BLOCK);                (*GET DATE AND TIME*)
END;
```

The result of the example call is that the system has placed the year in element 1 of array DANDT, the day in element 2, the hour in element 3, the minute in element 4, and the second in element 5. All values are binary; the year is the binary equivalent of the two least significant digits of the year.

## 15.4 DUMMY MAIN ROUTINE

Normally, a TIP task consists of a main routine, or program, and one or more routines that the main routine calls, directly or indirectly. A TIP task can be compiled with a dummy main routine and begin execution in a procedure. There are three instances when this should be done:

- When a task linked for minimal runtime code does not require predeclared files INPUT and OUTPUT
- When multiple tasks are linked with a TX990 system
- When a task executes stand-alone

When a task does not require the predeclared files, the runtime code can be reduced below that resulting from linking with the minimal runtime library by omitting the code for these files. When a task is linked with a dummy main routine, the runtime code for these files is omitted.

When multiple tasks are linked with TX990, each task must have a unique entry point. The entry point for a task beginning execution in the main routine is PSCL$$, which allows one of a set of tasks to have a main routine and use predeclared files INPUT and/or OUTPUT. The entry point for each of the other tasks is supplied by the TASK macro instruction for the task as the procedure operand, as in paragraph 14.3.7.

A program is compiled with a dummy main routine by declaring no variables in the declaration section of the main program and by placing a (*$ NO OBJECT *) option comment in the statements section of the main program. The following are the statements of a task with a dummy main routine:

```
PROGRAM <name>;                 (*Arbitrary name; not used *)
CONST ...                       (*Global constant declarations *)
TYPE ...                        (*Global type declarations *)
COMMON ...                      (*Declare all COMMON blocks here *)
PROCEDURE <name>;               (*Declaration of initial procedure - see text *)
    VAR ...
    PROCEDURE ...               (*Declaration of additional procedure *)
      .
      .
      .
    BEGIN
      .
      .
      .
    END;
  .
  .
  .
BEGIN                           (*Statements of main program *)
  (*$ NO OBJECT *)              (*The main program is a dummy *)
END.                            (*End of program *)
```

The declaration of the initial procedure for a task that executes under DX10 or RX990 or stand-alone should use the name PSCL$$; this name identifies the procedure as the one in which execution begins.

The case of multiple tasks to be linked with TX990 differs in several respects. Typically, there are several procedures, each of which is the entry procedure of a task, and one or more procedures shared by two or more of these tasks. The following is an example:

```
PROGRAM name ;                  (*Arbitrary name; not used*)
CONST ...                       (*Global constant declarations*)
TYPE ...                        (*Global type declarations*)
COMMON ...                      (*Declare all COMMON blocks here*)
PROCEDURE ...                   (*Procedure definition*)
    VAR ...
    BEGIN
```

```
        .
        .
        .
      END;
  PROCEDURE ...                        (*Procedure definition*)
        .
        .
        .
  PROCEDURE ...                        (*Procedure definition*)
        .
        .
        .
  BEGIN                                (*Body of main program*)
      (*$ NO OBJECT*)                  (*The main program is a dummy*)
  END.                                 (*End of program*)
```

The declaration of the initial procedure for each of the multiple tasks uses the name that is used as the operand of the TASK macro instruction (paragraph 14.3.7) for the task. The declaration of each procedure that is shared uses the name by which it is called.

Because the TASK macro instruction also specifies the size of stack and heap required, the parameters in the OCP command or in the routine that bids the task may be used by the initial procedure. The procedure declaration must declare the parameters as value parameters. The types of the parameters may be any type, but the maximum length of the two parameters is four bytes.

## 15.5 PROGRAM CONSIDERATIONS FOR LUNO I/O

Access to I/O devices and/or files by Logical Unit Number (LUNO) is available optionally for tasks executing under DX10 and is the only means of access for tasks executing under TX990 or RX990. This method of access requires that a LUNO be defined for each I/O operation, either explicitly or by default.

The SETLUNO procedure (paragraph 15.3.1.4) is the means of explicitly defining a LUNO for an I/O operation. When no LUNO is explicitly defined, the ASCII code for the first letter of the file name is the LUNO, by default. This results in LUNOs in the range of $41_{16}$ through $5A_{16}$, and $24_{16}$ (for file names that start with $).

The LUNOs for the predeclared files are:

- $3C_{16}$ for SYSMSG
- $3D_{16}$ for INPUT
- $3E_{16}$ for OUTPUT

Under TX990 and RX990, LUNOs are global and a LUNO must be not only unique to the task but also unique among all files simultaneously open in the system. To change the LUNO for a file declared by the user, call procedure SETLUNO to explicitly define a LUNO that is unique to the task.

The LUNOs for predeclared files are defined by a library module. A module may be assembled defining other LUNOs for these files. The following is the source code for this module, LU$MSG:

```
          IDT   'LU$MSG'
          DEF   LU$MSG,LU$IN,LU$OUT
LU$MSG    EQU   >3C              MESSAGE FILE
LU$IN     EQU   >3D              "INPUT"
LU$OUT    EQU   >3E              "OUTPUT"
          END
```

To change the LUNOs for these files, change the operands for the three EQU directives to the desired LUNOs. Then, assemble the source code and link in the new module with an INCLUDE command in the link edit control file.

Another way to change the LUNO for predeclared file OUTPUT is to use procedure SETLUNO along with the CLOSE and REWRITE procedures. This must be done in all tasks that attempt to use the LUNO concurrently. For example, if more than one task terminate abnormally and attempt to write abnormal termination dumps at the same time, each must have a different LUNO defined for file OUTPUT. Use the following procedure calls at the beginning of each task:

```
CLOSE (OUTPUT);              (*Close OUTPUT file (assigned to DUMY)*)
SETLUNO (OUTPUT, XX);        (*XX is desired LUNO*)
REWRITE (OUTPUT);
```

When the tasks are executed, LUNO $3E_{16}$ must be assigned to DUMY. Each task closes DUMY (which is automatically opened by the runtime code), redefines the LUNO for the OUTPUT file, and opens the file or device assigned to that LUNO.

A program may call both SETLUNO and SETNAME procedures. When the runtime code for SCI I/O is linked with the task, procedure SETNAME executes and procedure SETLUNO is ignored. When the runtime code for LUNO I/O is linked with the task, procedure SETLUNO is executed and procedure SETNAME is ignored.

## 15.6 PROGRAM CONSIDERATIONS FOR MULTIPLE TASKS

There are several programming considerations to be observed when writing the source code for multiple tasks. Some apply to multiple tasks that are to execute under DX10 or RX990; others apply to multiple tasks that are linked with TX990.

Assign user task numbers in the range of $60_{16}$ through $EF_{16}$ to prevent assigning a task number that has previously been assigned to a system task.

Blocks of COMMON data are accessible to all tasks (global) when those tasks are linked with TX990. Multiple tasks that are to execute under DX10 or RX990 may be linked so that COMMON blocks are global, or they may be linked differently, making all or part of the COMMON data accessible only to one task (local). When procedure $BLOCKS (described in paragraph 14.3.15) is provided, it is linked with a procedure segment and the COMMON blocks are global to all tasks that share that procedure segment. Otherwise, COMMON blocks are linked in the task segment and are local to the task.

Variables of pointer type should be local to one task. These variables access heap space. Under DX10 and RX990, the access to heap is by means of mapping, which differs from task to task. As a result, pointers of one task are invalid in another. Under TX990, heap space is deallocated when a task calls the DISPOSE procedure or when it terminates. Use of a pointer of one task by another is only valid until the original task calls DISPOSE or terminates.

When multiple tasks share procedure segments that include user routines, the user routines must be written with care to avoid invalid data references. A routine that is declared as EXTERNAL must not reference global variables. (In this context, global variables are variables that are declared neither in the routine nor in COMMON.) The reason for this limitation is that these global variables are accessed through the stack frame; the stack frame of the routine when called in the program in which it is declared and the stack frame when called in a program in which the routine is externally declared are different. The requirement can be met by using the GLOBALS option when compiling the routine (paragraph 9.6.3).

## 15.7 PROGRAM CONSIDERATIONS FOR STAND-ALONE TASKS

A stand-alone task must consist of a dummy main routine with the required routines as described in paragraph 15.4. No variables may be declared for the main routine, and the statement section includes a NO OBJECT option comment. Execution begins in a procedure named PSCL$$.

Because there is no operating system, none of the TIP I/O facilities may be used. I/O with CRU devices may be performed using the direct CRU I/O routines described in paragraph 15.2. TILINE addresses may be accessed using a type transfer to assign an integer constant to a pointer variable, used as the TILINE address. The following is an example of code to accomplish this:

```
TYPE TREC = RECORD ... END;  (*TILINE control block*)
VAR  TP: @TREC;              (*TILINE address*)
        .
        .
        .
TP::INTEGER := #F800;        (*Set TILINE address*)
WITH TP@ DO BEGIN
        .
        .                    (*TILINE operations*)
        .
             END
```

With the record defined according to the requirements of the TILINE device controller and the pointer set to the TILINE address, the TILINE operations consist of assigning values to the elements of the record and of performing other appropriate operations as if the record were in memory.

Any interrupt or XOP transfer vectors required by the task must be initialized by the task. The technique used to assign a value to a pointer to be used as a TILINE address may also be used to assign an absolute memory address at which to store a transfer vector. The TYPE declaration declares a record that represents a transfer vector rather than a TILINE control block. The type transfer assignment sets the absolute address of the transfer vector.

None of the following library routines may be called in a stand-alone task:

ACTIVATE
BID
DATE
DELAY
GETMSG

IDATE
ITIME
MESSAGE
OVLY$
PRGMSG
PUTMSG
READ
RESET
REWRITE
RLSCOM
STATIONID
SUSPEND
SVC$
SYSCOM
TASKID
TIME
WRITE
WRITELN

When a stand-alone task calls SVC$, the second byte of the supervisor call block is set to $FF_{16}$. When a stand-alone task calls MESSAGE, the label MSG$ is listed as an unresolved reference in the link map written for the linking operation. A call to any of the other library routines produces unpredictable results.

A library routine is provided to allow a stand-alone task to display a value on the programmer panel indicators. To use the routine, the task must declare the routine externally, as follows:

```
PROCEDURE DSPLY$ (VALUE: INTEGER); EXTERNAL;
```

The parameter is a value that is displayed in the indicators on the programmer panel as a 16-bit binary number.

## 15.8 PROGRAM CONSIDERATIONS FOR MINIMAL RUNTIME CODE

Programs may be linked to require a minimal amount of runtime code, that required for the exact capabilities of the program. Paragraph 15.4 describes the coding of a program with a dummy main routine, which eliminates runtime code for predeclared files INPUT and OUTPUT. A further reduction of memory requirements for a task results when debugging options are omitted during compilation of the final version. The following option comment should be included:

```
(*$ NO TRACEBACK, NO ASSERTS*)
```

Also, none of the following options should be specified:

CKINDEX
CKOVER
CKPREC
CKPTR
CKSET
CKSUB
CKTAG
PROBER
PROBES

# APPENDIX A
# TIP STANDARD ROUTINES

# APPENDIX A

# TIP STANDARD ROUTINES

## A.1 GENERAL

The TIP software includes a set of standard routines, which are listed in this appendix. Some of these are functions, and others are procedures; all may be called in a user program.

## A.2 TIP STANDARD FUNCTIONS

The TIP standard functions are listed in this paragraph with a brief description of each function. These functions are called with function calls as described in paragraph 8.4. The functions, listed in alphabetical order, are as follows:

- ABS(X)

  X is an expression of type INTEGER, LONGINT, REAL, FIXED, or DECIMAL. The result of ABS(X) is the absolute value of X and is of the same type as the input expression argument.

- ARCTAN(X)

  X is an expression of type INTEGER, LONGINT, REAL, DECIMAL, or FIXED. The result, of type REAL, is the arctangent of X.

- CHR(X)

  X is an expression of type INTEGER or LONGINT. The result of CHR(X), of type CHAR, is the character with the ordinal value of X modulo the ordinal of the last character.

- COLUMN(X)

  X is a variable of type TEXT. The result of COLUMN(X), of type INTEGER, is the column index (based at 1) at which the next character on the textfile X will be read or written.

- COS(X)

  X is an expression of type INTEGER, LONGINT, REAL, DECIMAL, or FIXED. The result, of type REAL, is the cosine of X.

- DEC(P, Q, X)

  X is an expression of type INTEGER, LONGINT, REAL, FIXED, or DECIMAL. The result is the decimal value corresponding to the value of X, but with precision P, a nonnegative INTEGER constant, and scale factor Q, an INTEGER constant.

- EOF(F)

  EOF —> EOF(INPUT)

  F is a variable of type FILE or TEXT. The result, of type BOOLEAN, is true if the sequential file F is not open for input or is in the end-of-file state or end-of-medium state. For random file F, EOF is true when the last read of file F attempted to access a nonexistent record.

- EOLN(H)

  EOLN —> EOLN(INPUT)

  H is a variable of type TEXT. The result, of type BOOLEAN, is TRUE if the last character of the current line in the file H has been read.

- EXP(X)

  X is an expression of type INTEGER, LONGINT, REAL, DECIMAL, or FIXED. The result, of type REAL, is the exponential of X.

- FIX(P, Q, X)

  X is an expression of type INTEGER, LONGINT, REAL, FIXED, or DECIMAL. The result is the fixed point value corresponding to the value of X, but with precision P, a nonnegative INTEGER constant, and scale factor Q, an INTEGER constant.

- FLOAT(X, P)

  X is an expression of type INTEGER, LONGINT, REAL, FIXED, or DECIMAL. P is a nonnegative INTEGER constant. The result is the REAL value with precision P corresponding to the input argument X.

- LINT(X)

  X is an expression of type INTEGER or LONGINT. The result, of type LONGINT, is the converted value of X.

- LN(X)

  X is an expression of type INTEGER, LONGINT, REAL, DECIMAL, or FIXED. The result, of type REAL, is the natural logarithm of X if X is greater than zero, otherwise there is an error.

- LOCATION(X)

  X is a variable or a procedure or function identifier. The result, of type INTEGER, is the address of the variable X, or the entry point of the procedure or function X. If X is a variable, it may not be a component of a packed structure or a file variable.

- LROUND(X)

  X is an expression of type REAL, DECIMAL, or FIXED. The result, of type LONGINT, is the rounded long integer.

  = LTRUNC(X + 0.5), X>=0
  = LTRUNC(X - 0.5), X<0

- LTRUNC(X)

  X is an expression of type REAL, DECIMAL, or FIXED. The result, of type LONGINT, is the whole part, i.e. the fractional part is discarded.

- ODD(X)

  X is an expression of type INTEGER or LONGINT. The result, of type BOOLEAN, is TRUE if and only if X is odd.

- ORD(X)

  X is an expression of type BOOLEAN, CHAR, or scalar type. The result, of type INTEGER, is the ordinal of the character X, or the ordinal of the scalar identifier X.

- PRED(X)

  X is an expression of some enumeration type. The result type is the same type as X and represents the predecessor of X. An error occurs if the subrange check option is set and the predecessor value in not an element of the enumeration. Otherwise, if X is not of type INTEGER or LONGINT and is the first element of the enumeration, PRED(X) is the undefined element of the enumeration type whose ordinal is ORD(X) - 1. If X is of type INTEGER or LONGINT, PRED(X) is X - 1.

- ROUND(X)

  X is an esxpression of type REAL, DECIMAL, or FIXED. The result, of type INTEGER, is the rounded integer.

  = TRUNC(X + 0.5), X>=0
  = TRUNC(X - 0.5), X<0

- SIN(X)

  X is an expression of type INTEGER, LONGINT, REAL, DECIMAL, or FIXED. The result, of type REAL, is the sine of X.

- SIZE(T)

  SIZE(T,T1, . . . , Tn)

  SIZE(V)

  SIZE(V,T1, . . . , Tn)

  T is a type and V is a variable. T may not be REAL(P), DECIMAL(P,Q), or FIXED(P,Q). To obtain the size of these types, a type identifier must be passed as the argument. The result, of type INTEGER, is the number of addressable storage units required to represent the type T or the variable V. If T is a record type, tag field values T1, . . . , Tn may be specified. n must be less than or equal to the number of variants and T1, . . . , Tn must represent a complete initial sequence of tag fields and be compile time constants.

- SQR(X)

  X is an expression of type INTEGER, LONGINT, REAL, DECIMAL, or FIXED. The result, of the same type as X, is the value of X squared.

- SQRT(X)

  X is an expression of type INTEGER, LONGINT, REAL, DECIMAL, or FIXED. The result, of type REAL, is the square root of X.

- STATUS(F)

  F is a variable of type FILE or TEXT. The result, of type INTEGER, is the status of the last I/O operation on the file F.

- SUCC(X)

  X is an expression of some enumeration type. The result, of the same type as X, represents the successor of X. An error occurs if the subrange check option is set and the successor value is not an element of the enumeration. Otherwise, if X is not of type INTEGER or LONGINT and is the last element of the enumeration, SUCC(X) is the undefined element of the enumeration type whose ordinal is ORD(X) + 1. If X is of type INTEGER or LONGINT, SUCC(X) is X + 1.

- TRUNC(X)

  X is an expression of type INTEGER, LONGINT, REAL, DECIMAL, or FIXED. The result, of type INTEGER, is the whole part, i.e. the fractional part is discarded.

- UB(A)

  UB(A,D)

  UB(S)

  A is a variable of type ARRAY or a type identifier for an array. If D is present, it is an INTEGER constant greater than zero. For arrays, the result is the upper bound of the Dth dimension of the array A. When D is absent, the result is the upper bound of the index type of A. The type of the result is the same as the Dth index type. Dimensions are numbered left to right starting with 1.

  S is a variable of type SET or a type identifier denoting a set or a set expression. For sets, the result is the upper bound of the base type of the set S. The type of the result is the same as the base type of S.

**A.3 TIP STANDARD PROCEDURES**

The TIP standard procedures are listed in this paragraph with a brief description of each procedure. These procedures are called with procedure statements as described in paragraph 8.4. The procedures, listed in alphabetical order, are as follows:

- CLOSE(F)

  F is a variable of type FILE or TEXT. This procedure places the file F in a closed state. If F is a sequential or text file that is open for output, write an end-of-file before closing. The textfile OUTPUT must not be closed.

- DATE(V)

  V is a variable of type PACKED ARRAY [1. .8] OF CHAR. This procedure assigns the current date to the variable V.

- DECODE(S, N, STAT, Q)

  S is a variable of string type. N is a constant or variable of type INTEGER. STAT is a variable of type INTEGER. Q is a read parameter. This procedure converts the characters starting at the Nth component of S to the internal representation of the read variable V and places the value in V. The status of the operation is returned in STAT.

- DISPOSE(P)

  DISPOSE(P, T1, . . . , Tn)

  P is a variable of pointer type. The T's, if present, are tag field values of the record variable P to be deallocated. n must be less than or equal to the total number of nested variants, and T1, . . . , Tn must represent a complete initial sequence of tag fields and be compile time constants. P may be a component of a packed structure.

- ENCODE(S, N, STAT, P)

  S is a variable of type string. N is a constant or variable of type INTEGER. STAT is a variable of type INTEGER. P is a write parameter. This procedure places the character repreesentation of the value of the write expression E into S starting at the Nth component. The status of the operation (paragraph 14.3) is returned in STAT.

- EXTEND(F)

  This procedure opens a sequential file F for output and positions it so the first component written will be the successor of the last component of the last logical file of the file. It opens a random file F for both input and output.

- HALT

  This procedure causes the execution of a program to be terminated and a memory dump (figure 14-2) to be written to the OUTPUT file. The HALT procedure is intended to be used for an abnormal termination of a program.

- IOTERM(F, OVAL, NVAL)

  F is a variable of type FILE or TEXT. OVAL is a variable of type BOOLEAN. NVAL is an expression of type BOOLEAN. The value of the I/O error flag for the file F is returned in OVAL and the flag's value is set to NVAL.

- MESSAGE(X)

  X is an expression of string type. This procedure places the string X into the system message file.

- NEW(P)

  NEW(P, T1, . . . , Tn)

  P is a variable of pointer type. The T's, if present, are tag field values of the record variable P to be allocated. n must be less than or equal to the total number of nested variants, and T1, . . . , Tn must represent a complete initial sequence of tag fields and be compile time constants. P may be a component of a packed structure.

- PACK(A, I, Z)

  A is a variable of type ARRAY[M. .N] OF T1. I is an expression, the type of which must be compatible with the index type of A. Z is a variable of type PACKED ARRAY[U. .V] OF T2. T1 and T2 are compatible types and (N - M) >= (V - U). This procedure packs the components of A into the array Z, starting at the Ith component of A.

- PAGE(H)

  PAGE —> PAGE(OUTPUT)

  H is a variable of type TEXT. This procedure causes a skip to the top of a new page when the textfile H is printed.

- READ

  This procedure is used to read sequential files, text files, and random files. The first argument indicates the file to be read from. If the first argument is not a file variable, "INPUT" is used as the default. For sequential files, the remaining argument(s) must specify variables which are of a type compatible with the particular file component type. The argument(s) for a text file read, so called read parameters, may be of different types. Details describing the forms these parameters may take are found in paragraph 6.5. If the file is a random file, the second argument specified must be of type INTEGER (LONGINT) and is an index into the random file.

- READLN

  This procedure may be applied only to text files. It is used to read and subsequently skip to the beginning of the next line. The values read may stretch over several lines. The arguments for READLN are the same as those for READ applied to text files. (paragraph 6.5.2)

- RESET(F)

  F is a variable of FILE or TEXT type. This procedure opens the file F for input and positions it to read its first component. For a sequential or text file, if the file is empty, EOF(F) becomes TRUE, otherwise it becomes FALSE.

- REWRITE(F)

  F is a variable of type FILE or TEXT. This procedure makes the file F empty and opens it for output. For a sequential or text file, EOF(F) becomes TRUE.

- SETLUNO(F,LUNO)

  F is a variable of type FILE or TEXT. LUNO is type INTEGER and specifes the logical unit number for file F.

- SETMEMBER(F, LIBNAME, MEMBER)

  F is a variable of FILE or TEXT type. LIBNAME and MEMBER are both expressions of type PACKED ARRAY [1. .8] OF CHAR. LIBNAME may not be the text file OUTPUT. This procedure associates the file F with the member MEMBER of library LIBNAME. LIBNAME is the synonym of a directory, and MEMBER is the file name within the directory.

- SETNAME(F, NAME)

  F is a variable of FILE or TEXT type. NAME is an expression of type PACKED ARRAY[1. .8] OF CHAR. NAME may not be the textfile OUTPUT. This procedure associates the file F with the external file specified by NAME.

- SKIPFILES(G, NFILE)

  G is a variable of type FILE or TEXT. NFILE is an expression of type INTEGER. SKIPFILES skips the number, NFILE, of file marks on the file G which is open for input. If NFILE is negative, the skip is in the "backward" direction; if NFILE is zero, the file is positioned to the beginning of the current logical file; if NFILE is positive, the skip is in the forward direction. The file is positioned at the start of a logical file. An attempt to position to a nonexistent file will cause an error. If EOF is TRUE following a skip, then end-of-medium has been reached.

- TIME(V)

  V is a variable of type PACKED ARRAY [1. .8] OF CHAR. This procedure assigns the current time of day to the variable V.

- UNPACK(Z, A, I)

  Z is a variable of type PACKED ARRAY [U. .V] OF T1. A is a variable of type ARRAY [M. .N] OF T2. I is an expression, the type of which must be compatible with the index type of A. T1 and T2 are compatible types and (N - M) >= (V -U). This procedure unpacks the components of Z into the array A starting at the Ith component of A.

- WRITE

  This procedure is used to write to sequential files, text files, and random files. The first argument indicates the file to be written to. If the first argument is not a file variable, "OUTPUT" is used as the default. For sequential files, the remaining argument(s) must specify expressions which are of a type compatible with the particular file component type. The argument(s) for a text file write, so called write parameters, may be of different types. If the file is a random file, the second argument specified must be of type INTEGER (LONGINT) and is an index into the random file. Refer to paragraph 6.5 for the forms of the parameters.

- WRITEEOF(G)

  G is a variable of type FILE or TEXT. This procedure writes an end-of-file mark on the file G which is open for writing. G cannot be a random file.

- WRITELN

  This procedure may be applied only to text files. It is used to terminate the current line of the text file. If the current line is empty, a blank line is written. The arguments for WRITELN are the same as those for WRITE applied to text files. (Paragraph 6.5.2)

# APPENDIX B
# EXAMPLE PROGRAMS

# APPENDIX B

# EXAMPLE PROGRAMS

## B.1 GENERAL

This appendix includes four examples of TIP programs. The examples are chosen to illustrate applications of the capabilities of TIP, rather than to show implementations of algorithms. The example programs are ROOT, QUAD, RANDOM, and GENERATE.

## B.2 ROOT

Program ROOT illustrates the use of function parameters in a routine, and computes the roots of an equation in the form F(X)=0 using Newton's method. Figure B-1 shows the listing of the program.

The declarations of the program declare two functions and a procedure. The main program calls the procedure, specifying the functions as parameters for the call. The procedure calls the function F to compute that function of the current value of X, and functions F and D in computing a new value of X. The procedure repeats these computations until the new value of F(X) has an absolute value equal to or greater than the absolute value of function F for the previous value of X. By using function parameters for procedure NEWTON it may be used for other functions when these functions and their derivatives are available to be substituted in the call.

## B.2 QUAD

Program QUAD illustrates a program that computes the roots of quadratic equations without declaring any routines. Program QUAD also illustrates the use of formatted output. Figure B-2 shows the listing of the program.

QUAD displays an appropriate heading, then reads values of a, b, and c (coefficients of equations of the form $ax^2 + bx + c$) and solves the equations. QUAD displays the values of a, b, and c and computes the discriminant ($b^2 - 4ac$). An IF statement computes and displays the real roots when the discriminant is zero or positive, and an ELSE clause computes and displays complex roots when the discriminant is negative. The program continues reading values and solving equations until the file is exhausted (end-of-file is true).

## B.3 RANDOM

Program RANDOM illustrates the use of COMMON and ACCESS declarations to make common variable RANSEED available to both procedure RANSET and procedure IRANDOM. The program also illustrates a program consisting only of procedures, with a dummy main program. The first procedure, RANSET, sets common variable RANSEED to the value of the parameter. The second procedure, IRANDOM, computes an integer random number and stores this number as a new value of RANSEED. Figure B-3 shows the listing of program RANDOM.

There is no way to compile a TIP routine without also compiling a main program, but the main program may be a dummy module as in this example. The NO OBJECT option inhibits writing of the object module. The procedures RANSET and IRANDOM may be used in other programs that declare them externally.

```
Program ROOT;
(*Author:          *)
(*Demonstration of Newton's method for solution of an equation. *)
(*The equation is in the form: F(X)=0.                          *)
(*A function representing the derivative of F is required for   *)
(*Newton's method.                                               *)

Var  X: Real;

Function F(X: Real): Real;
   Begin F := Sqr(X)-9  End;

Function D(X: Real): Real:
   Begin  D := X+X  End;

Procedure NEWTON(Function F(Real):Real; Function D(Real):Real;
                 Var X:Real);
   (*Finds the root of equation F(X)=0, where F'(X)=D(X). *)
   Var
      ERROR : Real;
   Begin(*NEWTON*)
      Repeat
         Writeln(X:18:7, F(X):18:7);
         ERROR := ABS(F(X));
         X := X-F(X)/D(X);
      Until ERROR <= ABS(F(X))
   End(*NEWTON*);

Begin(*ROOT*)
   X := 10.0;    (*Initial guess*)
   NEWTON(F, D, X);
   Writeln(X);
End(*ROOT*).
```

**Figure B-1. Listing of Example Program ROOTS**

## B.4 GENERATE

Program GENERATE illustrates the inclusion of externally compiled routines in a TIP program. It also illustrates the use of a record consisting of a variant part only. Figure B-4 shows the listing of the program.

Program GENERATE generates random real numbers from integer real numbers generated using the procedures of the preceding example. Procedures RANSET and IRANDOM are declared externally in program GENERATE to enable use of these routines in program GENERATE.

```
Program quad;
(* author:            *)
(* program to solve quadratic equations  *)
Var
   a,b,c: real,   (*coefficients of equation*)
   realpart, imagpart:  real;  (*real and imaginary part of complex result*)
   disc: real;  (*discriminant*)
Begin
   (*write heading*)
   writeln('COEFFICIENTS':22, 'ROOT 1':26,  'ROOT 2':22);
   writeln('A':8, 'B':10, 'C':10, 'REAL':15,  'IMAG':10,
           'REAL':12, 'IMAG':10);
   reset(input);
   while not eof do
   begin
      readln(a, b, c);
      write(' ', a:11:4, b:10:4, c:10:4);
      disc : = sqr(b) - 4*a*c;
      if disc  >=0
         then   (*roots are real*)
            writeln(  (-b+sqrt(disc))/(2*a)):13:4,
                      (-b-sqrt(disc))/(2*a)):22:4)
         else begin (*roots are complex*)
            realpart := b/(2*a);
            imagpart := sqrt(-disc)/(2*a);
            writeln(realpart:13:4, imagpart:10:4,
                    realpart:12:4, -imagpart:10:4);
           end (*else*)
   end (*while not eof*)
End (*quad*)
```

**Figure B-2. Listing of Example Program QUAD**

The variables declared for procedure RANDM include a packed record DOUBLEWORD. This record consists of a variant part that may be accessed as any of six bytes; as an array of 10 hexadecimal digits; as either of two integers; or as a real number. The tag field that selects one of these variants is neither declared nor used in the program; however, the record is accessed in each of these ways. For example, the first and second lines of the compound statement of the WITH statement contain assignment statements that assign values to the record as integer type values. The third and fourth lines are assignment statements that assign values to the record as bytes. Next is a WHILE statement that accesses the record as hexadecimal digits. Finally, the record is moved to the parameter X as a real number.

The main program GENERATE calls procedure RANSET to set the random number generator seed to zero, and calls procedure RANDM to generate a real random number. Procedure RANDM calls procedure IRANDOM to obtain random integers. RANDM moves the eight most significant bits of the resulting value to the least significant end of the number and places the exponent $40_{16}$ in the most significant byte. Next, RANDM normalizes the number as a real number, moving the leading zeros, if any, and correcting the exponent appropriately. The result is a real random number, assigned to global variable X.

```
PROGRAM RANDM;
(*AUTHOR:       *)
(*THIS PROGRAM DECLARES TWO PROCEDURES THAT MAY BE USED TO GENERATE *)
(*INTEGER RANDOM NUMBERS  *)
CONST
   RANMULTIPLIER = 7829; RANADDEND = 13849;
COMMON
   RANSEED : INTEGER;

PROCEDURE RANSET(I: INTEGER);
   ACCESS RANSEED;
   BEGIN
      RANSEED := I
   END;

PROCEDURE IRANDOM(VAR I: INTEGER);
   ACCESS RANSEED;
   BEGIN
      I := RANMULTIPLIER*RANSEED+RANADDEND;
      RANSEED := I
   END;

BEGIN (*RANDM*)
(*$NO OBJECT*)
END (*RANDM*)
```

**Figure B-3. Listing of Example Program RANDOM**

```
PROGRAM GENERATE;
(*AUTHOR:          *)
(*THIS PROGRAM MAKES USE OF THE TWO PROCEDURES FOR GENERATING INTEGER*)
(*RANDOM NUMBERS TO GENERATE A REAL RANDOM NUMBER IN THE RANGE 0 to 1*)
VAR X : REAL;

PROCEDURE RANSET(I: INTEGER); EXTERNAL;

PROCEDURE IRANDOM(VAR I: INTEGER); EXTERNAL;

PROCEDURE RANDM(VAR X: REAL);
   VAR
      I: INTEGER;
      DOUBLEWORD: PACKED RECORD
         CASE INTEGER OF
            0: (EXPONENT,BYTE1,BYTE2,BYTE3,BYTE4, BYTE5: 0. .#FF);
            1: (HEXDIGIT: PACKED ARRAY[0. .9] OF 0 . . #F),
            2: (MSW, LSW: INTEGER);
            3: (R: REAL)
         END (* DOUBLEWORD *);
   BEGIN (* RANDM *)
   WITH DOUBLEWORD DO
   BEGIN
      IRANDOM(I); MSW := I;
      IRANDOM(I); LSW := I;
      BYTE4 := EXPONENT;  (* SAVE THIS BYTE FOR LATER USE *)
      EXPONENT := #40;    (*FUTURE VALUE OF EXPONENT*)
      WHILE HEXDIGIT[2] = 0 DO
      BEGIN (*NORMALIZATION*)
         FOR I := 2 TO 8 DO HEXDIGIT[I] := HEXDIGIT[I+1];
         HEXDIGIT[9] := 0;
         EXPONENT := EXPONENT - 1
      END;(*NORMALIZATION*)
      X := R;          (* X IS THE RESULT *)
   END (*WITH*)
   END (*RANDM*);

BEGIN (*GENERATE*)
   RANSET(0);
   RANDM(X);
END (*GENERATE*).
```

**Figure B-4. Listing of Example Program GENERATE**

# APPENDIX C
# SYSTEM DEPENDENT INFORMATION FOR DX10 RELEASE 3

# APPENDIX C

# SYSTEM DEPENDENT INFORMATION FOR DX10 RELEASE 3

## C.1 INTRODUCTION

This appendix describes the implementation of the types of files available to TIP tasks under DX10. The appendix also describes device I/O provided by DX10. Finally, this appendix shows the runtime code sizes required for TIP features.

## C.2 FILES FOR TIP PROGRAMS

At the TIP source level there are three basic file types: sequential files, textfiles, and random files. The characteristics of these files as provided by DX10 are described in subsequent paragraphs.

The logical file name used in the file access procedures of a TIP program is implemented as a synonym to DX10 file management. The user assigns a value to the synonym which is the pathname of the file. When the user does not assign a value to a synonym, the runtime system assigns a pathname derived from the logical file name. The pathname begins with a period, meaning that the file is cataloged in the volume directory of the system disk. It consists of the characters of the logical file name (when the logical file name consists of fewer than seven characters) or the first six characters of the logical file name, followed by the numeric characters of the user's station number. For example, if neither OUTPUT nor INPUT were assigned values and the station number were ST03, the default pathnames would be:

```
.OUTPUT03
.INPUT03
```

The message file SYSMSG used by procedure MESSAGE is an exception. When no file is assigned or when the file cannot be opened for any reason, the diagnostic message is written to a file with the access name consisting of .SYSMSG followed by the digits of the station number. If the station number were ST03, the access name would be:

```
.SYSMSG03
```

When a TIP task is written and linked for accessing I/O using LUNOs, a LUNO must be associated with every file. When the user does not specify a LUNO, the ASCII code for the first letter of the file name is the LUNO, by default. This results in LUNOs in the range of $41_{16}$ through $5A_{16}$, and $24_{16}$ (for file names that start with $). Predeclared files have predefined LUNOs as follows:

- $3C_{16}$ for SYSMSG
- $3D_{16}$ for INPUT
- $3E_{16}$ for OUTPUT

**C.2.1 SEQUENTIAL FILES.** When a file is declared as a sequential file in a TIP program, the file has the following characteristics:

- Sequential file with logical record length equal to or greater than the element length.
- When opened for input (RESET procedure) has read-only access.
- When opened for output (REWRITE or EXTEND procedure) has exclusive write access.

A file element is the type declared for the file. Typically it would be a record but may be any data type except FILE, POINTER, or ARRAY or RECORD containing a FILE or POINTER type. When the file is created automatically, the logical record length is equal to the element length. When the user creates the file, the logical record length may not be less than the element length.

When the file is created automatically, it is not blank suppressed. Input and output operations are blank adjusted and transfer as many bytes as the element size.

**C.2.2 TEXTFILES.** When a file is declared as a textfile in a TIP program, the file has the following characteristics:

- Sequential file.
- When opened for input (RESET procedure) has read-only access.
- When opened for output (REWRITE or EXTEND procedure) has exclusive write access.

When a textfile is created automatically the logical record length is 80 characters and the file is blank suppressed. When the user creates the file, the logical record length specified becomes the maximum line length and determines the point at which end-of-line occurs. Input and output operations transfer variable length records.

**C.2.3 RANDOM FILES.** When a file is declared as a random file in a TIP program, the file has the following characteristics:

- Relative record file with logical record length equal to or greater than the element length.
- When opened by a RESET procedure, has read-only access.
- When opened by a REWRITE procedure, has exclusive all access.
- When opened by an EXTEND procedure, has shared access.

A file element is the type declared for the file. Typically it would be a record but may be any data type except FILE, POINTER, or ARRAY, or RECORD containing a FILE or POINTER type. When a random file is created automatically, the logical record length is equal to the element length. When the user creates the file, the logical record length may not be less than the element length.

When the file is created automatically it is not blank suppressed. Input and output operations transfer as many bytes as the file element size.

The standard function EOF may be called for random files and permits the user to detect the end-of-file record maintained by DX10. This end-of-file record indicates the highest numbered record written to the file and is detected when the end-of-file record is read or when an attempt to read a higher-numbered record is made. Function EOF must follow a READ attempt. The function result does not indicate whether any records have actually been written.

## C.3 DEVICE I/O

When the access name supplied for an input or output operation is a device name, the operation transfers data to or from that device. The logical record length is 80 characters for the following devices:

- ASR teleprinter
- ASR keyboard
- Cassette
- Card reader
- Video display terminal
- Dummy device

The logical record length for a line printer is two less than the logical record length returned for the printer by DX10.

A line feed and carriage return sequence is sent before each line during a write operation to the ASR teleprinter, VDT, or line printer, and during a read operation from the keyboard of an ASR or VDT. This makes it unnecessary to echo characters read from these devices.

TIP software includes an SCI procedure LPWIDTH to set the maximum paper width for a line printer. The paper width set by LPWIDTH applies to all I/O to the printer from a TIP program and to SCI procedures CC and PF as well. When the user enters the procedure name LPWIDTH, DX10 requests the following information:

```
LINE PRINTER NAME
        PAPER WIDTH
```

The name is a DX10 device name; e.g., LP01. The paper width is an integer, the number of characters per line, and has a default value of 80. The procedure adds two to the value entered to allow for the line feed and carriage return.

## C.4 RUNTIME SIZES

The runtime sizes shown in the descriptions of runtime options in Section XIV are minimum sizes. Table C-1 lists additional features and the amount of additional code they require.

**Table C-1. Runtime Size Required for Additional Features**

| Feature | Size (Bytes) |
|---|---|
| Input/Output | 4350* |
| plus: | |
| For sequential or textfiles | 640* |
| For sequential files | 480 |
| For random files | 670 |

Table C-1. Runtime Size Required for Additional Features (Continued)

| Feature | Size (Bytes) |
|---|---|
| For textfiles | 900* |
| plus: | |
| For CHAR and string | 2190* |
| For INTEGER conversions | 1570* |
| For hexadecimal integers | 1640 |
| For LONGINT conversions | 1720 |
| For REAL conversions | 4600 |
| For FIXED conversions | 4480 |
| For DECIMAL conversions | 2320 |
| For BOOLEAN conversions | 1650 |
| For SKIPFILES | 890 |
| For SETMEMBER | 440 |
| plus: | |
| For miscellaneous I/O library procedures | 0 to 330 |
| REAL arithmetic, single precision | 2340 |
| plus: | |
| For double precision | 1370 |
| Mathematical functions, single precision | 1750 to 3630 |
| plus: | |
| For double precision also | 2440 to 4850 |
| Mathematical functions, double precision only | 2440 to 4850 |
| LONGINT arithmetic | 0 to 460 |
| FIXED arithmetic | 0 to 1770 |
| DECIMAL arithmetic | 1870 to 2070 |
| Type conversions | 0 to 2760 |
| SET operations | 0 to 260 |
| Default Heap manager (NEW and DISPOSE) | 160* |
| Heap manager with space recovery (NEW and DISPOSE) | 430* |
| ASSERT checks | 210 (Note 1) |
| Optional runtime checks | 1310 (Note 2) |
| CASE label checks | 350 (Note 3) |
| FORTRAN linkage | 410 (Note 4) |

Table C-1. Runtime Size Required for Additional Features (Continued)

| Feature | Size (Bytes) |
|---|---|
| Overlays (DX10 execution only) | 160 |
| PACK | 590 |
| UNPACK | 860 |
| Procedure linkage (per static nesting level) | 42 |

Notes: *Included in the minimum .TIP.OBJ and .TIP.LUNOBJ

1. ASSERT checks require only 40 bytes in addition to the minimum for .TIP.MINOBJ.
2. Optional runtime checks require only 230 bytes in addition to the minimum for .TIP.MINOBJ.
3. CASE label checks require only 70 bytes in addition to the minimum for .TIP.MINOBJ.
4. Includes only the additional bytes from the Pascal runtime library. The size of the specified FORTRAN subroutine or function and the FORTRAN runtime routines it uses must be added, also.

# APPENDIX D
# TIP SYNTAX

# APPENDIX D

# TIP SYNTAX

## D.1 GENERAL

Throughout the manual, TIP syntax is defined using BNF productions and syntax diagrams. This appendix summarizes the BNF productions and the syntax diagrams used to define the language.

## D.2 TIP BNF PRODUCTIONS

The BNF productions that define the TIP language syntax and appear throughout the manual are summarized in this paragraph. The productions are arranged in appropriate categories.

**D.2.1 TIP PROGRAM SYNTAX.** The BNF productions that define the syntax of a TIP program are:

```
<program>                        ::= <program heading> <block>.

<program heading>                ::= PROGRAM <program identifier>;

<program identifier>             ::= <identifier>

<block>                          ::= <declarations><compound statement>

<declarations>                   ::= <label declaration part>
                                     <constant declaration part>
                                     <type declaration part>
                                     <variable declaration part>
                                     <common declaration part>
                                     <access declaration part>
                                     <procedure and function declaration part>

<label declaration part>         ::= LABEL <integer constant>
                                     {, <integer constant>};
                                     |<empty>

<empty>                          ::=

<constant declaration part>      ::= CONST <constant declaration>
                                     { ; <constant declaration>};
                                     | <empty>

<constant declaration>           ::= <identifier> = <constant expression>

<type declaration part>          ::= TYPE <type declaration>
                                     { ; <type declaration>};
                                     |<empty>

<type declaration>               ::= <identifier> = <type>

<variable declaration part>      ::= VAR <variable declaration>
                                     { ; <variable declaration>};
                                     |<empty>
```

```
<variable declaration>                 ::= <identifier list>:<type>

<common delcaration part>              ::= COMMON <variable declaration>
                                           { ; <variable declaration>};
                                           | <empty>

<access declaration part>              ::= ACCESS <access declaration>;
                                           |<empty>

<access declaration>                   ::= <identifier list>

<identifier list>                      ::= <identifier> {,<identifier> }

<procedure and function declaration part>
                                       ::= <procedure or function declaration>
                                           { ; <procedure or function declaration>};
                                           |<empty>

<procedure or function declaration>    ::= <procedure declaration>
                                           |<function declaration>

<procedure declaration>                ::= <procedure heading> <block>
                                           |<procedure heading> FORWARD
                                           |<procedure heading> EXTERNAL <linkage>

<procedure heading>                    ::= PROCEDURE <identifier>;
                                           |PROCEDURE <identifier><parameter list>;

<parameter list>                       ::= ([<any parameter>{ ; <any parameter>}])

<any parameter>                        ::= <parameter> |
                                           PROCEDURE <identifier>
                                           [([[VAR] <type specification>
                                            {; [VAR] <type specification> }])] |
                                           FUNCTION <identifier>
                                           [([[VAR] <type specification>
                                            { ; [VAR] <specification>} ])] :
                                           <type specification>

<parameter>                            ::= <identifier list> : <partype>
                                           | VAR <identifier list> : <partype>

<partype>                              ::= <type specification> |
                                           <dynamic parameter type>

<dynamic parameter type>               ::= [PACKED] ARRAY "[" <parameter index>
                                           {, <parameter index>} "]" OF
                                           <type specification> |
                                           [PACKED] SET OF
                                           <dynamic parameter index type>
```

```
<type specification>                 ::= INTEGER|LONGINT|CHAR|BOOLEAN|
                                         REAL[(<integer constant>) ] |
                                         FIXED(<integer constant>,
                                         [<sign>] <integer constant>) |
                                         DECIMAL(<integer constant>,
                                         [<sign>] <integer constant>) |
                                         <type identifier>

<parameter index>                    ::= <subrange type> | <type identifier>
                                         | <dynamic parameter index type>

<dynamic parameter index type>       ::= <manifest constant>. . ?

<linkage>                            ::= PASCAL|FORTRAN|REENTRANT FORTRAN
                                         | <empty>

<function declaration>               ::= <function heading> <block> |
                                         <function heading> FORWARD |
                                         <function heading> EXTERNAL <linkage>

<function heading>                   ::= FUNCTION <identifier> [<parameter list>]:
                                         <result type>;

<result type>                        ::= <type identifier>
```

**D.2.2 TIP TYPE SYNTAX.** The BNF productions that define the syntax for type definition are:

```
<type>                               ::= <simple type>|<structured type>

<simple type>                        ::= <enumeration type>
                                         |<type identifier>
                                         |REAL[(<integer constant>)]
                                         |FIXED(<integer constant>,
                                          [<sign>]<integer constant>)
                                         |DECIMAL(<integer constant>,
                                          [<sign>]<integer constant>)

<type identifier>                    ::= <identifier>

<enumeration type>                   ::= INTEGER|LONGINT|BOOLEAN|CHAR|
                                         <scalar type>|<subrange type>

<scalar type>                        ::= (<scalar identifier> { , <scalar identifier> })

<subrange type>                      ::= <manifest constant>..<manifest constant>
```

```
<manifest constant>            ::= <enumeration constant> |
                                   <integer constant expression>

<enumeration constant>         ::= <character constant>
                                   |<Boolean constant>
                                   |<scalar identifier>
                                   |<integer constant>

<scalar identifier>            ::= <identifier>

<structured type>              ::= [PACKED] <unpacked structure type>
                                   |<pointer type>|<file type>

<unpacked structured type>     ::= <array type>|<record type>
                                   |<set type>

<array type>                   ::= ARRAY '['<index type> {, <index type>} ']'
                                   OF <component type>

<index type>                   ::= <static index type>|<dynamic index type>

<static index type>            ::= <enumeration type> | <type identifier>

<dynamic index type>           ::= <manifest constant>..<dynamic upper bound>

<dynamic upper bound>          ::= <entire variable>
                                   |UB(< dynamic array variable>
                                    [, <manifest constant>])

<entire variable>              ::= <variable identifier>

<variable identifier>          ::= <identifier>

<dynamic array variable>       ::= <identifier>

<component type>               ::= <type>

<record type>                  ::= RECORD <field list> END

<field list>                   ::= <fixed part>|<fixed part>;<variant part>
                                   |<variant part>

<fixed part>                   ::= <record section> {;<record section>}

<record section>               ::= [<field indentifier> {,<field identifier>}
                                   : <type>]

<field identifier>             ::= <identifier>

<variant part>                 ::= CASE <tagfield><type identifier> OF
                                   <variant>{;<variant>}
```

| | |
|---|---|
| <tagfield> | ::= [<identifier>: ] |
| <variant> | ::= [<case label list>: (<field list>)] |
| <case label list> | ::= <case label> {, <case label>} |
| <case label> | ::= <manifest constant> \|<manifest constant>..<manifest constant> |
| <set type> | ::= SET OF <base type> |
| <base type> | ::= <static base type> \| <dynamic base type> |
| <static base type> | ::= <enumeration type> \| <type identifier> |
| <dynamic base type> | ::= <manifest constant>..<dynamic bound> |
| <dynamic bound> | ::= <entire variable> \|UB( <dynamic set variable>) |
| <dynamic set variable> | ::= <identifier> |
| <pointer type> | ::= @<type identifier> |
| <file type> | ::= [RANDOM] FILE OF <type> \| TEXT |

**D.2.3 TIP STATEMENT SYNTAX.** The BNF productions that define the syntax for TIP statements are:

| | |
|---|---|
| <statement> | ::= [<statement label>:] <simple statement> \| [<statement label>:] [<escape label>:] <structured statement> |
| <compound statement> | ::= BEGIN <statement> { ;<statement> } END |
| <statement label> | ::= <integer constant> |
| <simple statement> | ::= <empty statement> \| <assignment statement> \| <procedure statement> \| <escape statement> \| <goto statement> \| <assert statement> |
| <empty statement> | ::= <empty> |
| <assignment statement> | ::= <variable> := <expression> |
| <procedure statement> | ::= <procedure identifier> [([<actual parameter> {, <actual parameter>} ])] |

```
<procedure identifier>        ::= <identifier>

<actual parameter>            ::= <expression> | <variable> |
                                  <procedure identifier> |
                                  <function identifier>

<escape statement>            ::= ESCAPE <escape label> |
                                  ESCAPE <procedure identifier> |
                                  ESCAPE <program identifier>

<escape label>                ::= <identifier>

<goto statement>              ::= GOTO <statement label>

<statement label>             ::= <digit> { <digit>}

<assert statement>            ::= ASSERT <expression>

<structured statement>        ::= <compound statement> |
                                  <conditional statement> |
                                  <repetitive statement> |
                                  <with statement>

<conditional statement>       ::= <if statement> | <case statement>

<if statement>                ::= IF <expression> THEN <statement>
                                  [ELSE <statement>]

<case statement>              ::= CASE <expression> OF <case element>
                                  { ; <case element>}
                              [ OTHERWISE <statement>{ ;<statement>} ] END

<case element>                ::= <case label list> : <statement> |
                                  <empty>

<case label list>             ::= <case label>{ , <case label>}

<case label>                  ::= <manifest constant> |
                                  <manifest constant>..<manifest constant>

<repetitive statement>        ::= <for statement> | <while statement> |
                                  <repeat statement>

<for statement>               ::= FOR <control variable> <generator>
                                  DO <statement>

<control variable>            ::= <identifier>

<generator>                   ::= IN <set expression> |
                                  :=<initial value> TO <final value> |
                                  :=<initial value> DOWNTO <final value>
```

```
<set expression>              ::= <expression>

<initial value>               ::= <expression>

<final value>                 ::= <expression>

<while statement>             ::= WHILE <expression> DO <statement>

<repeat statement>            ::= REPEAT <statement> { ; <statement> }
                                  UNTIL <expression>

<with statement>              ::= WITH <with variable list> DO
                                  <qualified statement>

<with variable list>          ::= <with variable> { , <with variable> }

<with variable>               ::= <record variable> |
                                  <identifier> = <record variable>

<qualified statement>         ::= <statement>
```

**D.2.4 TIP EXPRESSION SYNTAX.** The BNF productions that define the syntax for TIP expressions are:

```
<expression>                  ::= <Boolean term> |
                                  <expression> OR <Boolean term>

<Boolean term>                ::= <Boolean factor> |
                                  <Boolean term> AND <Boolean factor>

<Boolean factor>              ::= <Boolean primary> |
                                  NOT <Boolean primary>

<Boolean primary>             ::= <simple expression> | <Boolean primary>
                                  <relational operator> <simple expression>

<relational operator>         ::= =
                                  | <>
                                  | <
                                  | <=
                                  | >
                                  | >=
                                  | IN

<simple expression>           ::= <term> |
                                  <adding operator> <term> |
                              <simple expression> <adding operator> <term>

<adding operator>             ::= +
                                  | -

<term>                        ::= <factor> |
                                  <term> <multiplying operator> <factor>
```

```
<multiplying operator>         ::= *
                                   | /
                                   | DIV
                                   | MOD

<factor>                       ::= ( <expression> )
                               | <function identifier> [( [ <expression>
                               { , <expression> } ] )]
                                   | <set>
                                   | <unsigned constant>
                                   | <variable>

<function identifier>          ::= <identifier>

<set>                          ::= "[" <element list> "]"

<element list>                 ::= <element>    , <element>
                                   | <empty>

<element>                      ::= <expression>
                                   | <expression> .. <expression>

<empty>                        ::=

<unsigned constant>            ::= <constant identifier>
                                   | <Boolean constant>
                                   | <scalar identifier>
                                   | NIL
                                   | <character constant>
                                   | <string constant>
                                   | <integer constant>
                                   | <real constant>
                                   | <fixed-point constant>
                                   | <decimal constant>

<constant identifier>          ::= <identifier>

<scalar identifier>            ::= <identifier>
```

**D.2.5 TIP VARIABLE SYNTAX.** The BNF productions that define the syntax for TIP variables are:

```
<variable>                     ::= <entire variable>
                                   | <component variable>
                                   | <type-transferred variable>

<entire variable>              ::= <variable identifier>

<variable identifier>          ::= <identifier>

<component variable>           ::= <indexed variable>
                                   | <field designator>
                                   | <referenced variable>
```

| | |
|---|---|
| <indexed variable> | ::=<br><array variable> "[" <expression><br>{, <expression> } "]" |
| <array variable> | ::= <variable> |
| <field designator> | ::= <record variable> . <field identifier> |
| <record variable> | ::= <variable> |
| <field identifier> | ::= <identifier> |
| <referenced variable> | ::= <pointer variable> @ |
| <pointer variable> | ::= <variable> |
| <type-transferred variable> | ::= <variable> :: <type identifier> |
| <type identifier> | ::= <identifier> |

**D.2.6 TIP CONSTANT EXPRESSION SYNTAX.** The BNF productions that define the syntax for TIP constant expressions are:

| | |
|---|---|
| <constant expression> | ::= <constant term><br>\| <adding operator> <constant term><br>\| <constant expression> <adding operator> <constant term> |
| <adding operator> | ::= + \| - |
| <constant term> | ::= <constant factor><br>\| <constant term> <multiplying operator> <constant factor> |
| <multiplying operator> | ::= * \| / \| DIV \| MOD |
| <constant factor> | ::= ( <constant expression> )<br>\| <unsigned constant> |
| <unsigned constant> | ::= <constant identifier><br>\| <Boolean constant><br>\| <scalar identifier><br>\| NIL<br>\| <character constant><br>\| <string constant><br>\| <integer constant><br>\| <real constant><br>\| <fixed-point constant><br>\| <decimal constant> |
| <constant identifier> | ::= <identifier> |
| <scalar identifier> | ::= <identifier> |

**D.2.7 TIP INTEGER CONSTANT EXPRESSION SYNTAX.** The BNF productions that define the syntax for TIP integer constant expressions are:

```
<integer constant expression>        ::= <integer constant term>
                                     | <adding operator><integer constant term>
| <integer constant expression><adding operator><integer constant term>

<adding operator>                    ::=+ | -

<integer constant term>              ::= <integer constant factor>
| <integer constant term> <intmult operator> <integer constant factor>

<intmult operator>                   ::= * | DIV | MOD

<integer constant factor>            ::= ( <integer constant expression> )
                                         | <integer constant identifier>
                                         | <integer constant>

<integer constant identifier>        ::= <identifier>
```

**D.2.8 TIP LANGUAGE ELEMENT SYNTAX.** The BNF productions that define the syntax for the language elements of TIP are:

```
<symbol>                             ::= <special symbol>
                                         | <keyword symbol>
                                         | <identifier>
                                         | <constant>

<constant>                           ::= <enumeration constant>
                                         | <real constant>
                                         | <string constant>
                                         | <fixed-point constant>
                                         | <decimal constant>
                                         | <constant identifier>

<separator>                          ::= <space>
                                         | <end of the logical source record>
                                         | <comment>
                                         | <remark>

<comment>                            ::= <open comment> <any sequence of graphic
                                     characters not containing <close comment> >
                                        <close comment>

<open comment>                       ::= " { " | (*

<close comment>                      ::= " } " | *)

<remark>                             ::= " <any sequence of graphic characters extending to
                                     the end of the logical source record>

<special symbol>                     ::= +|-|*|/|=|<|>|(|)|.|,|;|:|@|?
                                         |"[" |"]" |(. |.)|<=|>=|<>|..|:=|::
```

```
<keyword symbol>              ::=ACCESS|AND|ARRAY|ASSERT|BEGIN|BOOLEAN
                                 |CASE|CHAR|CONST|COMMON|DECIMAL|DIV|DO
                                 |DOWNTO|ELSE|END|ESCAPE|FALSE|FILE|FIXED
                                 |FOR|FUNCTION|GOTO|IF|IN|INPUT|INTEGER
                                 |LABEL|LONGINT|MOD|NIL|NOT|OF|OR|OTHERWISE
                                 |OUTPUT|PACKED|PROCEDURE|PROGRAM|RANDOM
                                 |RECORD|REAL|REPEAT|SET|TEXT|THEN|TO
                                 |TRUE|TYPE|UNTIL|VAR|WHILE|WITH

<identifier>                  ::= <letter> { <letter>|_|<digit> }

<letter>                      ::= A|B|C|D|E|F|G|H|I|J|K|L|M|N
                                  |O|P|Q|R|S|T|U|V|W|X|Y|Z|$

<digit>                       ::= 0|1|2|3|4|5|6|7|8|9

<Boolean constant>            ::= FALSE | TRUE

<character constant>          :: ‘ <character> ’

<string constant>             ::= ‘ <character><character> {<character> } ’

<character>                   ::= <graphic character>
                                  | #<hexdigit><hexdigit>

<graphic character>           ::= <special character>
                                  | <letter>
                                  | <digit>
                                  | <space>
                                  | < nonstandard character>

<special character>           ::= +|-|*|/|=|<|>|(|)|.|,|;|:|@|?
                                  |‘ ’ |“|##|_|[|]| |

<space>                       ::= “ ”

< nonstandard character>      ::= <any other character available
                                  on a particular system or device>

<hexdigit>                    ::= <digit>|A|B|C|D|E|F

<integer constant>            ::= <digit> { <digit> } [L]
                                  | #<hexdigit> { <hexdigit> } [L]

<real constant>               ::= [<sign>] <digits> . <digits>
                                   | [<sign>]<digits>.<digits>E<scale factor>
                                   | [<sign>] <digits>.<digits>Q<scale factor>
                                   | [<sign>] <digits>E<scale factor>
                                   | [<sign>] <digits>Q<scale factor>

<digits>                      ::= <digit> {<digit>}
```

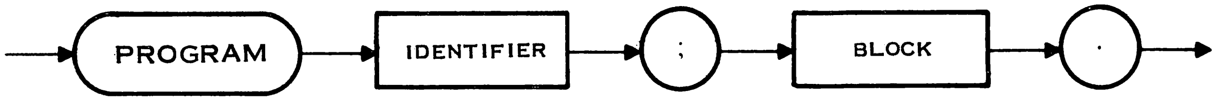
| | |
|---|---|
| <scale factor> | ::= <digits> <br> \| <sign><digits> |
| <sign> | ::= +\|- |
| <fixed-point constant> | ::= <digits>F <br> \| <digits>.<digits>F <br> \| <binary digits>B <br> \| <binary digits>.<binary digits>B |
| <binary digits> | ::= <binary digit>{ <binary digit>} |
| <binary digit> | ::= 0\|1 |
| <decimal constant> | ::= <digits>.<digits>D <br> \| <digits>D |

## D.3 TIP SYNTAX DIAGRAMS

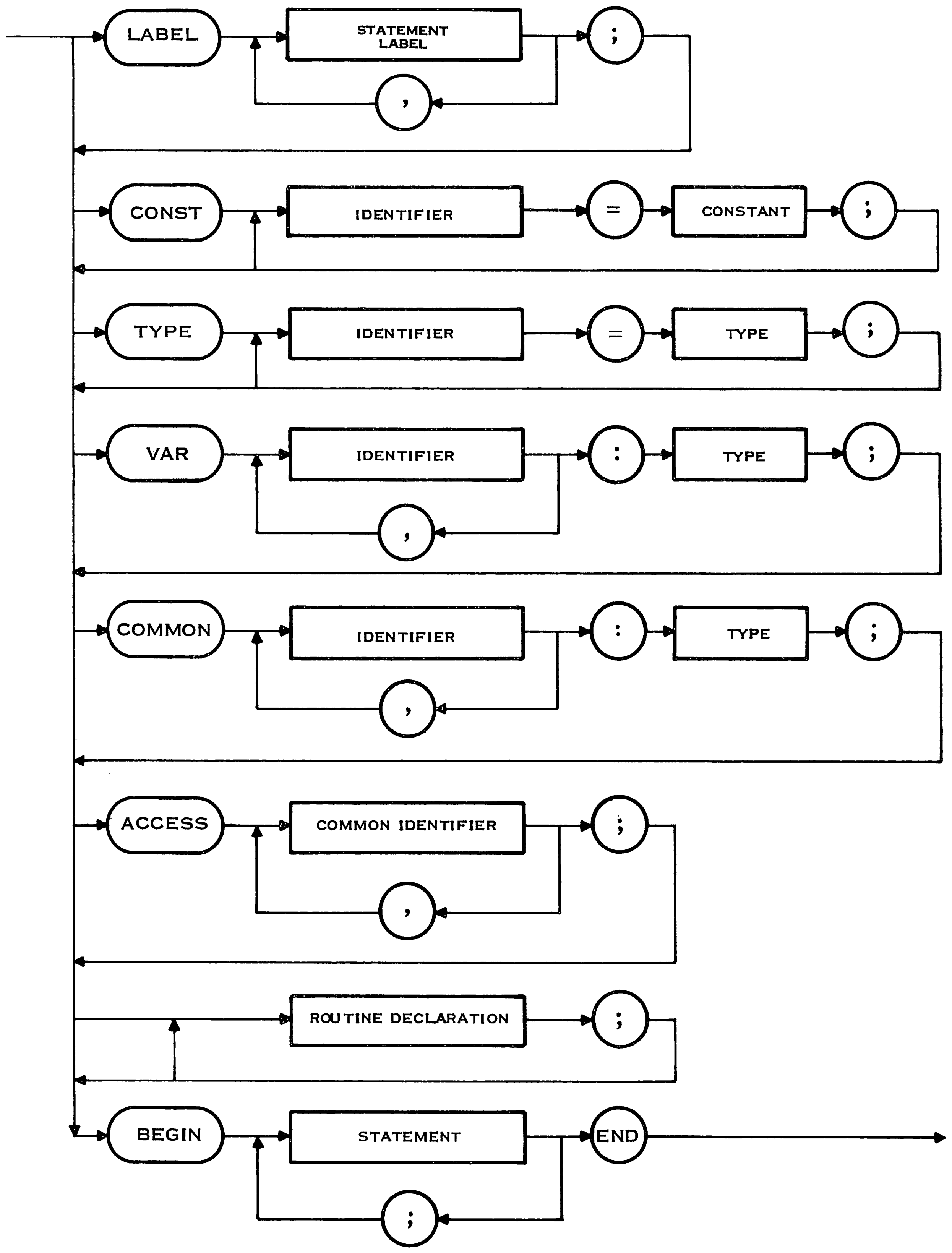
The syntax diagrams that appear throughout the manual to define the TIP language are summarized in this paragraph. The arrangement of syntax diagrams parallels that of the BNF productions in the preceding paragraphs.

**D.3.1 TIP PROGRAM.** The syntax diagrams that define the syntax of a TIP program are:

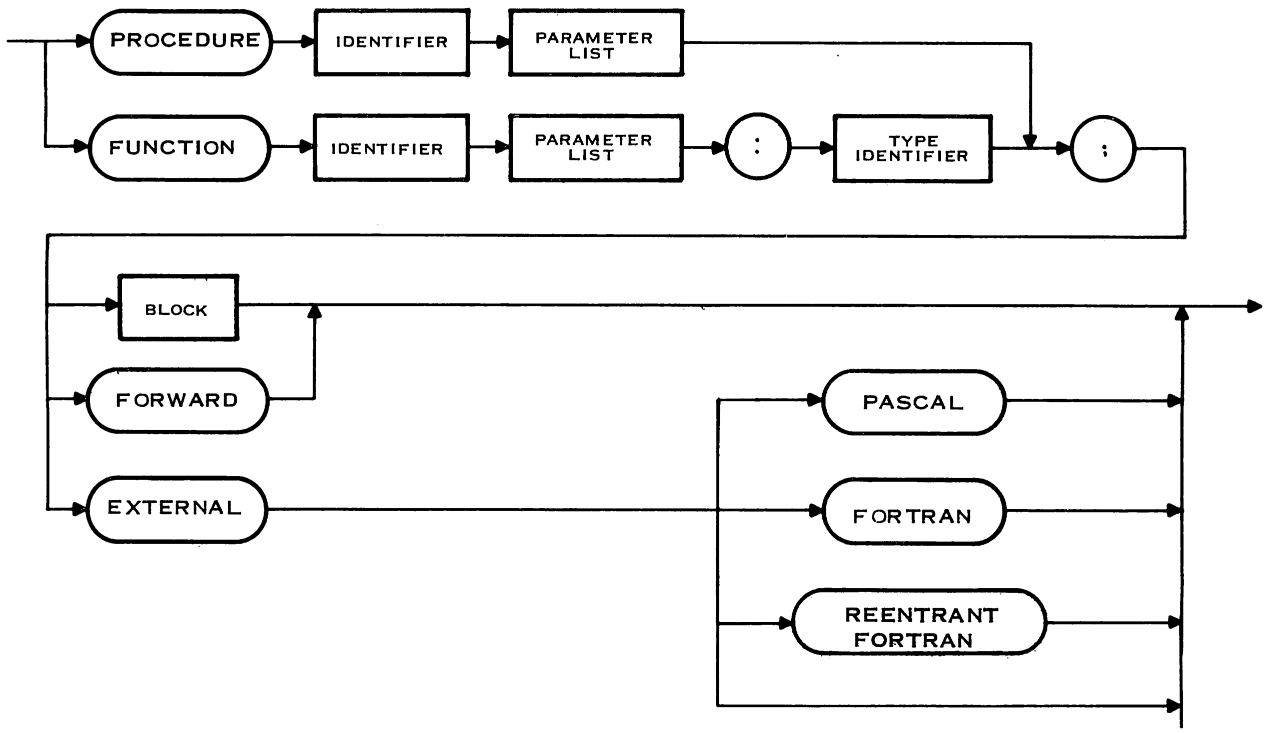
Program:



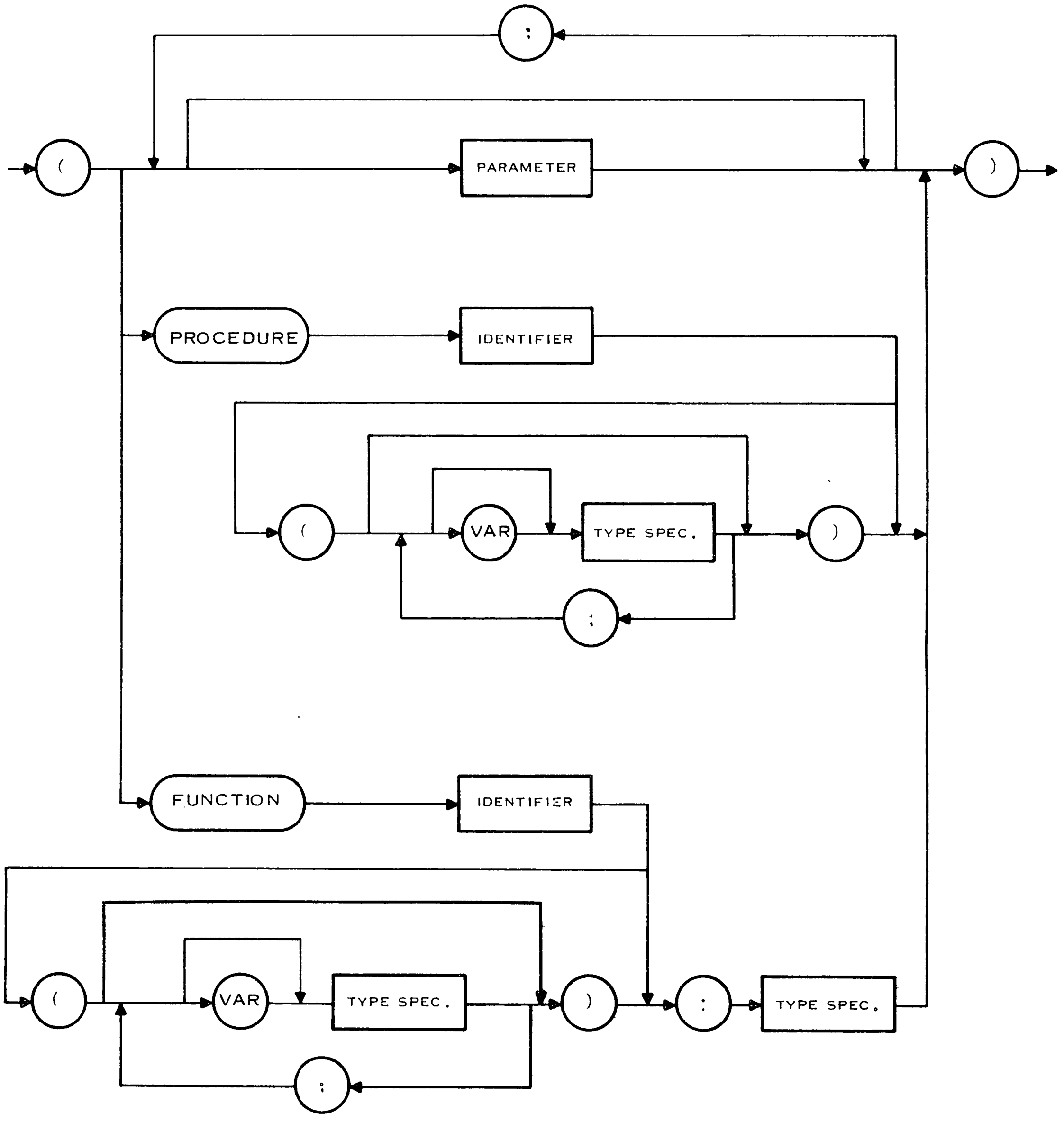
Block:


LABEL
STATEMENT LABEL
,
;
CONST
IDENTIFIER
=
CONSTANT
;
TYPE
IDENTIFIER
=
TYPE
;
VAR
IDENTIFIER
,
:
TYPE
;
COMMON
IDENTIFIER
,
:
TYPE
;
ACCESS
COMMON IDENTIFIER
,
;
ROUTINE DECLARATION
;
BEGIN
STATEMENT
;
END

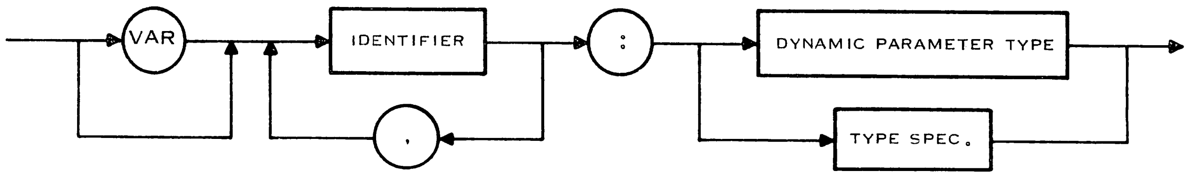
Routine declaration:



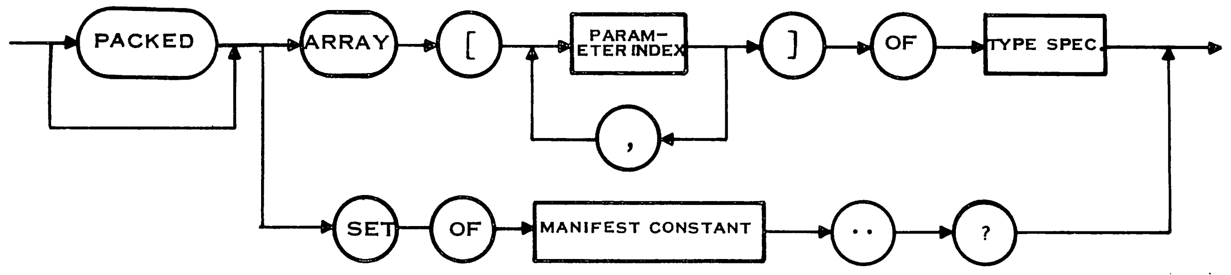
Parameter list:



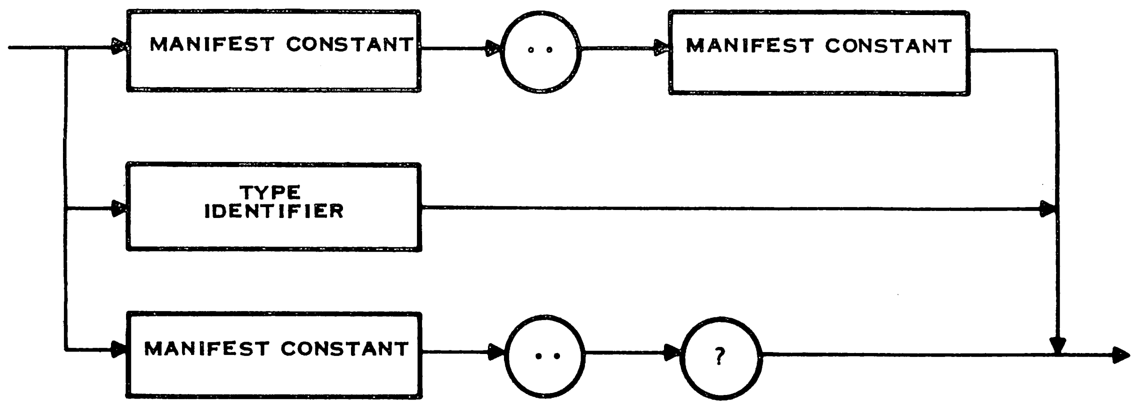
Parameter:



Dynamic parameter type:

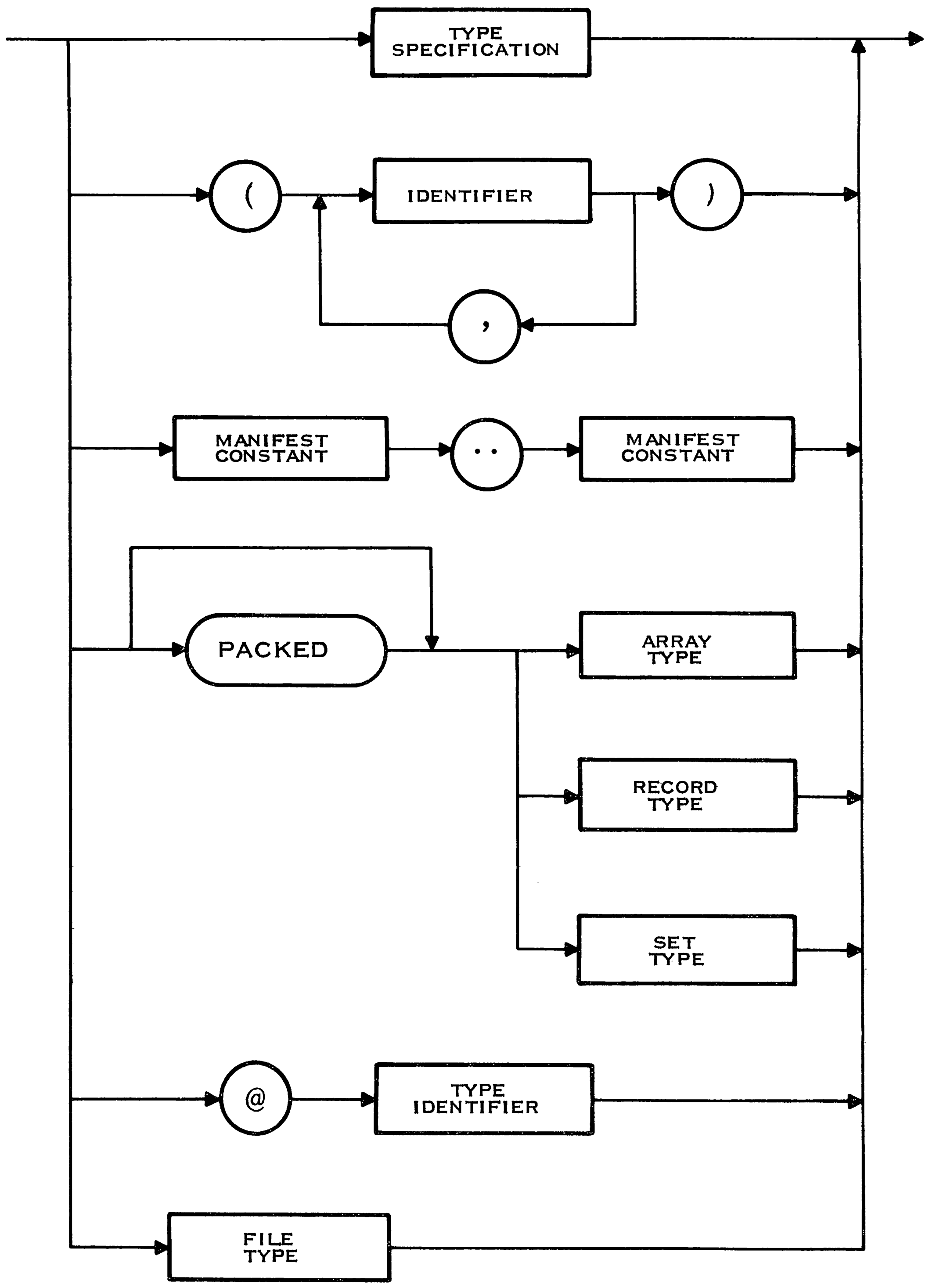


Parameter index:

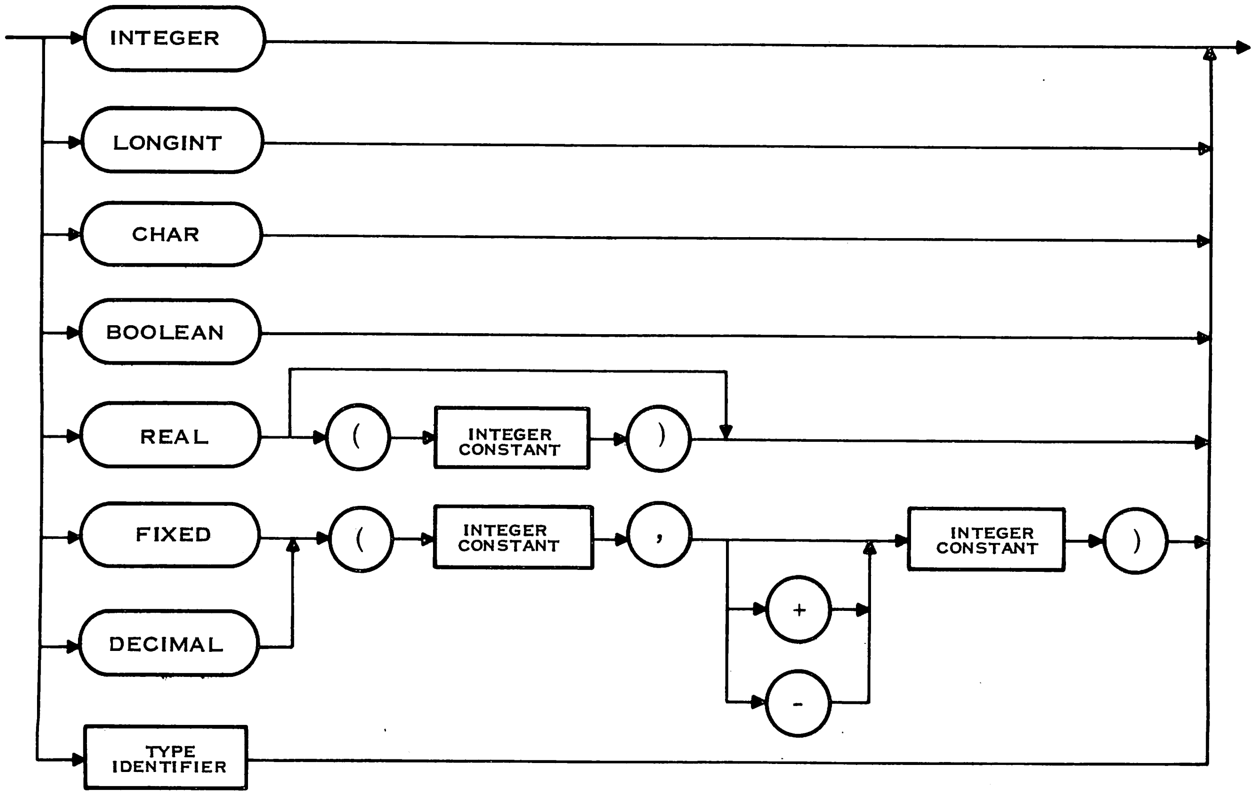


**D.3.2 TIP TYPE.** The syntax diagrams that define the syntax for type definition are:

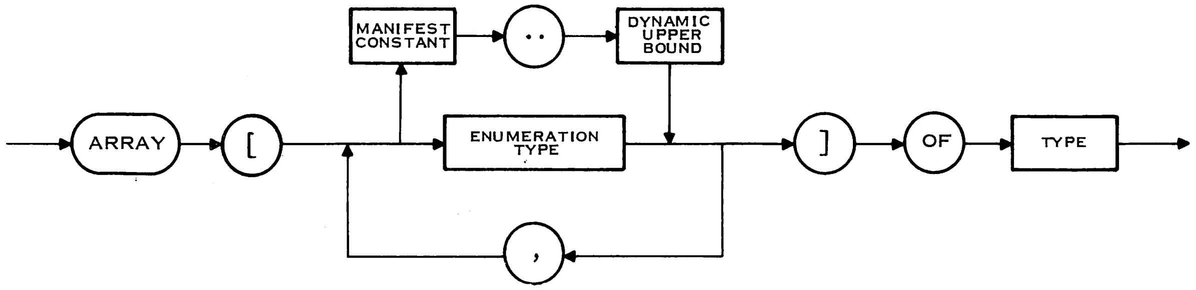
Type:



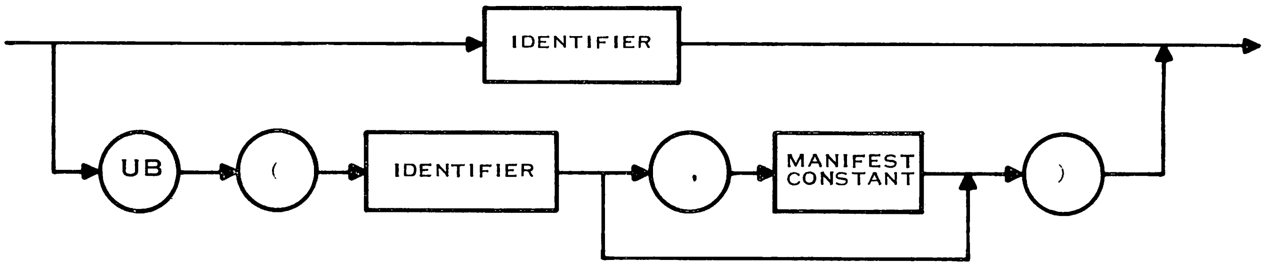
Type specification:



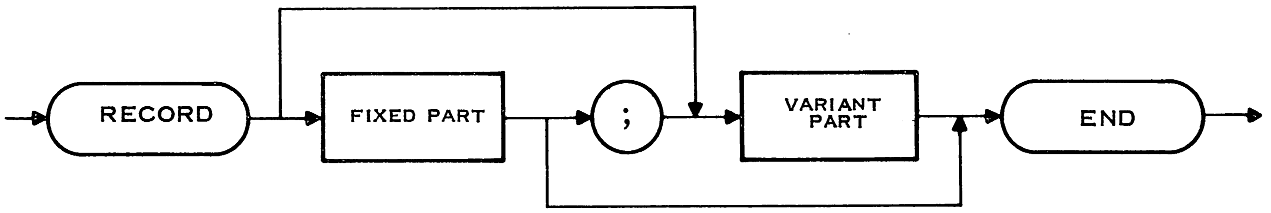
Array type:



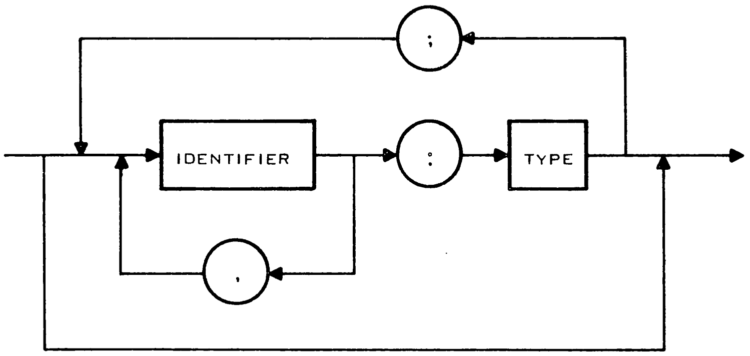
Dynamic upper bound:



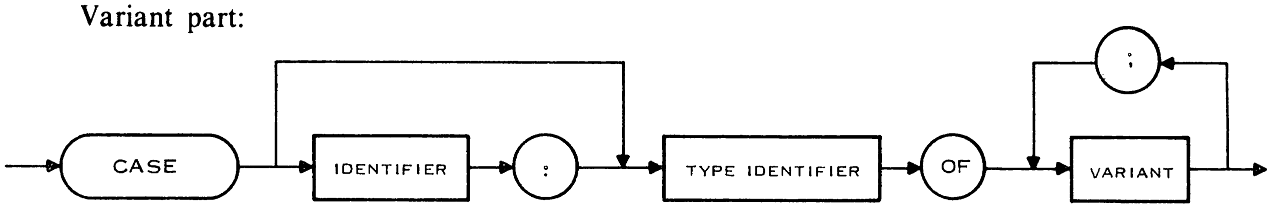
Record type:



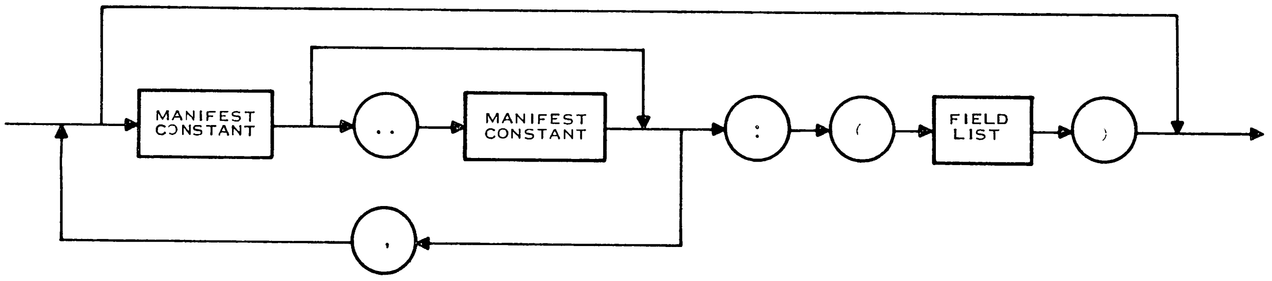
Fixed part:



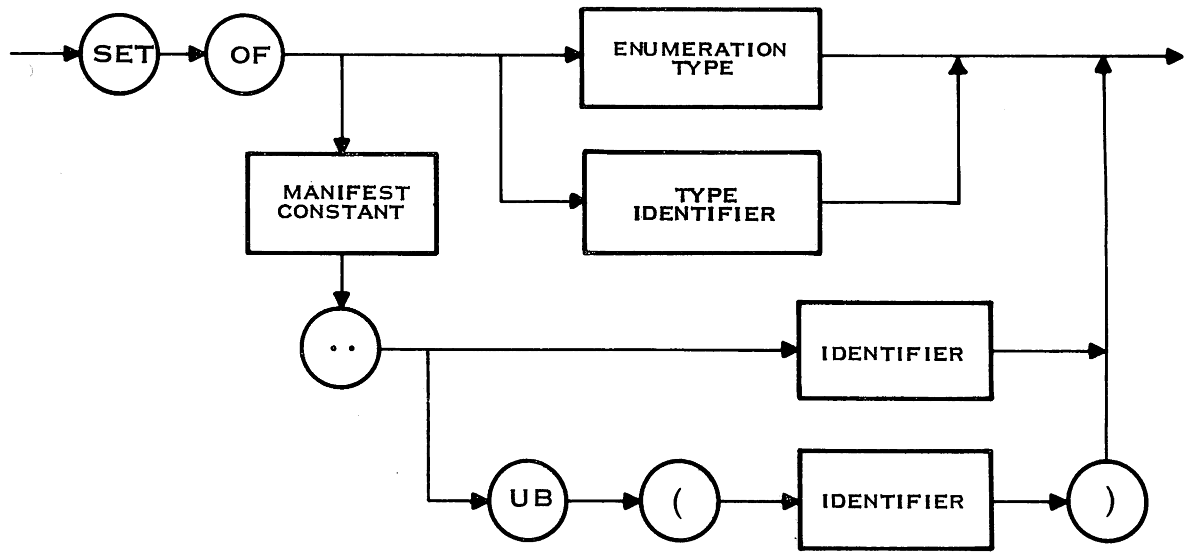
Variant part:



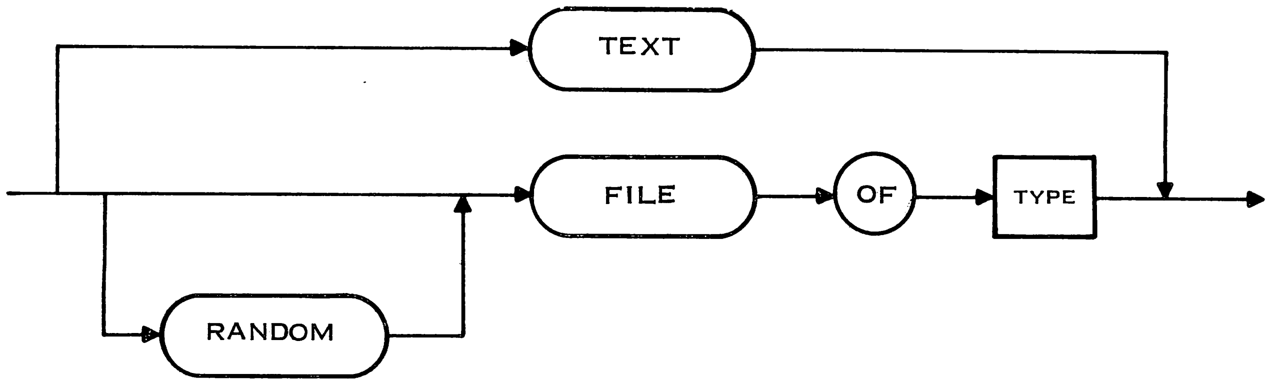
Variant:



Set type:

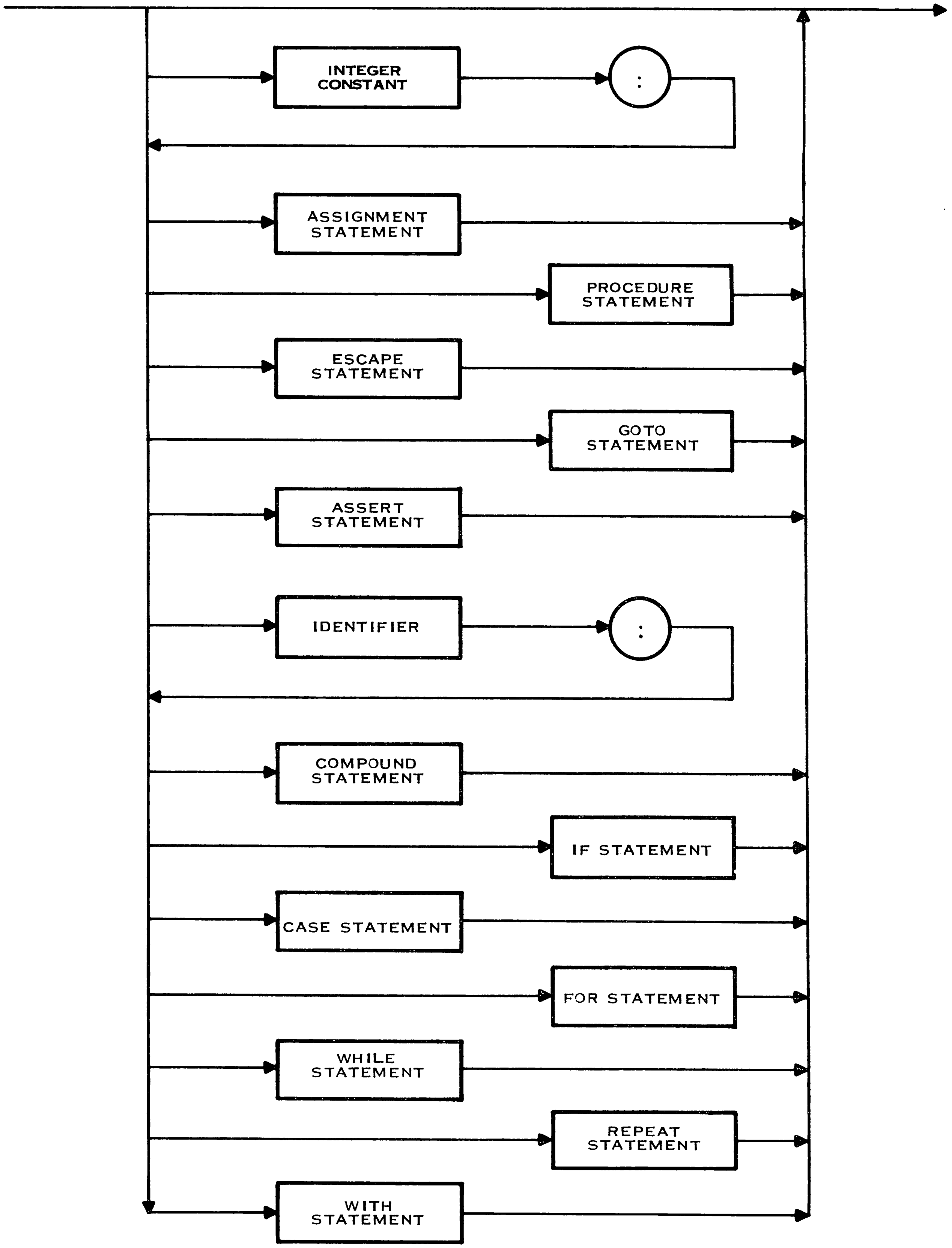


File type:

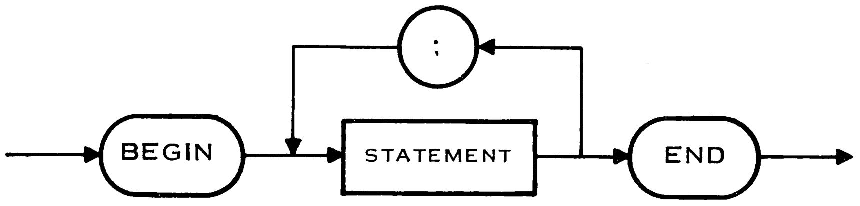


**D.3.3 TIP STATEMENT.** The syntax diagrams that define the syntax for TIP statements are:

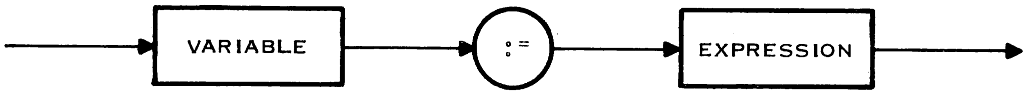
Statement:



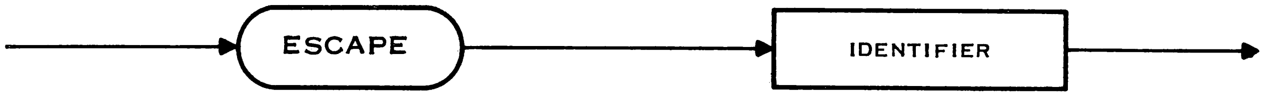
Compound statement:



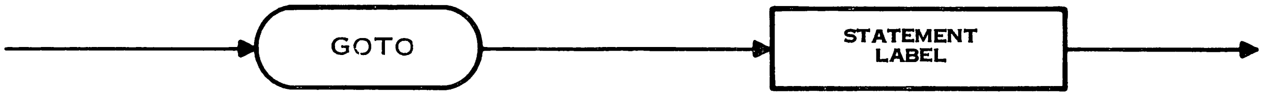
Assignment statement:



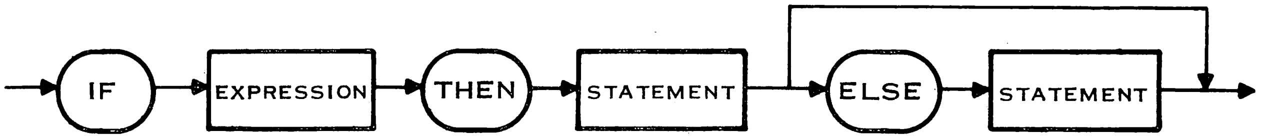
ESCAPE statement:



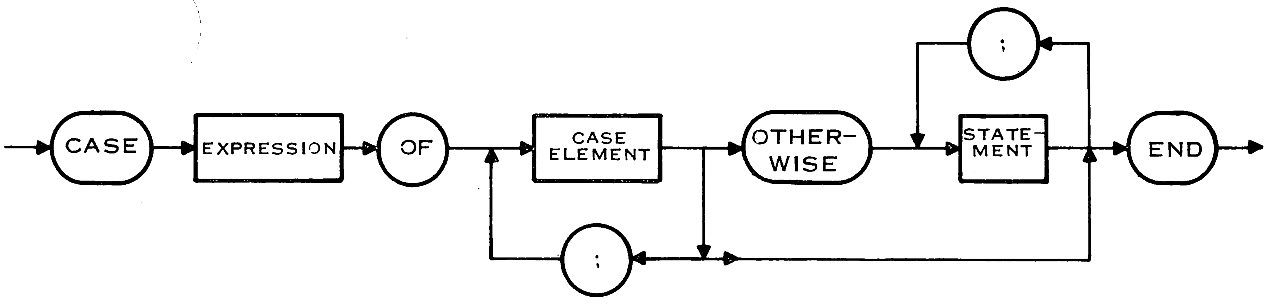
GOTO statement:



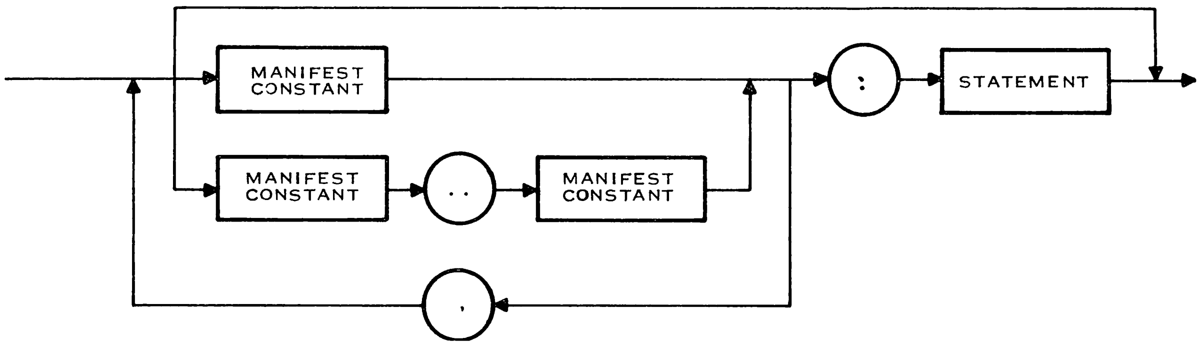
IF statement:



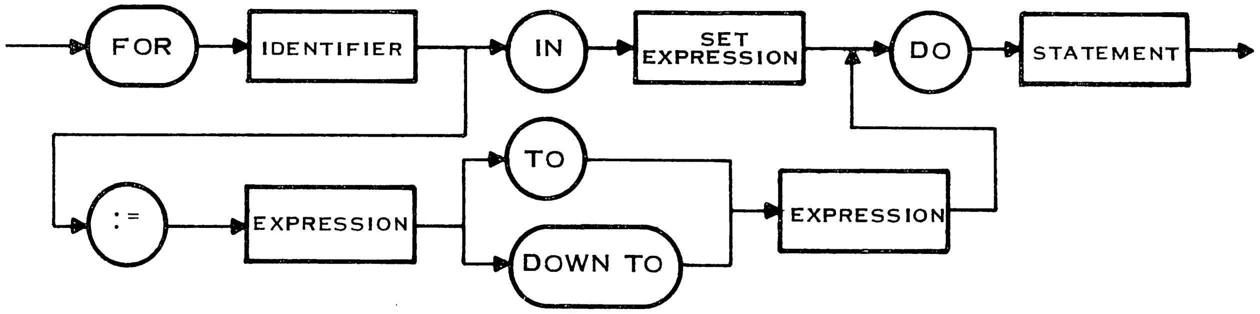
CASE statement:



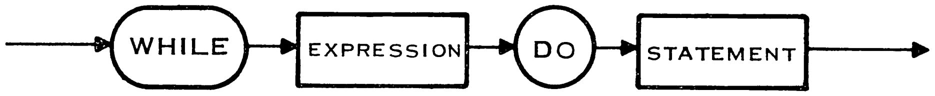
CASE element:



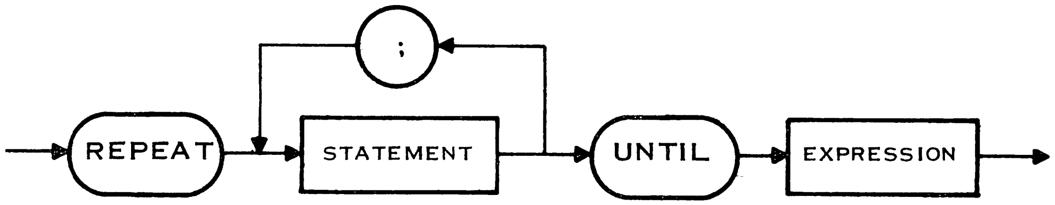
FOR statement:



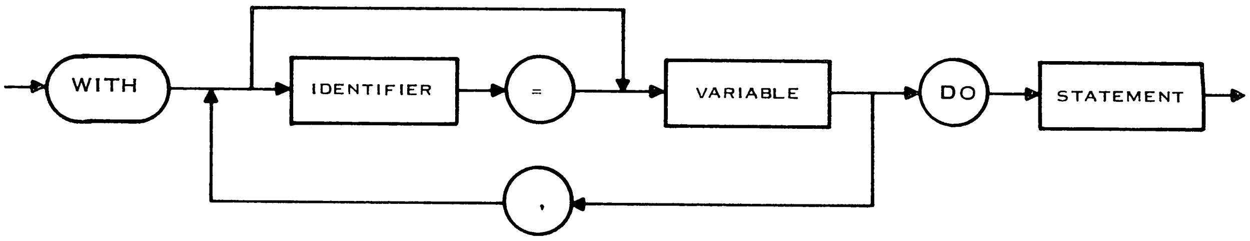
WHILE statement:



REPEAT statement:

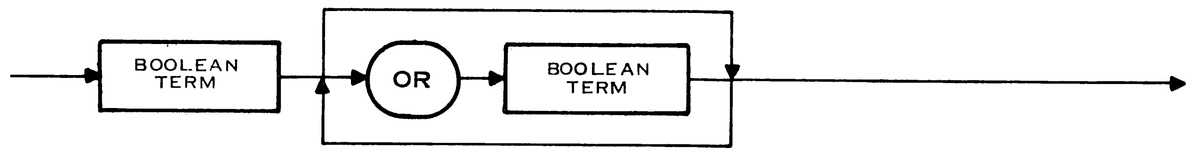


WITH statement:

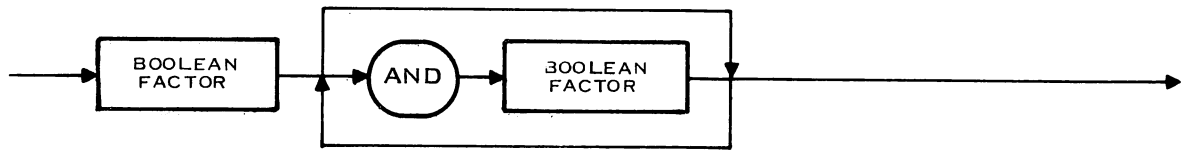


**D.3.4 TIP EXPRESSION.** The syntax diagrams that define the syntax for TIP expressions are:

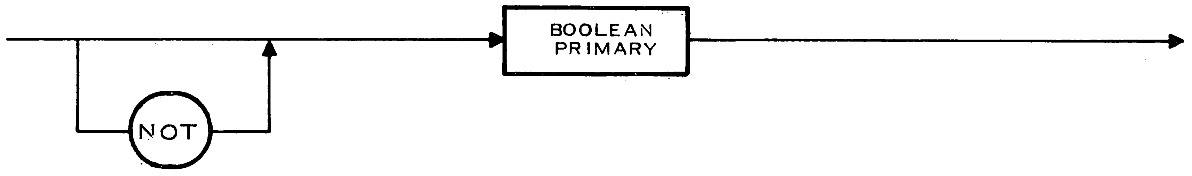
Expression:



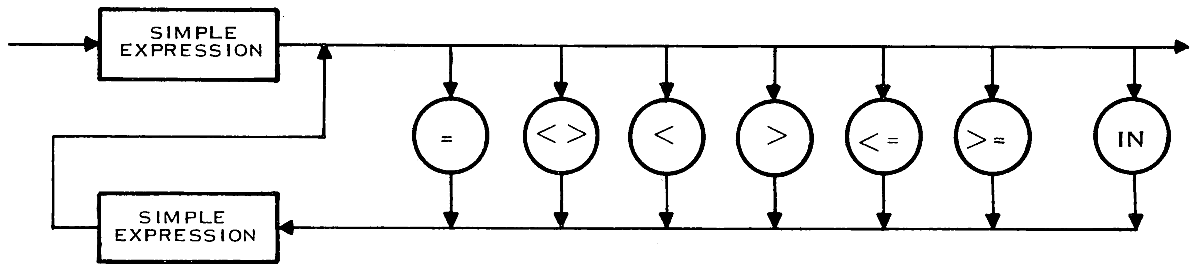
Boolean term:



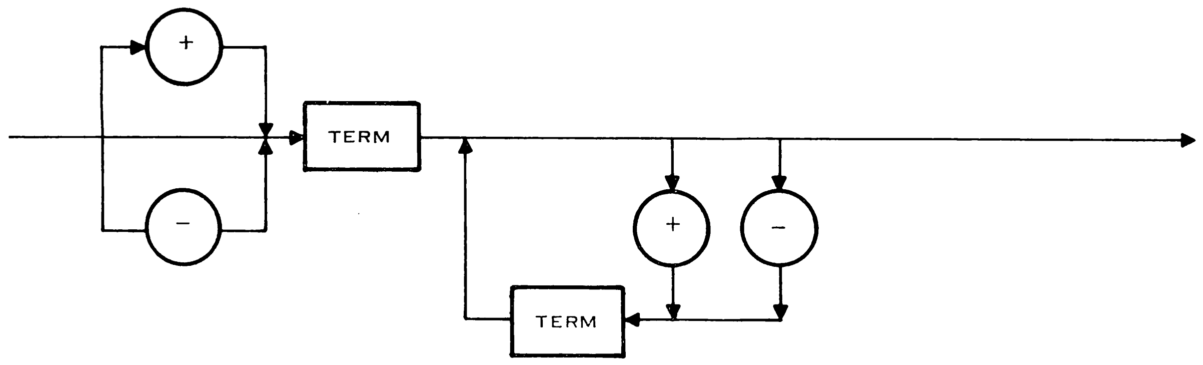
Boolean factor:



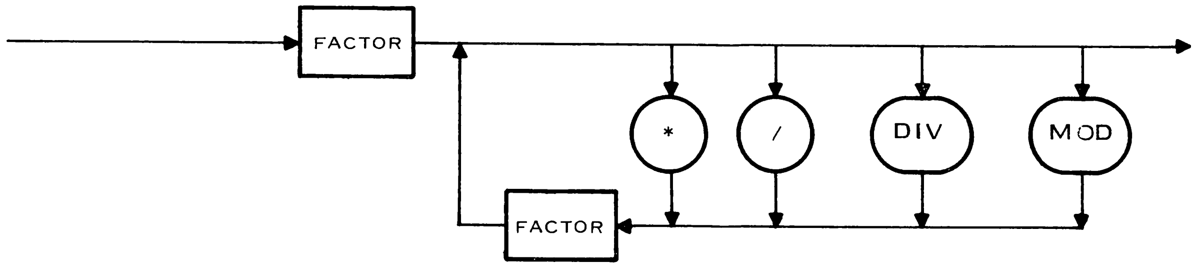
Boolean primary:



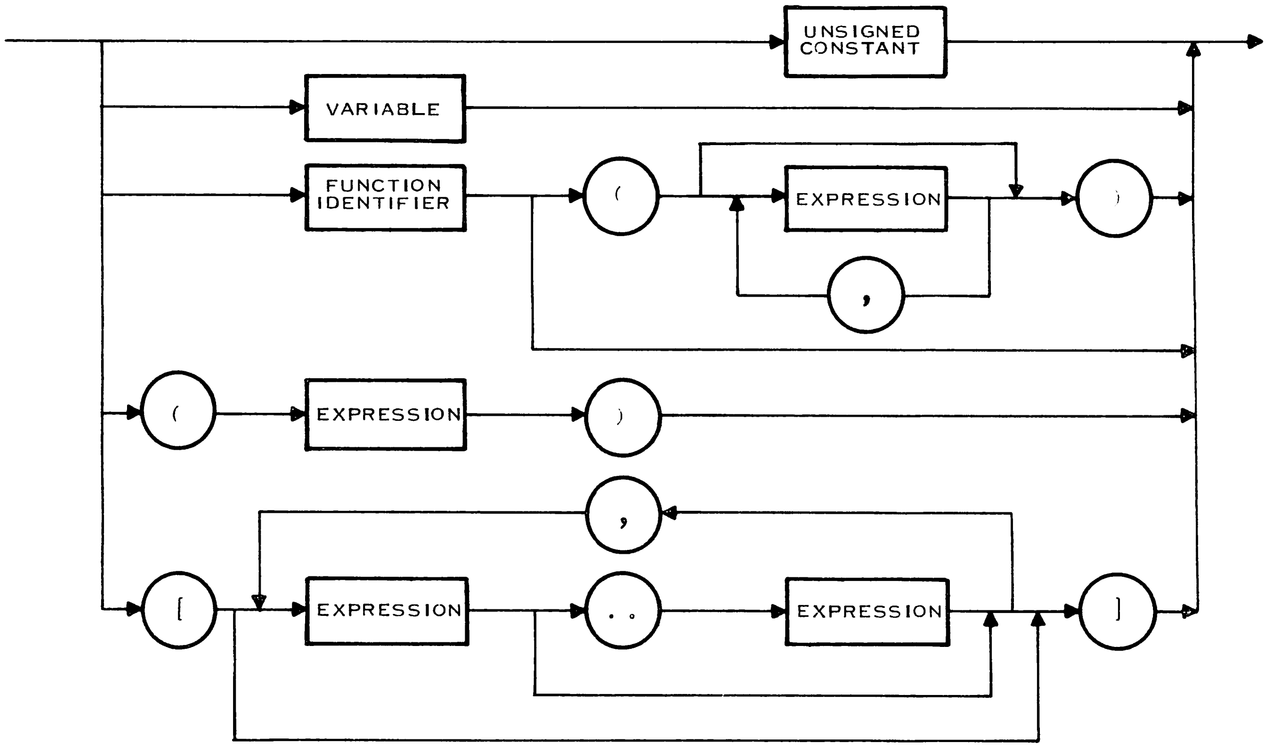
Simple expression:



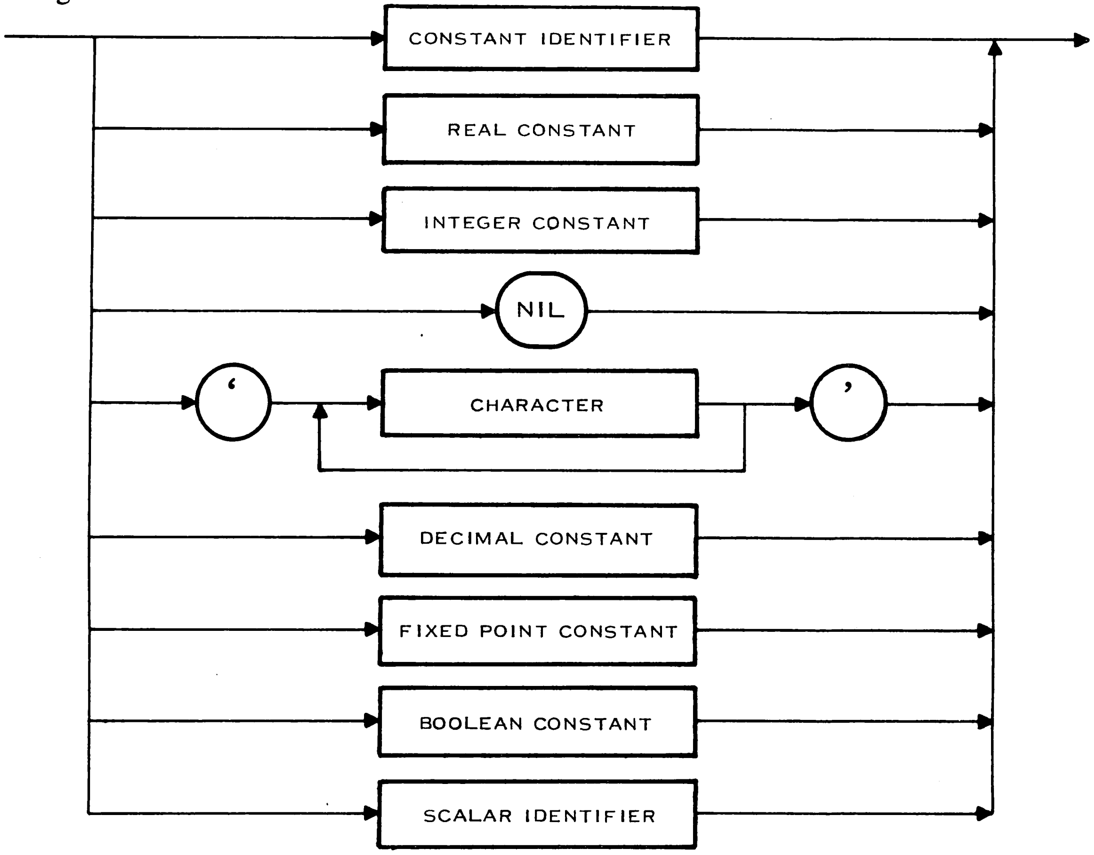
Term:


FACTOR
*
/
DIV
MOD
FACTOR

Factor:

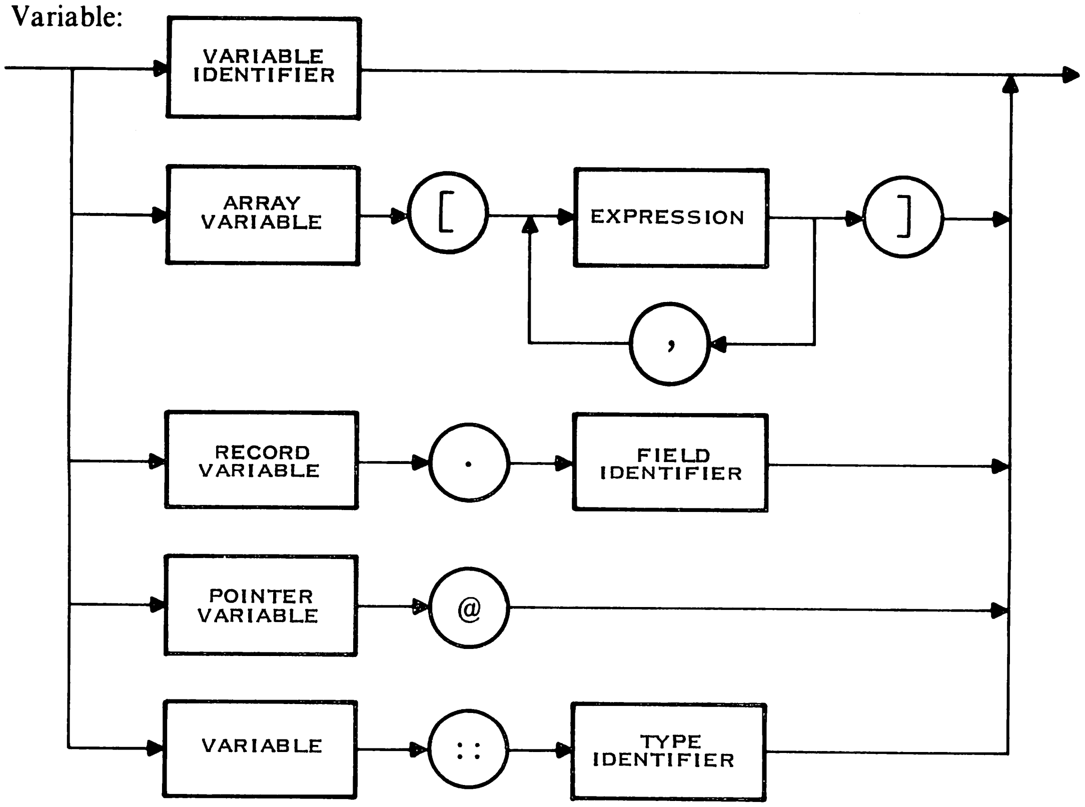

UNSIGNED CONSTANT
VARIABLE
FUNCTION IDENTIFIER
(
EXPRESSION
,
)
(
EXPRESSION
)
,
[
EXPRESSION
..
EXPRESSION
]

Unsigned constant:



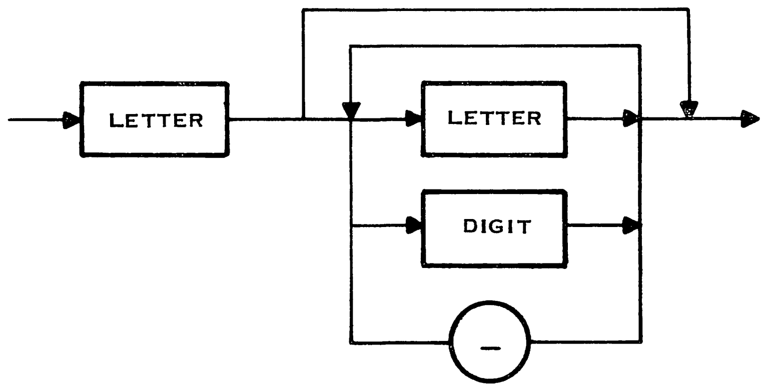
**D.3.5 TIP VARIABLE.** The syntax diagram that defines the syntax for a TIP variable is:

Variable:

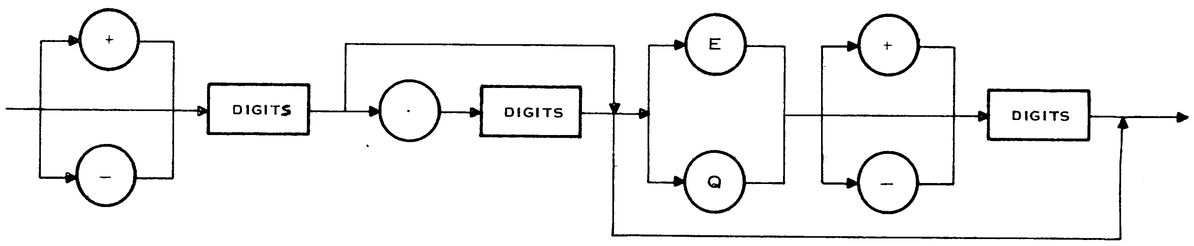


**D.3.6 TIP LANGUAGE ELEMENT.** The syntax diagrams that define the syntax for TIP language elements are:

Identifier:



Real constant:



## APPENDIX E
## ERROR MESSAGES

# APPENDIX E

# ERROR MESSAGES

## E.1 COMPILER ERROR MESSAGES

This paragraph lists the error messages issued by the TIP compiler. The error number is shown, followed by a letter and the error message text. The letter is either W, E, or F. A warning message is identified by a W; an E identifies a message describing an error that the compiler may be able to correct; an F identifies a message describing a fatal error. The messages are:

| | | |
|---|---|---|
| 1 | E | ERROR IN SIMPLE TYPE |
| 2 | E | IDENTIFIER EXPECTED |
| 3 | E | 'PROGRAM' EXPECTED |
| 4 | E | ')' EXPECTED |
| 5 | E | ':' EXPECTED |
| 6 | E | ILLEGAL SYMBOL |
| 7 | E | PARAMETER EXPECTED |
| 8 | E | 'OF' EXPECTED |
| 9 | E | '(' EXPECTED |
| 10 | E | ERROR IN TYPE |
| 11 | E | '(.' EXPECTED |
| 12 | E | '.)' EXPECTED |
| 13 | E | 'END' EXPECTED |
| 14 | E | ';' EXPECTED |
| 15 | E | INTEGER EXPECTED |
| 16 | E | '=' EXPECTED |
| 17 | E | 'BEGIN' EXPECTED |
| 18 | E | ERROR IN DECLARATION SECTION |
| 19 | E | ERROR IN FIELD LIST |
| 20 | E | ',' EXPECTED |
| 22 | E | '..' EXPECTED |
| 23 | E | '.' EXPECTED |
| 24 | E | ';' NOT ALLOWED |
| 40 | E | ILLEGAL PARAMETER TYPE |
| 42 | E | STATEMENT TERMINATOR EXPECTED |
| 43 | E | STATEMENT EXPECTED |
| 45 | W | 'EXTERNAL FORTRAN' EXPECTED |
| 49 | E | 'ARRAY' EXPECTED |
| 50 | E | CONSTANT EXPECTED |
| 51 | E | ':=' EXPECTED |
| 52 | E | 'THEN' EXPECTED |
| 53 | E | 'UNTIL' EXPECTED |
| 54 | E | 'DO' EXPECTED |
| 55 | E | 'TO' / 'DOWNTO' EXPECTED |
| 57 | E | 'FILE' EXPECTED |
| 58 | E | ERROR IN FACTOR |
| 60 | E | 'CHAR' EXPECTED |
| 61 | E | UNARY '+' / '-' NOT ALLOWED IN TERM, OR 'NOT' IN BOOLEAN PRIMARY |
| 62 | E | EXPRESSION OR '.)' EXPECTED |
| 63 | E | '..' ',' OR '.)' EXPECTED |

| | | |
|---|---|---|
| 64 | E | ',' OR '.)' EXPECTED |
| 65 | E | USE 'DIV' FOR INTEGER '/' |
| 66 | E | TYPE IDENTIFIER EXPECTED |
| 67 | W | QUESTION MARK UPPER BOUND EXPECTED |
| 68 | E | PROGRAM PARAMETERS NOT IMPLEMENTED |
| 80 | W | OPTION IDENTIFIER EXPECTED |
| 81 | W | ILLEGAL OPTION IDENTIFIER |
| 82 | W | PROGRAM SENSITIVE OPTION MAY NOT BE CONTROLLED HERE |
| 83 | W | STATEMENT SENSITIVE OPTION MAY NOT BE CONTROLLED HERE |
| 84 | W | NULL BODY EXPECTED |
| 101 | E | IDENTIFIER DECLARED TWICE |
| 102 | F | LOWER BOUND EXCEEDS UPPER BOUND |
| 103 | F | IDENTIFIER IS NOT OF APPROPRIATE CLASS |
| 104 | E | UNDECLARED IDENTIFIER |
| 105 | F | CLASS OF IDENTIFIER IS NOT VARIABLE |
| 107 | E | INCOMPATIBLE SUBRANGE TYPES |
| 108 | E | FILE NOT ALLOWED HERE |
| 109 | E | TYPE OF IDENTIFIER MUST BE ARRAY OR SET |
| 110 | E | TAGFIELD MUST BE SCALAR OR SUBRANGE |
| 111 | E | RECORD VARIANT CONSTANT INCOMPATIBLE WITH TAGFIELD TYPE |
| 117 | E | UNSATISFIED FORWARD REFERENCE TO A TYPE IDENTIFIER OF A POINTER |
| 119 | E | ';' EXPECTED (PARAMETER LIST NOT ALLOWED) |
| 120 | E | FUNCTION RESULT MUST BE SCALAR, SUBRANGE, OR POINTER |
| 121 | E | FILE VALUE PARAMETER NOT ALLOWED |
| 122 | E | ';' EXPECTED (FUNCTION RESULT NOT ALLOWED) |
| 123 | E | FUNCTION RESULT EXPECTED |
| 126 | F | IMPROPER NUMBER OF PARAMETERS |
| 127 | F | TYPE OF ACTUAL PARAMETER DOES NOT MATCH FORMAL PARAMETER |
| 128 | E | PARAMETER INCOMPATIBLE WITH PREVIOUS PARAMETER (DYNAMIC ARGUMENT) |
| 129 | E | TYPE CONFLICT OF OPERANDS IN AN EXPRESSION |
| 130 | F | EXPRESSION IS NOT OF SET TYPE |
| 131 | W | INTEGER OPERANDS CONVERTED TO REAL FOR '/' |
| 132 | W | BASE TYPE OF SET IS LONG INTEGER |
| 133 | W | LONG INTEGER ELEMENT IN SET |
| 134 | F | ILLEGAL TYPE OF OPERANDS |
| 135 | F | TYPE OF EXPRESSION MUST BE BOOLEAN |
| 136 | F | SET ELEMENT TYPE MUST BE SOME ENUMERATION TYPE |
| 137 | F | SET ELEMENT TYPE NOT COMPATIBLE |
| 138 | F | TYPE OF VARIABLE IS NOT ARRAY |
| 139 | F | INDEX TYPE IS NOT COMPATIBLE WITH DECLARATION |
| 140 | F | TYPE OF VARIABLE IS NOT RECORD |
| 141 | F | TYPE OF VARIABLE IS NOT POINTER |
| 142 | F | TYPE OF VARIABLE IS NOT FUNCTION |
| 143 | F | INCOMPATIBLE "FOR" EXPRESSIONS |
| 145 | F | TYPE CONFLICT IN ASSIGNMENT |
| 146 | F | ASSIGNMENT OF FILE NOT ALLOWED |
| 147 | F | LABEL TYPE INCOMPATIABLE WITH CASE SELECTOR |
| 148 | E | SET BOUNDS OUT OF RANGE |
| 149 | E | INDEX TYPE MAY NOT BE INTEGER |
| 150 | F | ONLY ASSIGNMENT TO LOCAL FUNCTION ALLOWED |

| | | |
|---|---|---|
| 151 | F | ASSIGNMENT TO FORMAL FUNCTION IS NOT ALLOWED |
| 152 | E | NO SUCH FIELD IN THIS RECORD |
| 154 | F | ACTUAL PARAMETER MUST BE A VARIABLE |
| 155 | F | CANNOT ASSIGN TO “FOR” CONTROL VARIABLE |
| 156 | E | MULTIDEFINED CASE LABEL |
| 161 | W | PROCEDURE OR FUNCTION ALREADY DECLARED AT A PREVIOUS LEVEL |
| 162 | E | PROCEDURE OR FUNCTION AGAIN DECLARED FORWARD OR EXTERNAL |
| 165 | F | MULTIDEFINED LABEL |
| 167 | F | UNDECLARED LABEL |
| 168 | W | UNDEFINED LABEL |
| 170 | E | PROCEDURE PARAMETERS NOT ALLOWED FOR FUNCTIONS |
| 171 | E | ERROR IN FIXED PRECISION |
| 172 | E | ERROR IN DECIMAL PRECISION |
| 173 | E | ERROR IN FIXED OR DECIMAL SCALE FACTOR |
| 177 | F | EXPRESSION OPERANDS ARE NOT COMPATIBLE |
| 179 | E | STATEMENT TO BE ESCAPED MUST BE A STRUCTURED STATEMENT |
| 180 | E | ESCAPING BROTHERS PROCEDURES NOT ALLOWED |
| 181 | E | CANNOT “GOTO” INTO A “FOR” OR “WITH” STATEMENT |
| 182 | F | “FOR” EXPRESSION MUST BE OF SOME ENUMERATION TYPE |
| 183 | F | “CASE” EXPRESSION MUST BE OF SOME ENUMERATION TYPE |
| 184 | W | IMPLICIT “FOR” IDENTIFIER ALREADY DECLARED |
| 185 | E | RECORD VARIANT LABEL MUST BE NONNEGATIVE AND LESS THAN “SETMAX” |
| 186 | E | MULTIDEFINED VARIANT LABEL |
| 187 | W | SHOULD NOT USE TYPE TRANSFER IN FUNCTION |
| 188 | W | SHOULD NOT USE “LOCATION” IN FUNCTION |
| 189 | W | SHOULD NOT ASSIGN THROUGH LOCAL POINTER IN FUNCTION |
| 190 | F | CANNOT CALL A PROCEDURE FROM A FUNCTION |
| 191 | F | CANNOT CALL AN EXTERNAL FUNCTION FROM A FUNCTION |
| 192 | F | CANNOT ASSIGN TO A GLOBAL VARIABLE, A REFERENCE PARAMETER, OR THROUGH A GLOBAL POINTER IN A FUNCTION |
| 193 | E | ACCESS TO GLOBAL VARIABLE NOT DECLARED |
| 194 | E | FILE ELEMENT MUST NOT BE FILE, POINTER, OR CONTAIN POINTERS |
| 195 | E | TYPE OF COMMON MAY NOT BE A FILE |
| 196 | E | DYNAMIC ARRAYS OR SETS NOT ALLOWED IN FILES, RECORDS, OR COMMONS |
| 197 | E | ACTUAL REFERENCE PARAMETER CANNOT BE “PACKED” OR “FOR” IDENTIFIER |
| 198 | F | ILLEGAL TYPE TRANSFER |
| 201 | E | FRACTION EXPECTED |
| 202 | E | STRING CONSTANT TOO LONG OR CROSSES A CARD BOUNDARY |
| 203 | E | INTEGER CONSTANT TOO LARGE |
| 205 | E | BINARY DIGIT EXPECTED |
| 206 | E | EXPONENT EXPECTED |
| 207 | E | HEXADECIMAL DIGIT EXPECTED |
| 208 | E | ILLEGAL LONG INTEGER CONSTANT |
| 210 | E | FIELD WIDTH MUST BE OF TYPE INTEGER OR LONGINT |
| 211 | E | FRACTION LENGTH MUST BE OF TYPE INTEGER OR LONGINT |
| 212 | E | HEX FORMAT ALLOWED ONLY FOR TYPE INTEGER OR LONGINT |
| 213 | E | BINARY FORMAT ALLOWED ONLY FOR FIXED TYPE |

| | | |
|---|---|---|
| 214 | E | F-FORMAT ALLOWED FOR REAL, DECIMAL, OR FIXED ONLY |
| 215 | E | RANDOM FILE RECORD NUMBER MUST BE OF TYPE INTEGER OR LONGINT |
| 216 | E | READ/WRITE PARAMETER NOT COMPATIBLE WITH FILE TYPE |
| 217 | F | GLOBAL REFERENCE PARAMETER NOT ALLOWED IN FUNCTION |
| 218 | W | READ/WRITE OF A GLOBAL FILE IN A FUNCTION |
| 219 | E | PARAMETER MUST BE OF TYPE FILE |
| 220 | E | PARAMETER MUST BE OF TYPE INTEGER OR LONGINT |
| 221 | F | INCORRECT NUMBER OF PARAMETERS IN STANDARD PROCEDURE CALL |
| 222 | F | INCORRECT NUMBER OF PARAMETERS IN STANDARD FUNCTION CALL |
| 223 | F | PARAMETER MUST BE OF TYPE POINTER |
| 224 | E | PARAMETER MUST BE NON-NEGATIVE INTEGER CONSTANT |
| 225 | E | PARAMETER MUST BE A CONSTANT |
| 226 | E | MISSING CORRESPONDING VARIANT DECLARATION |
| 227 | E | PARAMETER MUST BE A CONSTANT OR A VARIABLE |
| 228 | E | MANIPULATION OF GLOBAL FILE NOT ALLOWED IN FUNCTION |
| 229 | F | ILLEGAL TYPE OF PARAMETER IN STANDARD FUNCTION CALL |
| 230 | F | ILLEGAL TYPE OF PARAMETER IN STANDARD PROCEDURE CALL |
| 231 | E | PARAMETER MUST BE AN INTEGER OR LONGINT CONSTANT |
| 232 | F | ACTUAL PARAMETER HAS WRONG STRING LENGTH |
| 233 | W | MANIPULATION OF FILE “INPUT”/“OUTPUT” NOT RECOMMENDED |
| 235 | E | DIMENSIONALITY OF ARRAY SMALLER THAN SPECIFIED CONSTANT |
| 236 | E | ARRAY COMPONENT TYPES ARE NOT COMPATIBLE |
| 237 | E | ARGUMENT MUST BE PACKED ARRAY |
| 238 | E | ARGUMENT MUST BE NON-PACKED ARRAY |
| 239 | E | FILE ARGUMENT IS THE WRONG FILE KIND |
| 240 | W | MAXIMUM REAL PRECISION EXCEEDED |
| 241 | E | ARRAY TOO LONG |
| 242 | W | RUN-TIME COMPATIBILITY CHECK REQUIRED |
| 243 | E | DYNAMIC UPPER BOUND MUST BE GLOBAL VARIABLE OR PARAMETER |
| 250 | E | TOO MANY NESTED SCOPES |
| 251 | E | IMPROPER PARAMETER TYPE FOR ACTUAL PROCEDURE PARAMETER |
| 252 | E | MISMATCHED REFERENCE PARAMETER FOR ACTUAL PROCEDURE PARAMETER |
| 253 | E | VARIABLE PARAMETER EXPECTED FOR ACTUAL PROCEDURE PARAMETER |
| 254 | E | IMPROPER NUMBER OF PARAMETERS FOR ACTUAL PROCEDURE |
| 257 | F | TOO MANY IDENTIFIERS DECLARED |
| 260 | W | FORTRAN ARRAY AND RECORD VALUE PARAMETERS WILL BE PASSED BY REFERENCE |
| 261 | E | FORTRAN PROCEDURE AND FUNCTION PARAMETERS NOT ALLOWED |
| 262 | W | DYNAMIC ACTUAL PARAMETER WILL BE PASSED BY REFERENCE TO FORTRAN |
| 300 | E | DIVISION BY ZERO |
| 301 | E | SCALAR “SUCC” OR “PRED” RESULT OUT OF BOUNDS |
| 302 | F | INDEX EXPRESSION OUT OF BOUNDS |
| 303 | E | VALUE TO BE ASSIGNED IS OUT OF BOUNDS |
| 304 | F | SET ELEMENT EXPRESSION IS OUT OF BOUNDS |

## E.2 RUNTIME ERROR MESSAGES

The Pascal runtime library routines write error messages on the SYSMSG file and the OUTPUT file. Some of the I/O error messages include the two-digit status code returned by DX10. These codes are listed in the *Model 990 Computer DX10 Operating System Release 3 Reference Manual, Vol. 6.* The I/O status codes more commonly returned are:

| Code | Explanation |
|---|---|
| 01 | Unassigned logical unit number |
| 06 | Device timeout or abort |
| 0F | Another program has exclusive access to device |
| 1A | Disk unit write protected |
| 21 | Invalid device or volume name |
| 27 | Specified file name does not exist |
| 3B | Another program has exclusive access to file |
| 43 | Magnetic tape unit is offline |
| 44 | Write to tape with no write ring |
| 72 | Attempt to open a directory or program file |
| A1 | Directory is full |
| E0 | Disk volume is full |

Some error conditions result in a brief message on the SYSMSG file and a more detailed explanation on the OUTPUT file. The following list of messages includes the messages that are placed on the OUTPUT file; abbreviations of these messages appear on the SYSMSG file.

Some error messages specify a statement number, which is the line number within the body of the routine. The routine name is shown in the memory dump (paragraph 14.7.4); a source listing written with option WIDELIST in effect lists the line numbers in the second column from the left (figure 9-1).

The runtime error messages are as follows:

2nd ERROR

An error condition was detected while processing a previously detected error condition. Execution is terminated immediately. This will occur when the OUTPUT file cannot be opened; failure of the attempt to open the OUTPUT file to write the memory dump causes this message to be written.

ADDRESSING ERROR

The program has attempted to access a memory location outside the memory region of the program. Two common causes are:

- Accessing an array with an invalid index value.
- Invalid value in a pointer variable.

"ASSERT" FAILED

The value of the expression in the ASSERT statement on the line number specified in the OUTPUT file is FALSE.

ATTEMPT TO SKIP BEYOND EOM

An attempt has been made to skip beyond the end-of-medium (EOM) in a call to procedure SKIPFILES. Correct the program to prevent the call. When EOF is true following execution of SKIPFILES, the file is at EOM and no further call to SKIPFILES should be made.

ATTEMPT TO READ PAST EOF

The program has attempted to read from a file that is positioned at EOF. The program should test for end-of-file by calling function EOF. If it is desired to read a file that follows the EOF of the same medium, call procedure SKIPFILES to position the file at the beginning of the next file.

CANNOT LOAD OVERLAY

The library routine OVLY$ has received an error status code from the operating system when attempting a load overlay operation (supervisor call code $14_{16}$).

CANNOT GET MEMORY

Memory required for the program plus the memory requested for stack and heap exceeds the amount of available memory. Try to execute the program with smaller stack and heap values.

"CASE" ALTERNATIVE ERROR

The selector expression of a CASE statement has a value that is not equal to any of the CASE labels, and there is no OTHERWISE clause. The OUTPUT file message includes the statement number and the selector value.

CODE = ____ PC = ____ STACK: ____ HEAP: ____

A program using the minimal runtime library has terminated. The code value is one of the error codes listed in paragraph E.3. The PC value is the contents of the Program Counter when the error occurred, or zero for normal termination. The stack and heap values are hexadecimal numbers of bytes of stack and heap, respectively, used by the program.

DIVIDE BY ZERO

An integer or fixed-point division was attempted with a divisor of zero.

DYNAMIC ARRAY SIZE OUT OF RANGE

The value of the size of a dynamic array was either zero or negative when the routine that contains the array was entered.

ERROR CODE = _____

An error has occurred. The value is one of the error codes listed in paragraph E.3.

ESCAPE-FROM-ROUTINE ERROR

An ESCAPE statement that references a routine name has been executed, but the statement is not within the scope of the named routine.

FATAL COMPILATION ERRORS

An attempt to execute a routine that has fatal compilation errors has been made; the name of the routine follows this message.

FIELD EXCEEDS RECORD SIZE

The field specified for a formatted READ from or WRITE to a textfile is longer than the logical record length of the file.

FIELD WIDTH TOO LARGE

A parameter of a call to a WRITE procedure has a field length longer than the logical record length of the file.

FILE _____ ACCESS NAME _____

An I/O error occurred on the file the name of which follows the word FILE. The access name, if known, or the word UNKNOWN, follows the words ACCESS NAME. A message describing the nature of the problem follows.

FLOATING POINT ERROR — ILLEGAL INSTRUCTION

The floating point interpreter did not recognize the code of a floating point instruction. Either CODEGEN wrote bad object code or the program code area has been modified, possibly by storing data at the wrong location. The OUTPUT file contains the bad instruction and the address following the instruction. See table E-1.

FLOATING POINT ERROR — DIVISION BY ZERO

The divisor of a division operation of REAL numbers was zero.

FLOATING POINT ERROR — OVERFLOW

A floating point operation has resulted in a value that is too large to be represented (magnitude greater than $10^{75}$). The OUTPUT file contains the instruction that produced the error and the address following the instruction. Table E-1 lists the instruction codes.

FLOATING POINT ERROR — RELINK FOR EXTENDED PRECISION

The runtime routines for double precision REAL values were not linked with a program that attempts to perform double precision REAL operations. Add the following to the link control file:

```
INCLUDE (FL$ITD)
INCLUDE (TEN$D)
```

**Table E-1. Floating-Point Instruction Values**

| Single Precision Instruction | Double Precision Instruction | Operation |
|---|---|---|
| 0C00 | 0C01 | Convert REAL to INTEGER |
| 0C02 | 0C03 | Negate |
| 0C04 | 0C05 | Convert REAL to LONGINT |
| 0C06 | 0C07 | Convert LONGINT to REAL |
| 0C40-0C7F | 0E40-0E7F | Add |
| 0C80-0CBF | 0E80-0EBF | Convert INTEGER to REAL |
| 0CC0-0CFF | 0EC0-0EFF | Subtract |
| 0D00-0D3F | 0F00-0F3F | Multiply |
| 0D40-0D7F | 0F40-0F7F | Divide |
| 0D80-0DBF | 0F80-0FBF | Load |
| 0DC0-0DFF | 0FC0-0FFF | Store |

FTN ARGUMENT ERROR

A FORTRAN subroutine or function has been called with the wrong number of arguments.

FTN FLOATING POINT ERROR

A floating-point arithmetic error has occurred while executing a FORTRAN routine. The error may be overflow, divide by zero, illegal instruction, or double precision routines required.

FTN STACK OVERFLOW

Insufficient stack space remained for a call to a reentrant FORTRAN routine in the program.

HALT CALLED

Execution has been terminated by a call to the library routine HALT, which may be either in the user's program or in a runtime library routine. When a library routine calls HALT, a message that gives the reason for the HALT is also written. The additional message either precedes this message on the SYSMSG file or is written on the OUTPUT file.

HEAP FULL

The specified amount of heap space for executing the program is inadequate. Execute the program again with a larger amount of heap space.

I/O ERROR: ___ _____ NAME= _______

An I/O error has occurred. Two numbers follow the words I/O ERROR. The first of these is a reason code, one of the following:

| | |
|---|---|
| 0 | Open error. |
| 4 | Read error. |
| 5 | Attempt to read past EOF, or attempt to skip beyond EOM, or file or device not opened for reading. |

| | |
|---|---|
| 6 | Write error. |
| 7 | Not opened for writing. |

The reason code is followed by the status code returned by the operating system. The status code has significance when the reason code is 0, 4, or 6. Refer to the partial list of status codes at the beginning of this paragraph, or to the complete list referenced there. A reason code of 0 and a status code of 00 is a special case that means open error — element size greater than record length. The NAME shown is the name of the file.

ILLEGAL OPCODE

The program attempted to execute a machine instruction that is illegal. This may be the result of storing data in the program area or of executing a program having unresolved references in the linking operation.

ILLEGAL SUPERVISOR CALL

The program either executed a supervisor call that referenced an invalid supervisor call code or executed an XOP instruction for an undefined extended operation. The first byte of the area referenced by a supervisor call as the supervisor call block is interpreted as the supervisor call code. The second operand of an XOP instruction specifies the extended operation. This error may be the result of storing data in the program area.

ILLEGAL TILINE ADDRESS

The operating system has terminated the program with a task error code 3, indicating that an illegal TILINE address has been detected. This is either the address of a non-existent memory or of a non-existent device.

INCOMPLETE DATA

A READ operation of a textfile obtained a value that is syntactically incomplete. For example, the value 1.0E for a REAL number is incomplete. Correct the data in the file and execute the program again.

INVALID CHARACTER IN FIELD

A READ operation of a textfile obtained a character that is invalid for the specified data type. For example, a decimal point (.) is invalid in an integer value.

INVALID LENGTH OF PACKET

The DISPOSE routine has been called with a pointer operand that does not contain the address of a valid heap packet.

INVALID PACKET POINTER

The DISPOSE routine has been called with a pointer operand that does not contain the address of a valid heap packet.

INVALID RELEASE POINTER

The RELEASE routine has been called with a parameter that is not within the allocated portion of the current heap region.

MATH PACK ERROR -- ARGUMENT TOO LARGE IN _______

The value of the parameter of a call to the SIN, COS, or EXP function is too large. The name of the called function follows the word IN. For the SIN and COS the absolute value of the operand must be less than $2^{55}$.

MATH PACK ERROR -- DIVISION BY ZERO IN _______

A floating-point divide error has occurred in one of the mathematical functions. The name of the divide routine follows the word IN. The name of the function that called the divide routine appears in the abnormal termination dump.

MATH PACK ERROR -- NEGATIVE ARGUMENT IN _______

The square root or logarithm function has been called with a parameter value less than zero. The name of the function follows the word IN.

MATH PACK ERROR -- OVERFLOW IN LN$

The library function LN has been called with an operand that is too large.

MEMORY PARITY

The program has been terminated because the operating system was interrupted by a memory parity error interrupt. A memory parity error is a hardware malfunction.

NORMAL TERMINATION

The execution of the program has terminated without errors at the end of the main program.

NOT OPENED FOR READING

A READ operation was attempted on a file that has not been opened for reading. Execute a RESET operation on the file before reading, or either a RESET or EXTEND operation if the file is a RANDOM file.

NOT OPENED FOR WRITING

A WRITE operation was attempted on a file that has not been opened for writing. A REWRITE or EXTEND operation must precede a WRITE operation on a file.

OPEN ERROR — ELEMENT SIZE GREATER THAN RECORD LENGTH

The logical record length specified when the file was created is not large enough for the file element size specified in the program. Either create the file again with a logical record length at least as long as the file element size, or delete the file and allow the system to create the file automatically.

OPEN ERROR — STATUS = __

The operating system has returned the status code shown in the message at the return from an attempt to open a file. Refer to the partial list of status codes at the beginning of this paragraph or to the complete list referenced there.

OVERFLOW

An arithmetic operation of INTEGER, FIXED, or DECIMAL type operands has resulted in an overflow. Normally this message is written only if the CKOVER compiler option applied when the code was compiled.

PARAMETER OUT OF RANGE

An I/O utility routine has been called with an invalid argument value. This error should only occur when the user program calls a runtime routine (a routine that has a name in which at least one of the characters is a dollar sign) with an invalid argument. When this error occurs for a system call of a runtime routine, the error is a system error. Contact Texas Instruments software support personnel.

PRECISION CHECK FAILURE

An arithmetic operation of FIXED or DECIMAL type operands has occurred resulting in the loss of one or more most significant digits. This message apears only when the CKPREC compiler option applied when the code was compiled.

PRIVILEGED INSTRUCTION

The program area contains a privileged machine instruction. Pascal programs execute in the nonprivileged mode. This error may be the result of storing data in the program area.

READ ERROR — STATUS = __

A READ operation failed and the operating system returned the status code shown. Refer to the partial list of status codes at the beginning of this paragraph, or to the complete list referenced there.

READ PAST END OF FILE

A READ operation has been attempted on a textfile that is positioned at end-of-file. The program should use function EOF to test for end-of-file. When blank lines precede end-of-file, procedure READLN should be called to read the blank line, followed by a call to function EOLN. When EOLN is false following a READLN operation, call EOF to test for end-of-file.

RUN-TIME COMPATIBILITY CHECK: INDEX

The length of a dynamic array is not compatible with the valid length requirements of the array type. Change the value of the variable that becomes the value of the upper bound.

RUN-TIME COMPATIBILITY CHECK: LONGINT INDEX

The length of a dynamic array which is indexed by an index that is a subrange of LONGINT type is not compatible with the valid length requirements of the array type. Change the value of the variable that becomes the value of the upper bound.

RUN-TIME COMPATIBILITY CHECK: SET BOUND

The size of a dynamic set is not compatible with the valid length requirements of the set type. Change the value of the variable that becomes the value of the upper bound.

RUN-TIME ERROR: INDEX

An array index was set to a value outside the range specified for the array. This message only occurs when the CKINDEX compiler option applied when the code was compiled.

RUN-TIME ERROR: LONGINT CASE ALTERNATIVE

The value of the selector expression of a CASE statement is not equal to any of the case labels; the OTHERWISE clause was omitted; and the type of the selector expression is LONGINT.

RUN-TIME ERROR: LONGINT INDEX

An array index that is a subrange of LONGINT was set to a value outside of the range specified for the array. This message only occurs when the CKINDEX compiler option applied when the code was compiled.

RUN-TIME ERROR: LONGINT SUBRANGE

A variable or field of type subrange of LONGINT was set to a value outside of the subrange. This message only occurs when the CKSUB compiler option applied when the code was compiled.

RUN-TIME ERROR: NIL POINTER

The program has attempted to use a pointer variable that has a value of NIL. This message only occurs when the CKPTR compiler option applied when the code was compiled.

RUN-TIME ERROR: RECORD VARIABLE

A reference to the variant portion of a record is inconsistent with the current value of the tag field. This message only occurs when the CKTAG compiler option applied when the code was compiled.

RUN-TIME ERROR: SCALAR BOUND

A scalar value outside the defined range has been generated. This message only occurs when the CKSUB compiler option applied when the code was generated. Executing function PRED for the smallest value in the range or function SUCC for the largest value in the range causes this error.

RUN-TIME ERROR: SET BOUND

A set element that is not within the range of the base type of the set was specified. This message only occurs when the CKSET compiler option applied when the code was compiled.

RUN-TIME ERROR: SUBRANGE

A variable or field of subrange or scalar type has been assigned a value outside of the subrange. This message only occurs when the CKSUB compiler option applied when the code was compiled.

STACK OVERFLOW

The specified amount of stack space has been used, and the program requests additional stack space. Execute the program again, requesting a larger amount of stack space. Another cause of this error is a recursive routine that continues to call itself indefinitely. Additional stack space only postpones the error in this case; a limit must be placed on the recursion by correcting the program.

STACK USED = ____ HEAP USED = ____

This message follows the normal termination message, indicating the number of bytes of stack and heap space used. Values are decimal numbers.

TASK KILLED

Execution of the task has been manually terminated with an SCI command (Kill Task or Kill Background Task).

TCA ACCESS ERROR

One of the routines that use the SCI Terminal Communications Area has made an unsuccessful attempt to access the Terminal Communications Aea.

TEXT FILE I/O ERROR: ____________ NAME= ________

An error occurred during format conversion of a value being read from or written to a textfile. Either a number or a message may follow the word ERROR. The number has the following significance:

| Error Code | Significance |
|---|---|
| 1 | Parameter out of range |
| 2 | Field width too large |
| 3 | Incomplete data |
| 4 | Invalid character in field |
| 5 | Value too large. |
| 6 | Read past end-of-file |
| 7 | Field exceeds record size |

When a message is printed, it is one of the messages defined in this list. The name of the file follows the word NAME=.

VALUE TOO LARGE

A READ operation of a textfile obtained a value too large to be represented as the type specified. For example, 33000 is too large to be represented as an INTEGER type variable. Either correct the value in the file being read or change the type specification to a type compatible with the file.

WRITE ERROR — STATUS = __

A WRITE operation failed and the operating system returned the status code shown. Refer to the partial list of status codes at the bginning of this paragraph or to the complete list referenced there.

## E.3 RUNTIME ERROR CODES

The error codes listed in this paragraph are placed in the completion code field of the process record (bytes $3C_{16}$ and $3D_{16}$), with the character P replaced by a zero. The three numerals to the right of the P are hexadecimal numerals. The codes as shown are printed in the termination messages and in the optional abnormal termination dump when the minimal runtime code is linked with the task. The recovery procedures for the error messages listed in paragraph E.2 apply to the corresponding error code listed in this paragraph. Most error codes are followed by the corresponding error message in capital letters. Those error codes that do not correspond with an error message are followed by explanatory text.

| | |
|---|---|
| P000 | NORMAL TERMINATION |
| P100 | TASK KILLED (TX990) |
| P101 | MEMORY PARITY |
| P102 | ILLEGAL OPCODE |
| P104 | ILLEGAL SUPERVISOR CALL |
| P105 | ADDRESSING ERROR |
| P106 | PRIVILEGED INSTRUCTION |
| P107 | TASK KILLED (DX10) |
| P201 | STACK OVERFLOW (on entry to a routine) |
| P202 | STACK OVERFLOW (on allocation of a dynamic array) |
| P203 | STACK OVERFLOW (detected by CKTOP$) |
| P300 | HALT CALLED |
| P301 | HEAP FULL |
| P302 | DIVIDE BY ZERO (integer or fixed-point) |
| P303 | FTN ARGUMENT ERROR (wrong number of arguments) |
| P304 | FTN STACK OVERFLOW |
| P305 | DYNAMIC ARRAY SIZE OUT OF RANGE |
| P306 | FTN FLOATING POINT ERROR |
| P365 | INVALID PACKET POINTER |
| P366 | INVALID LENGTH OF PACKET |
| P400 | CANNOT GET MEMORY |
| P401 | Program linked for DX10 was executed memory resident |
| P402 | 2ND ERROR (error during error recovery) |
| P403 | Process exited with a return instead of a resume |
| P421 | TEXT FILE I/O ERROR: PARAMETER OUT OF RANGE |
| P422 | TEXT FILE I/O ERROR: FIELD WIDTH TOO LARGE |
| P423 | TEXT FILE I/O ERROR: INCOMPLETE DATA |
| P424 | TEXT FILE I/O ERROR: INVALID CHARACTER IN FIELD |
| P425 | TEXT FILE I/O ERROR: VALUE TOO LARGE |
| P426 | TEXT FILE I/O ERROR: READ PAST END OF FILE |
| P427 | TEXT FILE I/O ERROR: FIELD EXCEEDS RECORD SIZE |
| P440 | FLOATING POINT ERROR — ILLEGAL INSTRUCTION |
| P441 | FLOATING POINT ERROR — DIVISION BY ZERO |
| P442 | FLOATING POINT ERROR — OVERFLOW |
| P443 | FLOATING POINT ERROR — RELINK FOR EXTENDED PRECISION |
| P460 | MATH PACK ERROR — ARGUMENT TOO LARGE |
| P461 | MATH PACK ERROR — NEGATIVE ARGUMENT |
| P463 | MATH PACK ERROR — WRONG NUMBER OF ARGUMENTS |
| P464 | MATH PACK ERROR — OVERFLOW |
| P465 | MATH PACK ERROR — DIVISION BY ZERO |
| P481 | ATTEMPT TO READ PAST EOF |
| P482 | ATTEMPT TO SKIP BEYOND EOM |
| P483 | NOT OPENED FOR READING |

| | |
|---|---|
| P4A1 | RUN-TIME COMPATIBILITY CHECK: INDEX |
| P4A2 | RUN-TIME COMPATIBILITY CHECK: LONGINT INDEX |
| P4A3 | RUN-TIME COMPATIBILITY CHECK: SET BOUND |
| P4B1 | RUN-TIME ERROR: INDEX |
| P4B2 | RUN-TIME ERROR: LONGINT INDEX |
| P4B3 | RUN-TIME ERROR: SUBRANGE |
| P4B4 | RUN-TIME ERROR: LONGINT SUBRANGE |
| P4B5 | RUN-TIME ERROR: SCALAR BOUND |
| P4B6 | RUN-TIME ERROR: LONGINT SCALAR BOUND |
| P4B8 | RUN-TIME ERROR: LONGINT CASE ALTERNATIVE |
| P4B9 | RUN-TIME ERROR: SET BOUND |
| P4BA | RUN-TIME ERROR: NIL POINTER |
| P4BB | RUN-TIME ERROR: RECORD VARIABLE |
| P4C0 | "CASE" ALTERNATIVE ERROR |
| P4C3 | OVERFLOW |
| P4C7 | PRECISION CHECK FAILURE |
| P4CA | "ASSERT" FAILED |
| P4CF | FATAL COMPILATION ERRORS |
| P500 | OPEN ERROR — ELEMENT SIZE > RECORD LENGTH (on a RESET) |
| P5xx | OPEN ERROR — STATUS = xx (on a RESET) |
| P600 | OPEN ERROR — ELEMENT SIZE > RECORD LENGTH (REWRITE or EXTEND) |
| P6xx | OPEN ERROR — STATUS = xx (REWRITE or EXTEND) |
| P7xx | READ ERROR — STATUS = xx |
| P800 | NOT OPENED FOR WRITING |
| P8xx | WRITE ERROR — STATUS = xx |
| P9xx | CANNOT LOAD OVERLAY (System error code = xx) |
| P9FF | INVALID OVERLAY NUMBER |

## E.4 DEBUG COMMAND ERROR MESSAGES

The Pascal debug commands of SCI write the following error messages:

D01F — TOO MANY PROCESS RECORDS.

The Pascal task consists of more than 100 processes. Either the task is not a Pascal task or task memory has been altered.

D020 — INVALID BREAKPOINT LOCATION.

The string entered in response to the WHERE prompt is unrecognizable.

D021 — INVALID STACK FRAME.

Either the address of the bottom of the stack frame is not less than the address of the top of the stack frame or the address of the top of the stack frame is not equal to the address of the bottom of the next stack frame. This message results if an SPS or LPS command is issued before the task has been initialized (immediately following the execution of XPT specifying the debug mode). To recover from this condition, assign a breakpoint and enter an RT command, then enter the SPS or LPS command. Other causes of this error are when a Pascal debug command is entered for a task that is not a Pascal task, or when task memory has been altered.

D022 — INVALID PROCESS RECORD.

Either the process record cannot be found or the structure of the process record is incorrect. Either the task is not a Pascal task or task memory has been altered.

D023 — INVALID ROUTINE NAME.

The routine name entered in the command cannot be found. Either the routine does not exist or the routine was compiled with the Traceback option off.

D024 — NONSTANDARD OR FORTRAN FRAME.

The stack frame at the top of the stack is for a nonstandard or FORTRAN routine. The Pascal debug commands require the stack frame to be that of a Pascal routine or an assembly language routine with standard interface.

# APPENDIX F
# ASSEMBLY LANGUAGE ROUTINES

# APPENDIX F

# ASSEMBLY LANGUAGE ROUTINES

## F.1 GENERAL

A user may write a procedure or function in assembly language. The procedure or function must comply with the following requirements:

- Format the routine in a manner compatible with the TIP compiler and runtime system.
- Specify the static nesting level of the routine.
- Call the appropriate entry handler for the routine.
- Declare the routine as external in the appropriate declaration section of the calling program.

The format of the routine involves both the format of the data for the routine in the stack frame of the routine and the format of the routine itself. The format also varies depending on the category of the routine. The runtime system categorizes routines in three categories according to the degree of complexity of the routine. The runtime system includes a separate handler for each category. The more complex routines require a more complex handler; less complex routines may use a simpler handler that requires less time to execute.

This appendix describes the static nesting level and its relation to the routine, and the categories of routines. The appendix then describes the stack frame and finally, the routine itself.

In some applications, the user may provide routines for program error termination. This appendix includes descriptions of termination routines in the TIP libraries to illustrate the interface with these routines.

## F.2 STATIC NESTING LEVEL

A TIP program consists of a main program that may declare one or more routines. Each routine may declare one or more routines. The main program is the static nesting level of one. Any or all routines declared in the main program are at static nesting level two. Any or all routines declared in a level two routine are at static nesting level three. The general rule is that the static nesting level of a routine is one greater than that of the routine in which the routine is declared. The runtime support system supports 16 static nesting levels. The static nesting level must be explicitly identified within a routine in order for the runtime system to properly interface with the routine.

## F.3 ROUTINE CATEGORIES

A routine that neither declares nor calls another routine and that requires no more than 64 bytes of stack space is in the short category. A routine that does not declare another routine and that either calls one or more routines or requires more than 64 bytes of stack space is in the medium category. A routine that declares one or more routines is in the standard category.

A routine written in assembly language does not actually declare a routine; i.e., there is no PROCEDURE or FUNCTION declaration in assembly language. The equivalent of declaring a routine in assembly language is calling a routine that has a static level number greater than that of the calling routine.

The category of the routine determines the uses of registers and the entry handler that the routine calls.

## F.4 THE STACK FRAME

The stack frame contains the workspace for the routine, pointers stored by the entry handler, the result (for routines that are functions), the parameters, local variables, and compiler-generated temporary data. Figure F-1 shows the structure of the stack frame.

**F.4.1 WORKSPACE.** The workspace is the set of workspace registers (R0 through R15) used by the routine. The entry to the routine is by a BLWP instruction which places the lowest address in the stack frame in the WP register, making the first 32 bytes of the stack frame the workspace.

For standard category routines only registers R9 and R10 are dedicated (i.e., the contents of these registers must not be altered). Register R9 contains the address of the bottom of the stack frame (i.e., the lowest address of the stack frame of the routine). Register R10 contains the address of the top of the stack (i.e., the word beyond the word at the highest address in the stack frame of the routine). Other registers may be used as desired. Register R12 contains the address of the process record.

For medium category routines, in addition to the registers used for standard routines, registers R13 and R14 are dedicated. Register R13 contains the calling routine's workspace pointer and register R14 contains the return address in the calling routine. These uses of R13 and R14 are those uses made by the computer hardware; the entry handler does not store the values for restoration by the return handler.

For short category routines register R15 is dedicated, in addition to the registers used for medium and standard category routines. Register R15 contains the contents of the status register of the computer at the time of the call to the routine.

**F.4.2 SYSTEM STORAGE.** The four word area following the workspace in the stack frame is used by the entry handler for storage of return information. The use of the words differs for each category of routine.

Before discussing the use of this area by the handler, the display pointer must be defined. The runtime system maintains an array labelled DISPLAY in the process record. The array contains the address of the stack frame of static nesting level one in the first element (address 2 relative to the process record address in R12). Successive elements of the array contain the addresses of the stack frames of the current routines at successive nesting levels. That is, element two of the array contains the stack frame address of the most recently called routine at static nesting level two, element three contains such an address for level three, etc. The addresses in the DISPLAY array are called display entries. The display entries enable a routine to access the variables in all currently active stack frames.

Array DISPLAY contains pointers to stack frames of the most recently called routines at the static nesting levels. However, routines may call other routines at the same level, and also may call themselves (recursion). When a routine calls itself or another routine at the same level, dynamic nesting results. The dynamic nesting relationships are implied in the order of stack frames in the stack and in the information stored in the stack frames.

The entry handler for a standard category routine stores the display entry in the first word of the area of the stack frame that follows the workspace (address $20_{16}$ relative to the address of the stack frame). The entry handler stores the WP register of the calling routine in the following word (address $22_{16}$), and the return address in the next word (address $24_{16}$). The last word in this area, address $26_{16}$ relative to the stack frame address, is reserved for tasks that use SCI I/O. Tasks that use LUNO I/O place the current process record address in address $26_{16}$.

The entry handler for a medium category routine uses only the word at address $24_{16}$. The handler stores -1 in that word, identifying the stack frame as that of a medium category routine. The return information remains in registers R13 and R14, which are not available to the routine.

| RELATIVE ADDRESS (HEXADECIMAL) | 0 | 2 | 4 | 6 | 8 | A | C | E |
|---|---|---|---|---|---|---|---|---|
| 0 | R0 | R1 | R2 | R3 | R4 | R5 | R6 | R7 |
| 10 | R8 | R9 | R10 | R11 | R12 | R13 | R14 | R15 |
| 20 | SAVED | CALL-ER | RET-ADR | PRP | RESULT/ARGS/ LOCALS/TEMPS | | | |

| | | |
|---|---|---|
| 0 – 11 | R0 – R8 | WORKSPACE REGISTERS AVAILABLE TO ROUTINE. |
| 12 – 13 | R9 | CONTAINS ADDRESS OF STACK FRAME (BOTTOM). |
| 14 – 15 | R10 | CONTAINS ADDRESS OF NEXT WORD BEYOND STACK FRAME (TOP) |
| 16 – 17 | R11 | WORKSPACE REGISTER, LINK TO OUT-OF-LINE ROUTINES. |
| 18 – 19 | R12 | CONTAINS ADDRESS OF PROCESS RECORD. |
| 1A – 1B | R13 | FOR STANDARD CATEGORY, AVAILABLE TO ROUTINE.<br>FOR OTHER CATEGORIES, CALLING ROUTINE WP. |
| 1C – 1D | R14 | FOR STANDARD CATEGORY, AVAILABLE TO ROUTINE.<br>FOR OTHER CATEGORIES, RETURN ADDRESS. |
| 1E – 1F | R15 | FOR STANDARD AND MEDIUM CATEGORIES, AVAILABLE TO ROUTINE.<br>FOR SHORT CATEGORY, STATUS REGISTER CONTENTS. |
| 20 – 21 | SAVED | FOR STANDARD CATEGORY, PREVIOUS DISPLAY ENTRY.<br>FOR OTHER CATEGORIES, RESERVED. |
| 22 – 23 | CALLER | FOR STANDARD CATEGORY, CALLING ROUTINE WP.<br>FOR OTHER CATEGORIES, RESERVED. |
| 24 – 25 | RETADR | FOR STANDARD CATEGORY, CALLING ROUTINE PC<br>FOR MEDIUM CATEGORY, $FFFF_{16}$ (−1).<br>FOR SHORT CATEGORY, 0000. |
| 26 – 27 | PRP | PROCESS RECORD POINTER – CONTAINS CURRENT PROCESS ADDRESS (LUNO I/O). NOT USED WITH SCI I/O. |
| 28 – 2F | RESULT | THE RESULT OF A FUNCTION (TYPE OTHER THAN DECIMAL). |
| 28 – 31 | RESULT | THE RESULT OF A FUNCTION (TYPE DECIMAL). |
| | ARGS | THE PARAMETERS (ARGUMENTS) OF THE ROUTINE. |
| | LOCALS | LOCAL VARIABLES OF THE ROUTINE. |
| | TEMPS | TEMPORARY VARIABLES. |

(A)138381

**Figure F-1. Stack Frame Structure**

The entry handler for a short category routine uses only the word at address $24_{16}$. The handler stores zero in that word, identifying the stack frame as that of a short category routine. The return information remains in registers R13, R14, and R15, which are not available to the routine.

**F.4.3 DATA.** The remainder of the stack frame is of variable length and contains the data required by the routine. The first area applies only to functions: the area into which the routine must place the result. When the type of the function result in DECIMAL, the area is ten bytes in length. Results of other types must be left justified in an eight byte field.

The second area for functions (the first area for procedures) is the area for parameters or arguments. Parameters are passed by value or by reference in TIP. When a parameter is passed by value, a copy of the parameter is placed in the stack frame immediately prior to the call of the routine. The parameter occupies a variable amount of space determined by the type of the parameter. When a parameter is passed by reference the address of the parameter is placed in the stack. A parameter passed by reference requires one word in the stack regardless of the type of the parameter.

Two types of parameters require special additional information: parameters with dynamic bounds and procedure or function parameters. When a parameter with one or more dynamic bounds is passed by reference, the address of the parameter (array or set), the upper bound (or bounds, if an array has more than one dimension with a dynamic upper bound), and the size are placed in the stack frame. When a parameter with dynamic bounds is passed by value, the same information is placed in the stack frame, and a copy of the parameter is placed in the dynamic portion of the stack frame. This requires that the called routine extend the stack frame, copy the parmeter into the extended stack frame, and change the address in the stack frame to point to the copy.

An example of an assembly language procedure having a parameter with a dynamic bound is shown in figure F-2. This procedure, P$PARM, is a level 2 routine that returns a specified SCI parameter in array STRING. The length of array STRING is determined by the length of the corresponding parameter in the procedure call. The procedure calls SCI routine S$PARM (described in the *Model 990 Computer DX10 Operating System Release 3 Reference Manual, Volume V*) to obtain the parameter. The declaration for the procedure is shown in the first six comment lines, except that the keyword EXTERNAL must follow the parameter list.

The routine uses EQU directives to define displacements for the parameters. Note that the first parameter, I, is in the first word of the parameter list of the stack frame. The next word contains the upper bound of array STRING, followed by the size of the array, and the address of the array. The last word of the parameter list of the stack frame contains the address of the third parameter, ERROR.

When a parameter is a procedure or function, the parameter is represented in the stack frame by a structure that contains the entry point address of the procedure and a copy of the set of display pointer values that provides the correct environment for the procedure or function. The environment of the function or procedure passed as a parameter may be quite different from that of the called routine. The TIP runtime system uses the environment of the parameter routine rather than that of the called routine. This requires that the DISPLAY array containing the environment of the parameter routine be copied into the stack frame. When the parameter routine is executed, the software saves the contents of the DISPLAY array, and places the copy from the stack frame into the array. Upon return from the parameter routine, the saved contents of the DISPLAY array are restored.

The parameters in the stack frame are followed by the local variables for the routine. This area is of variable length, determined by the number of variables and the type of each variable.

```
        IDT 'P$PARM'
* PROCEDURE P$PARM
* (    I: INTEGER; { Sequence number of desired parameter }
*      VAR STRING: { Buffer to hold returned parameter }
*          PACKED ARRAY [0..?] OF CHAR;
*      VAR ERROR: INTEGER { Error indicator: nonzero implies error }
* )
* This routine provides a TIP interface to the SCI990 routine S$PARM.
*    The Ith parameter is fetched from the TCA file if
*      (1) it exists, and
*      (2) STRING is large enough to contain it.
*    The length of the parameter is returned in STRING[0];
*    STRING[0]=0 implies that the Ith parameter does not exist.
*    ERROR <> 0 implies that an SCI-detected error has occurred;
*    ERROR is the error code returned by SCI.
*    (ERROR = >901B implies STRING too small to hold result.)
*    Note: STRING need not be preinitialized in any way.
*
* Register Equates:
BOT     EQU R9              Start of stack frame
ARG     EQU >28             Beginning of argument list
*
I       EQU ARG+0           Value of I
UB      EQU ARG+2           Upper bound of STRING
SIZE    EQU ARG+4           Total length of STRING
STRING  EQU ARG+6           Address of STRING
ERROR   EQU ARG+8           Address of ERROR
*
        REF ENT$S,RET$S
        REF S$PARM
*
L1      TEXT 'P$PARM  '
        DATA 2              Level Number
        DATA L2             Epilogue Code
        DATA L1             Literals Area
        DEF  P$PARM
P$PARM  BL   @ENT$S
        DATA >32            Stack Frame Size
*                                   Set up arguments:
        MOV  @I(BOT),R1               Parameter number
        MOV  @STRING(BOT),R2          STRING address
        MOVB @UB+1(BOT),*R2           STRING Length (first byte of STRING)
        BLWP @S$PARM                Call S$PARM
        BYTE R1,R2                    (specify argument locations)
        MOV  @ERROR(BOT),R3         Get error code
        MOV  R0,*R3
L2      B    @RET$S                 Return to caller
        END
```

Figure F-2. Assembly Language Routine Example

When compiling routines written in TIP, the compiler places temporary variables in the stack frame following the local variables. Temporary variables are variables in which the routine stores intermediate results, or similar data. When writing a routine in assembler language the programmer may allow space in the stack frame for temporary variables by increasing the stack frame length appropriately and accessing the additional words as required.

## F.5 THE ROUTINE MODULE

The module for the routine contains required constant information, constants and literals required by the routine, and the executable code for the routine. Figure F-3 shows the structure of the module.

The first section of the code for the module contains constants. The first statement must be a TEXT directive to place the module name in the first four words of the module. Fill the operand to the right with blanks when the module name contains fewer than eight characters. The next statement must be a DATA directive to place the static nesting level in the next word.

These constants required by the runtime support system are followed by constants and literals required for the routine. The directives required to define these items, if any, are followed by two more DATA directives. The operand of the first data directive is the label of the epilogue of the routine, described in a subsequent paragraph. The operand of the second is either the label of the TEXT directive that contains the module name, or zero, as described in the next paragraph.

The constants required by the runtime support system are used in the event of an escape from the module. When a routine does not require any constants and an escape from the module cannot occur, all of the constants previously described may be omitted except for a single DATA directive with an operand of zero, indicating that the constants section is empty.

```
literals TEXT  'ROUTINE'           8 CHARACTER MODULE NAME
         DATA  n                   STATIC NESTING LEVEL
         .
         .    constants and literals
         .
         DATA  epilogue            (ENTRY POINT - 4)
         DATA  literals            (ENTRY POINT - 2)
ROUTINE  EQU   $                   ROUTINE ENTRY POINT
         BL    @ENT$n              BRANCH TO ENTRY HANDLER
         DATA  frame size          SIZE OF STACK FRAME REQUIRED
         .
         .   routine body
         .
epilogue EQU   $
         .
         .   epilogue code
         .
         B      @RET$n             BRANCH TO RETURN HANDLER
```

**Figure F-3. Structure of a Standard Category Routine at Level n**

The entry point of the routine follows the constants section and must be a BL instruction to branch to the appropriate entry handler. The branch to the entry handler for a short category routine is BL @ENT$S. The branch to the entry handler for a medium category routine is BL @ENT$M. The branch to the entry handler for a standard category routine is BL @ENT$n, where n is the static nesting level of the routine. The BL instruction is followed by a DATA directive with the frame size as the operand.

The executable code of the routine follows the DATA directive. Following the code for the routine is the epilogue, which performs any required termination functions and calls the return handler. There is a return handler for each category, corresponding to the entry handler. For a short category routine, the branch to the return handler is B @RET$S. For a medium category routine, the branch to the return handler is B @RET$M. For a standard category routine, the branch to the return handler is B @RET$n, where n is the static nesting level of the routine.

**F.5.1 ACCESS TO VARIABLES.** A TIP routine has access to its local variables, to variables declared in the routine in which that routine was declared, to those declared in the routine in which that routine was declared, and to those of routines active at each level back to the global variables declared in the main program. Access to parameters passed by value and to local variables at each level involves accessing the variable from the stack frame. Access to parameters passed by reference involves accessing the address in the stack frame and accessing the variable at that address.

To access a parameter passed by value or a local variable, use indexed addressing. R9 contains the stack frame address. The following is an example of code to access a value parameter or a local variable:

```
NUM     EQU    >28                  DISPLACEMENT OF PARAMETER
        .
        .
        .
        MOV    @NUM(R9),R0          MOVE PARAMETER TO R0
```

The DISPLAY array in the process record contains the stack frame addresses of all active stack frames. R12 contains the address of the process record upon return from the entry handler. If R12 has been used for other storage, the address of the process record may be restored in R12 as follows:

```
REF     CUR$$                       REFERENCE ROUTINE
 .
 .
 .
BL      @CUR$$                      CALL ROUTINE
```

To obtain the address of the stack frame for a particular level, compute the displacement by multiplying the level number by two. Then use the displacement within the stack frame for a particular variable to access the variable. The following is an example:

```
INT     EQU    >30                  DISPLACEMENT OF VARIABLE INT
*                                   IN LEVEL 2 STACK FRAME
DLEV2   EQU    2*2                  DISPLACEMENT FOR LEVEL 2
*                                   DISPLAY POINTER
        .
        .
        .
```

```
          MOV    @DLEV2(R12),R2       MOVE LEVEL 2 STACK FRAME
*                                     ADDRESS INTO R2
          MOV    @INT(R2),R3          MOVE INT TO R3
```

Accessing parameters passed by reference requires accessing the address, then accessing the parameter at the address. The following is an example of accessing a reference parameter:

```
NUM       EQU    >28                  DISPLACEMENT OF PARAMETER
          .
          .
          .
          MOV    @NUM(R9),R2          MOVE PARAMETER ADDRESS INTO
*                                     R2
          MOV    *R2,R0               MOVE PARAMETER TO R0
```

Accessing a parameter passed by reference to a routine at another level requires accessing the address from the stack frame at that level, then accessing the parameter at the address. The following is an example:

```
INT       EQU    >30                  DISPLACEMENT OF PARAMETER
*                                     INT IN LEVEL 2 STACK FRAME
DLEV2     EQU    2*2                  DISPLACEMENT FOR LEVEL 2
*                                     DISPLAY POINTER
          .
          .
          .
          MOV    @DLEV2(R12),R2       MOVE LEVEL 2 STACK FRAME
*                                     ADDRESS INTO R2
          MOV    @INT(R2),R4          MOVE PARAMETER ADDRESS INTO
*                                     R4
          MOV    *R4,R3               MOVE INT TO R3
```

**F.5.2 CALLING A ROUTINE.** Calling a routine requires making parameters, if any, available to the called routine, and transferring control to the entry point of the called routine. Copies of value parameters must be moved into the stack frame of the called routine. Addresses of reference parameters must be placed in the stack frame of the called routine. Register R10 contains the address of the first word beyond the stack frame of a routine, which is to become the address of the stack frame of the called routine. Copies of variables or addresses of parameters may be moved into the stack frame of a called routine by using the displacement for the variable indexed by R10. The following is an example of moving a variable of a routine into the stack frame of the called routine:

```
VAL       EQU    >28                  DISPLACEMENT OF LOCAL
*                                     VARIABLE VAL
NUM       EQU    >2A                  DISPLACEMENT OF VALUE
*                                     PARAMETER FOR CALLED ROUTINE
          .
          .
          .
          MOV    @VAL(R9),@NUM(R10)   MOVE COPY TO NEW STACK FRAME
```

The following is an example of moving a parameter address from a stack frame at a higher level to the stack frame of the called routine:

```
INT     EQU   >30                   DISPLACEMENT OF PARAMETER
*                                   INT IN LEVEL 2 STACK FRAME
DLEV2   EQU   2*2                   DISPLACEMENT FOR LEVEL 2
*                                   DISPLAY POINTER
VAL     EQU   >28                   DISPLACEMENT FOR PARAMETER
*                                   IN CALLED ROUTINE STACK FRAME
        .
        .
        MOV   @DLEV2(R12),R2        MOVE LEVEL 2 STACK FRAME
*                                   ADDRESS INTO R2
        MOV   @INT(R2),@VAL(R10)    MOVE PARAMETER ADDRESS TO
*                                   CALLED ROUTINE STACK FRAME
```

Transfer of control to a routine is performed by a BLWP instruction. The transfer vector must be created at runtime because the stack frame address is determined at runtime. Register R10 contains the stack frame address for the called routine, which is also the workspace address. The code to call a routine is as follows:

```
TOP     EQU   R10                   POINTER TO STACK FRAME OF
*                                   CALLED ROUTINE
        .
        .
        .
        LI    TOP+1,ROUTINE         PLACE ENTRY POINT OF ROUTINE
*                                   IN R11
        BLWP  TOP                   BRANCH TO ROUTINE
```

## F.6 AN ALTERNATIVE METHOD

When it is necessary to write a routine in assembly language, the user may write a dummy routine in TIP and compile the program. The dummy routine should declare the routine precisely so that the stack frame is built by the compiler to provide for the parameters and variables required. The statement portion of the routine may approximate the processing required for the routine, or may consist only of the BEGIN and END keywords required. The object module produced by the compiler may be submitted to the Reverse Assembler to list the assembly language corresponding to the object code. The interface with the runtime system is thus produced by the compiler, and the user need only add the assembly language code to perform the processing. The preceding paragraphs of this appendix help the user to identify the interface and the point at which to add executable code.

## F.7 USER TERMINATION ROUTINES

The user may write routines to perform error termination. The following paragraphs describe the TIP library routines and show examples of the code. The termination sequence provided by the standard library (.TIP.OBJ) is described in the next paragraph. Differences that exist when other libraries are used are described in subsequent paragraphs.

**F.7.1 STANDARD TERMINATION ROUTINES.** When a TIP task terminates, control passes to library routine TERM$. TERM$ is a Pascal callable routine with the address of the process record for the terminating task as its parameter. TERM$ is part of the initiate/terminate process, a separate process from the process for the user task. The parameter, the address of the task's process record, allows TERM$ to access the contents of the WP, PC and ST registers in bytes $22_{16}$ through $27_{16}$ of the process record. TERM$ can also access the termination code, bytes $3C_{16}$ and $3D_{16}$ of the process record.

TERM$ writes a message that specifies the reason for termination and calls the abnormal termination dump routines when appropriate. It displays the amount of stack and heap used and then calls P$TERM, which actually terminates execution.

Routine P$TERM is a Pascal callable procedure without parameters. The following partial listing of P$TERM illustrates how it accesses the termination code:

```
        IDT    'P$TERM'
LO      TEXT   'P$TERM '
        DATA   2
        REF    ENT$S
        REF    S$STOP
        REF    T$CC,T$EC,T$MSG
        DATA   EPI,LO
        DEF    P$TERM
P$TERM  BL     @ENT$S              R12 := PROCESS RECORD ADDRESS
        DATA   40
*
*              ---TEST FOR NORMAL OR ABNORMAL COMPLETION---
*
        MOV    @>3A(R12),R8        R8 := TASK INFO. BLOCK ADDRESS
        MOV    @T$CC(R8),R1        R1 := COMPLETION CODE
        MOV    @T$EC(R8),R3        R3 := ERROR CODE
        JEQ    L1                  SKIP IF NORMAL TERMINATION
        ORI    R1,>C000            SET $$CC TO INDICATE ABORT
L1      EQU    $
*
*              ---  SET $$CC AND TERMINATE  ---
*
        MOV    @T$MSG(R8),R2       MESSAGE ADDRESS (CURRENTLY 0)
EPI     BLWP   @S$STOP             TERMINATE ($$CC IN R1)
        END
```

Routine S$STOP, which P$TERM calls with the value for $$CC in R1, sets the completion code synonym $$CC to the value in R1 and stops.

**F.7.2 LUNO I/O TERMINATION ROUTINES.** When a TIP task that is linked for LUNO I/O terminates, control passes to library routine TERM$ as described in the preceding paragraph. The TERM$ routine is similar to the one previously described. It calls P$TERM to terminate execution. The version of P$TERM in library .TIP.LUNOBJ is much simpler than the version in library .TIP.OBJ; it only executes an end-of-task supervisor call.

**F.7.3 MINIMAL RUNTIME TERMINATION ROUTINES.** When a TIP task that is linked for minimal runtime terminates, control passes to library routine TERM$$. TERM$$ is an assembly language routine that is entered with the address of the process record for the terminating task in R12. R13 contains the address of a workspace in which R13, R14, and R15 contain the WP, PC, and ST values applicable to an error termination. The contents of these workspace registers are not significant for a normal termination. The process record contains the termination code in bytes $3C_{16}$ and $3D_{16}$.

After writing the termination message, TERM$$ calls P$STOP to terminate the task. The following is the listing of P$STOP:

```
            IDT     'P$STOP'
            DEF     P$STOP
    P$STOP  LI      R1,>400         SET OP CODE FOR END OF TASK CALL
            XOP     R1,15                   SUPERVISOR CALL
            END
```

For tasks linked with TX990, the code produced by the expansion of the macro instructions in the configuration module (paragraph 14.3.7) includes the code for performing termination of tasks. This code begins at label ABND$0. When the dump task is included, the code bids the dump task. Otherwise, the code calls routine TXTRM$ to write the termination message. The code passes the process record address to TXTRM$ in workspace register R12.

# APPENDIX G
# ASCII CHARACTER SET

# APPENDIX G

# ASCII CHARACTER SET

The internal representation of the TIP character set is the ASCII code. Table G-1 lists the complete ASCII character set with the hexadecimal and decimal equivalents of the representation of each character. The first 32 characters (represented by $00_{16}$ through $1F_{16}$) are control characters. These characters are included because they may be specified in strings by the hexadecimal representation preceded by a pound sign (#).

Table G-1. ASCII Character Set

| Hexadecimal Value | Decimal Value | Character | Hexadecimal Value | Decimal Value | Character |
|---|---|---|---|---|---|
| 00 | 00 | NUL | 23 | 35 | # |
| 01 | 01 | SOH | 24 | 36 | $ |
| 02 | 02 | STX | 25 | 37 | % |
| 03 | 03 | ETX | 26 | 38 | & |
| 04 | 04 | EOT | 27 | 39 | ' |
| 05 | 05 | ENQ | 28 | 40 | ( |
| 06 | 06 | ACK | 29 | 41 | ) |
| 07 | 07 | BEL | 2A | 42 | * |
| 08 | 08 | BS | 2B | 43 | + |
| 09 | 09 | HT | 2C | 44 | , |
| 0A | 10 | LF | 2D | 45 | - |
| 0B | 11 | VT | 2E | 46 | . |
| 0C | 12 | FF | 2F | 47 | / |
| 0D | 13 | CR | 30 | 48 | 0 |
| 0E | 14 | SO | 31 | 49 | 1 |
| 0F | 15 | SI | 32 | 50 | 2 |
| 10 | 16 | DLE | 33 | 51 | 3 |
| 11 | 17 | DC1 | 34 | 52 | 4 |
| 12 | 18 | DC2 | 35 | 53 | 5 |
| 13 | 19 | DC3 | 36 | 54 | 6 |
| 14 | 20 | DC4 | 37 | 55 | 7 |
| 15 | 21 | NAK | 38 | 56 | 8 |
| 16 | 22 | SYN | 39 | 57 | 9 |
| 17 | 23 | ETB | 3A | 58 | : |
| 18 | 24 | CAN | 3B | 59 | ; |
| 19 | 25 | EM | 3C | 60 | < |
| 1A | 26 | SUB | 3D | 61 | = |
| 1B | 27 | ESC | 3E | 62 | > |
| 1C | 28 | FS | 3F | 63 | ? |
| 1D | 29 | GS | 40 | 64 | @ |
| 1E | 30 | RS | 41 | 65 | A |
| 1F | 31 | US | 42 | 66 | B |
| 20 | 32 | SP | 43 | 67 | C |
| 21 | 33 | ! | 44 | 68 | D |
| 22 | 34 | " | 45 | 69 | E |

Table G-1. ASCII Character Set (Continued)

| Hexadecimal Value | Decimal Value | Character | Hexadecimal Value | Decimal Value | Character |
|---|---|---|---|---|---|
| 46 | 70 | F | 63 | 99 | c |
| 47 | 71 | G | 64 | 100 | d |
| 48 | 72 | H | 65 | 100 | e |
| 49 | 73 | I | 66 | 101 | f |
| 4A | 74 | J | 67 | 102 | g |
| 4B | 75 | K | 68 | 103 | h |
| 4C | 76 | L | 69 | 104 | i |
| 4D | 77 | M | 6A | 106 | j |
| 4E | 78 | N | 6B | 107 | k |
| 4F | 79 | O | 6C | 108 | l |
| 50 | 80 | P | 6D | 109 | m |
| 51 | 81 | Q | 6E | 110 | n |
| 52 | 82 | R | 6F | 111 | o |
| 53 | 83 | S | 70 | 112 | p |
| 54 | 84 | T | 71 | 113 | q |
| 55 | 85 | U | 72 | 114 | r |
| 56 | 86 | V | 73 | 115 | s |
| 57 | 87 | W | 74 | 116 | t |
| 58 | 88 | X | 75 | 117 | u |
| 59 | 89 | Y | 76 | 118 | v |
| 5A | 90 | Z | 77 | 119 | w |
| 5B | 91 | [ | 78 | 120 | x |
| 5C | 92 | \ | 79 | 121 | y |
| 5D | 93 | ] | 7A | 122 | z |
| 5E | 94 | Λ | 7B | 123 | { |
| 5F | 95 | — | 7C | 124 | \| |
| 60 | 96 | ` | 7D | 125 | } |
| 61 | 97 | a | 7E | 126 | ∿ |
| 62 | 98 | b | 7F | 127 | DEL |

# APPENDIX H
# OVERLAYS IN TIP PROGRAMS

# APPENDIX H

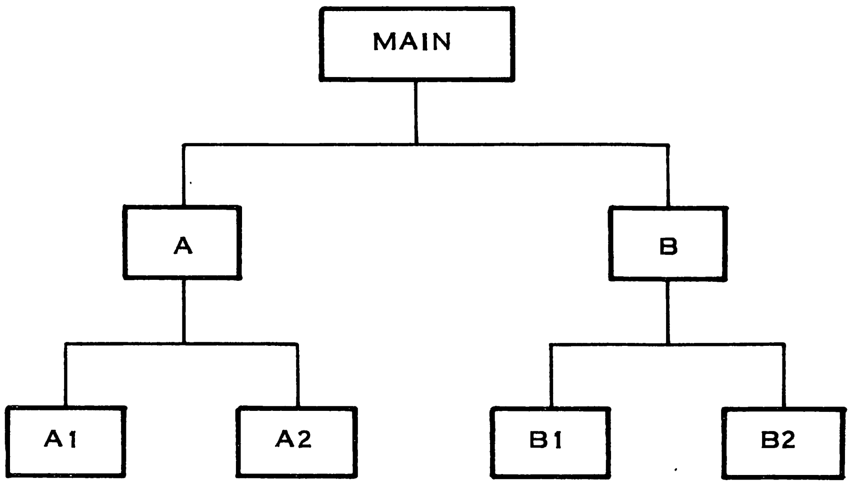
# OVERLAYS IN TIP PROGRAMS

## H.1 GENERAL

TIP Software supports defining one or more modules of a TIP object program as an overlay or overlays and loading overlays appropriately during the execution of the program. Specifically, the software includes a procedure for loading an overlay. The object code must be placed in a library of object modules as is done by CONFIG or SPLIT. The modules must be linked in the desired overlay structure. This appendix describes the organizing of a program into overlays, the use of the overlay procedure OVLY$, the control file for linking the modules into the desired structure, and the System Command Interpreter (SCI) commands to install the program.

## H.2 OVERLAY STRUCTURE

To illustrate the overlay structure that a TIP program could have, the following tree structure shows a program MAIN with procedures A, A1, A2, B, B1, and B2:



Program MAIN calls procedure A or procedure B; procedure A calls procedure A1 or procedure A2; and procedure B calls procedure B1 or procedure B2. The declarations for the TIP program corresponding to this structure are:

```
PROGRAM MAIN;
VAR PATH:INTEGER;
PROCEDURE A;
PROCEDURE A1;
BEGIN                (* A1 *)
END;                 (* A1 *)


PROCEDURE A2:
BEGIN                (* A2 *)
END;                 (* A2 *)
```

```
BEGIN                    (* A *)
   CASE PATH OF          (* KIND OF PATH *)
        1: A1;
        2: A2;
        END;
END;                     (* A *)
PROCEDURE B;
PROCEDURE B1;
BEGIN                    (* B1 *)
END;                     (* B1 *)
PROCEDURE B2;
BEGIN                    (* B2 *)
END;                     (* B2 *)
BEGIN                    (* B *)
   CASE PATH OF          (* KIND OF PATH *)
        3: B1;
        4: B2;
        END;
END;                     (* B *)
BEGIN                    (* MAIN *)
   CASE PATH OF
        1,2: A;
        3,4: B;
        END;
END.                     (* MAIN *)
```

This structure can be implemented with overlays, with program MAIN as the root, modules A and B as overlays, modules A1 and A2 as overlays called in module A and modules B1 and B2 as overlays called in module B. The modules are assigned the following overlay numbers:

A - overlay 1
B - overlay 2
A1 - overlay 3
A2 - overlay 4
B1 - overlay 5
B2 - overlay 6

This example illustrates a program having an overlay structure identical to the calling structure. However, the TIP software does not require that the overlay structure coincide with the calling structure. For example, procedures B1 and B2 do not need to be nested within procedure B, but could be declared following the declaration of procedure A.

## H.3 PROCEDURE OVLY$

The overlay handler provided in the TIP software is procedure OVLY$. OVLY$ is called in the module that calls the overlay prior to the call of the procedure in the overlay. The procedure must be declared in the main program, as follows:

```
PROCEDURE OVLY$(LOAD:INTEGER); EXTERNAL;
```

The handler must be initialized by the following call:

```
OVLY$(0);
```

This call must precede any call to OVLY$ to load an overlay. The argument 0 causes the procedure to initialize the handler. The argument is the overlay number in subsequent calls to OVLY$ to load an overlay. The code for the example program including appropriate calls to OVLY$ is as follows:

```
PROGRAM MAIN;
VAR PATH:INTEGER;
PROCEDURE OVLY$(LOAD:INTEGER); EXTERNAL;
PROCEDURE A;
PROCEDURE A1;
BEGIN                        (* A1 *)
END;                         (* A1 *)
PROCEDURE A2;
BEGIN                        (* A2 *)
END;                         (* A2 *)
BEGIN                        (* A *)
   CASE PATH OF              (* KIND OF PATH *)
        1: BEGIN
             OVLY$(3); A1; END;
        2: BEGIN
             OVLY$(4); A2; END;
        END;
END;
                             (* A *)
PROCEDURE B;
PROCEDURE B1;
BEGIN                        (* B1 *)
END;                         (* B1 *)
PROCEDURE B2;
BEGIN                        (* B2 *)
END;                         (* B2 *)
BEGIN                        (* B *)
   CASE PATH OF              (* KIND OF PATH *)
        3: BEGIN
             OVLY$(5); B1; END;
        4: BEGIN
             OVLY$(6); B2; END;
        END;
END;                         (* B *)
BEGIN                        (* MAIN *)
   OVLY$(0);                 (* INITIALIZE HANDLER *)
   CASE PATH OF
        1,2: BEGIN
             OVLY $(1); A; END;
        3,4: BEGIN
             OVLY$(2); B; END;
        END;
END.                         (* MAIN *)
```

Notice that the first statement of the main program calls OVLY$ to initialize the handler. When PATH is one, overlay 1 is loaded by a call to OVLY$ with a parameter of 1, and procedure A is called. In procedure A, the call to OVLY$ loads overlay 3, and is followed by a call to procedure A1.

Procedure OVLY$ maintains a variable that contains the number of the most recently loaded overlay, and compares the argument in the call with that varialble. A subsequent call to load the most recently loaded overlay is ignored.

The overlay numbers used as parameters for procedure OVLY$ differ for execution under TX990 from those used with DX10. Refer to paragraph H.7.

## H.4 LINK CONTROL FILE

The object modules must be properly linked to support the overlay structure. The object modules must be available as members of a library, which may be written by utility CONFIG or SPLIT. The following example of the commands to link the program in the desired overlay structure assumes that the modules have been cataloged as members of library .USER.OBJECT:

```
LIBRARY .TIP.OBJECT
PHASE 0,OVLYTEST
INCLUDE (MAIN)
INCLUDE .USER.OBJECT(MAIN)
PHASE 1,OVERLAY1
INCLUDE .USER.OBJECT(A)
PHASE 2,OVERLAY3
INCLUDE .USER.OBJECT(A1)
PHASE 2,OVERLAY4
INCLUDE .USER.OBJECT(A2)
PHASE 1,OVERLAY2
INCLUDE .USER.OBJECT(B)
PHASE 2,OVERLAY5
INCLUDE .USER.OBJECT(B1)
PHASE 2,OVERLAY6
INCLUDE .USER.OBJECT(B2)
END
```

These commands are supplied to the link editor as the control file, and the link editor writes the linked object to a file specified for the linked output.

## H.5 INSTALLING THE PROGRAM

The root module and the overlay modules must be installed in the system. The following is an example of the batch stream of SCI commands to install the program. The example assumes that the linked object is on file. .USER.LOBJ and that the program file on which the modules are written is .USER.OTEST. The batch stream is as follows:

```
BATCH
.SYN P=".USER.OTEST"
AGL AN=.USER.LOBJ
DT PF=P,TN=OVLYTEST
IT PF=P,TN=OVLYTEST,OBJ=$$LU
IO PF=P,ON=OVERLAY1,OI=1,OBJ=$$LU,REL=NO,ASS=OVLYTEST
IO PF=P,ON=OVERLAY3,OI=3,OBJ=$$LU,REL=NO,ASS=OVLYTEST
IO PF=P,ON=OVERLAY4,OI=4,OBJ=$$LU,REL=NO,ASS=OVLYTEST
IO PF=P,ON=OVERLAY2,OI=2,OBJ=$$LU,REL=NO,ASS=OVLYTEST
IO PF=P,ON=OVERLAY5,OI=5,OBJ=$$LU,REL=NO,ASS=OVLYTEST
IO PF=P,ON=OVERLAY6,OI=6,OBJ=$$LU,REL=NO,ASS=OVLYTEST
RGL L=@$$LU
EBATCH
```

## H.6 LINK CONTROL FILE FOR TX990

The same overlay structure could be executed under TX990. The following link control file is an example of the file required for linking for TX990 execution, assuming the modules have been cataloged as members of library .USER.OBJECT:

```
FORMAT IMAGE
LIBRARY .USER.OBJECT
LIBRARY .TIP.LUNOBJ
LIBRARY .TIP.OBJ
PHASE 0, OVLYTEST
INCLUDE (TXMAIN)
INCLUDE .USER.OBJECT.MAIN; Implicit reference required to link TX990 version
INCLUDE (OVLY$TX);         OVLY$ for TX990 program files (on .TIP.LUNOBJ)
PHASE 1, OVERLAY1
INCLUDE (A)
PHASE 2, OVERLAY3
INCLUDE (A1)
PHASE 2, OVERLAY4
INCLUDE (A2)
PHASE 1, OVERLAY2
INCLUDE (B)
PHASE 2, OVERLAY5
INCLUDE (B1)
PHASE 2, OVERLAY6
INCLUDE (B2)
END
```

Execute the version of the link editor for TX990 by entering TXXLE, and specify the control file in the preceding example. TXXLE creates a TX990 program file and writes the task and overlay modules on the file.

## H.7 OVERLAY NUMBERS

The overlay numbers used in calls to procedure OVLY$ are assigned by the Install Overlay (IO) commands shown in the example in paragraph H.5. There is no analogous command for TX990. Instead, TXXLE arbitrarily assigns overlay numbers in an order related to the link control file contents. These numbers are the ones required as parameters for the OVLY$ procedure.

TXXLE assigns overlay number 1 to the overlay resulting from the first PHASE command that specifies a phase greater than 0. The overlay resulting from the next such PHASE command is overlay number 2, etc. The overlay numbers assigned by TXXLE when linking in accordance with the file described in the preceding paragraph are:

| Number | Overlay Name |
|---|---|
| 1 | OVERLAY1 |
| 2 | OVERLAY3 |
| 3 | OVERLAY4 |
| 4 | OVERLAY2 |
| 5 | OVERLAY5 |
| 6 | OVERLAY6 |

For example, when the task calls the overlay consisting of object module B, the statement for the call is:

```
OVLY$(4)
```

# APPENDIX I

# TYPE CONVERSIONS

# APPENDIX I

# TYPE CONVERSIONS

## I.1 INTRODUCTION

TI Pascal provides a limited amount of implicit conversion of data types and a set of functions to perform explicit conversions. This section describes the instances in which a conversion is implied and the methods of converting from one type to another. In several of the conversions it is possible for a mathemetical overflow to occur. A mathematical overflow results in an incorrect conversion; it does not cause an interruption of processing or abnormal termination of the program unless compiler options CKOVER (paragraph 9.5.2) and CKPREC (paragraph 9.5.3) have been specified.

## I.2 IMPLICIT CONVERSIONS

The TIP compiler provides implicit conversion of data from one type to another or to a different precision of the same type in the following instances:

- Assignment operations
- Arithmetic operations
- Comparisons
- I/O operations
- Routine parameters passed by value

**I.2.1 CONVERSIONS IN ASSIGNMENT OPERATIONS.** In an assignment operation, the data represented by the identifier or expression to the right of the assignment operator is converted to the type and precision of the variable to the left of the operator.

The following conversions are implicit:

- INTEGER to LONGINT
- LONGINT to INTEGER
- INTEGER to REAL
- LONGINT to REAL
- REAL of one precision to REAL of another precision
- FIXED of one precision to FIXED of another precision
- DECIMAL of one precision to DECIMAL of another precision

**I.2.2 CONVERSIONS IN ARITHMETIC OPERATIONS AND COMPARISONS.** Both operands of arithmetic operations and comparisons must either be the same type or must be in one of the following types that are implicitly converted to the type of the other operand:

- If one operand is INTEGER type and the other is LONGINT or REAL type, the INTEGER type operand is converted to the type of the other operand.

- If one operand is LONGINT type and the other is REAL type, the LONGINT type operand is converted to REAL type.

- If the operands of a / operation are both INTEGER or LONGINT type, the operands are converted to REAL type.

The type of the result of an arithmetic operation is the type of the operands after conversion (if required). The result of a comparison is BOOLEAN type.

When the operands of an arithmetic operation or of a comparison are REAL, FIXED, or DECIMAL, the precision of either operand often is converted. The following paragraphs discuss the conversion rules that apply.

**I.2.2.1 Precision Conversion for REAL Type.** When the precisions of the operands of REAL type differ, the operand having the smaller precision is converted to the larger precision. The larger precision is the precision of the result of arithmetic operations.

**I.2.2.2 Precision Conversion for FIXED and DECIMAL Types.** When an expression consists of an arithmetic operation between operands of FIXED or DECIMAL type, the precision and scale factor of the result is determined by the formulas shown in this paragraph. However, the precision and scale factor can be forced using the FIX or DEC function, with the desired precision and scale factor, and the expression as arguments. This results in optimization of the code, for example, when the operation is multiplication and the operands are FIXED type of 15-bit precision.

In the formulas shown, M is a parameter having the value of 15, the limit to the precision supported. The following rules apply to the precisions of the operands:

- The precision of each operand is M (15) or less.

- The precision specified for the result of function FIX or DEC is M (15) or less.

*Addition and Subtraction.* In the formulas that apply to addition and subtraction, the precision P minus the scale factor Q of an operand is the integer precision R for that operand. Similarly, the precision, scale factor, and integer precision of the other operand are P′, Q′, and R′, respectively. Using these values in the following formulas provides R″, Q″, and P″, the integer precision scale factor, and precision of the result.

When

$$1 + \mathrm{MAX}(R,R') + \mathrm{MAX}(Q,Q')$$

is less than or equal to M, the following formulas apply:

$$R'' = 1 + \mathrm{MAX}(R,R')$$

$$Q'' = \mathrm{MAX}(Q,Q')$$

$$P'' = Q'' + R''$$

The scale factor of the result (Q″) is the larger of the two operand scale factors. The operand with the smaller scale factor is shifted left before the addition or subtraction is performed. The integer precision (R″) is one greater than the larger of the two operand integer precisions to allow for the possiblitity of a carry. Overflow cannot occur.

Otherwise, when

$1 + \mathrm{MAX}(R,R') + \mathrm{MAX}(Q,Q')$

is greater than M, the following formulas apply:

$R'' = \mathrm{MAX}(R,R')$

$Q'' = M - R''$

$P'' = M$

The integer precision of the result is the larger of the two operand integer precisions. A mathematical overflow can occur. The scale factor of the result is reduced as required so that the precision P″ is equal to M.

*Multiplication.* For multiplication, the operands are multiplied without shifting. The result may be truncated on the left if $P + P'$ is greater than M. The formulas for Q″, P″, and R″ are:

$Q'' = Q + Q'$

$P'' = \mathrm{MIN}(M,P + P')$

$R'' = P'' - Q''$

For multiplication of FIXED type operands, when the sum of $P + P'$ is less than or equal to 31, truncation of the result on the left may be avoided by using the FIX function. Specifically, the result of FIX(PP,QQ,X*Y) is not truncated when PP is less than or equal to M.

*Division.* For division, the dividend of precision (P,Q) is shifted left to precision $(M,Q + (M - P))$ prior to performing the division. The formulas for P″, R″, and Q″ of the result are:

$P'' = M$

$R'' = R + Q'$

$Q'' = M - R''$

*Other Operations.* For comparisons, each operand is converted as for subtraction before the comparison. For negation or absolute value operations, the result has the same precision as the operand.

**I.2.3 CONVERSIONS IN I/O OPERATIONS.** FIXED type numbers in F format are converted on input and output. On input, a FIXED type number in F format is read into an intermediate value which is then converted. The precision and scale factor of the intermediate value are computed and the intermediate value is converted as in conversion from DECIMAL type to FIXED type (paragraph I.6.4). If the precision of the intermediate value is greater than the maximum precision, significant digits may be lost.

On output, the number is converted to an intermediate value and output in F format. The precision and scale factor of the intermediate value are computed and the intermediate value is converted as in conversion from FIXED type to DECIMAL type (paragraph I.7.4).

**I.2.4 CONVERSIONS IN ROUTINE CALLS.** When an argument of an ARCTAN, COS, EXP, LN, SIN, or SQRT or a REAL type value parameter of any routine has a type other than REAL, it is converted to type REAL of default precision. The result has the precision of the argument, except that when the argument is type REAL of a specified precision less than the default, the result is of REAL type with default precision. When the argument is type REAL of a specified precision greater than the default, the result is type REAL with the specified precision.

## I.3 CONVERSION TO INTEGER TYPE

LONGINT, REAL, FIXED, and DECIMAL types may be converted to INTEGER type. The methods of converting each of these types are described in the following paragraphs.

**I.3.1 FROM LONGINT TYPE.** The value of the LONGINT data is assigned to an INTEGER data item. When the magnitude of the value is too large to be represented as an INTEGER, a mathematical overflow occurs.

**I.3.2 FROM REAL TYPE.** The value of the integer part of the REAL data is assigned to an INTEGER data item. When the magnitude of the value is too large to be represented as an INTEGER, a mathematical overflow occurs.

**I.3.3 FROM FIXED TYPE. The value of the FIXED data is first converted to scale factor Q = 0, and the converted value is assigned to an INTEGER data item. When the magnitude of the converted value is too large to be represented as an INTEGER, a mathematical overflow occurs.**

**I.3.4 FROM DECIMAL TYPE.** The value of the DECIMAL data is first converted to FIXED type. The value of the FIXED type is converted as in paragraph I.3.3.

## I.4 CONVERSION TO LONGINT TYPE

INTEGER, REAL, FIXED, and DECIMAL types may be converted to LONGINT type. The methods of converting each of these types are described in the following paragraphs.

**I.4.1 FROM INTEGER TYPE.** The value of the INTEGER data is assigned to a LONGINT data item.

**I.4.2 FROM REAL TYPE.** The value of the integer part of the REAL data is assigned to a LONGINT data item. When the magnitude of the value is too large to be represented as LONGINT, a mathematical overflow occurs.

**I.4.3 FROM FIXED TYPE.** The value of the FIXED data is first converted to scale factor Q = 0, and the converted value is assigned to a LONGINT data item. When the magnitude of the converted value is too large to be represented as LONGINT, a mathematical overflow occurs.

**I.4.4 FROM DECIMAL TYPE.** The value of the DECIMAL data is first converted to FIXED type. The value of the FIXED type is converted as in paragraph I.4.3.

## I.5 CONVERSION TO REAL TYPE

INTEGER, LONGINT, FIXED, and DECIMAL types may be converted to REAL type. A REAL type may be converted to a REAL type with a different precision. Conversions of each of these types are described in the following paragraphs.

**I.5.1 FROM INTEGER TYPE.** The value of the INTEGER data is assigned to a REAL data item of the required precision.

**I.5.2 FROM LONGINT TYPE.** The value of the LONGINT data is assigned to a REAL data item of the required precision.

**I.5.3 FROM FIXED TYPE.** The value of the FIXED data is assigned to a REAL data item of the required precision.

**I.5.4 FROM DECIMAL TYPE.** The value of the DECIMAL data is assigned to a REAL data item of the required precision.

**I.5.5 FROM REAL TYPE OF A DIFFERENT PRECISION.** When the precision of the data being converted is greater than that of the result, the data is truncated on the right to obtain the desired precision. When the precision of the data being converted is less than that of the result, zeros are added to the right of the data to obtain the desired precision.

## I.6 CONVERSION TO FIXED TYPE

INTEGER, LONGINT, REAL, and DECIMAL types may be converted to FIXED type. A FIXED type may be converted to a FIXED type of different precision and/or scale factor. Conversions of each of these types are described in the following paragraphs.

**I.6.1 FROM INTEGER TYPE.** The value of the INTEGER data is assigned to a FIXED data item with scale factor $Q = 0$.

**I.6.2 FROM LONGINT TYPE.** The value of the LONGINT data is assigned to a FIXED data item with scale factor $Q = 0$.

**I.6.3 FROM REAL TYPE.** The value of the REAL data is assigned to a FIXED data item with the required precision and scale factor. This conversion only occurs in assignment operations and when explicitly specified by a FIX function call.

**I.6.4 FROM DECIMAL TYPE.** The value of the DECIMAL data of precision P and scale factor Q is converted to a FIXED data item of precision P′ and scale factor Q′. P′ and Q′ correspond to precision P and scale factor Q of the DECIMAL data, and are computed as follows:

$$P' = 1 + \text{CEIL}(P*3.322)$$

When Q is a positive value:

$$Q' = \text{CEIL}(Q*3.322)$$

When Q is a negative value:

$$Q' = -\ \text{CEIL}(-Q*3.322)$$

When $Q = 0$, $Q' = 0$.

The values of CEIL for an appropriate range of numbers are listed in table I-1. Function CEIL has the value of the next integer higher than the quotient of the number divided by 3.322, an approximation of the base 2 logarithm of 10. The FIXED data item of precision P′ and scale factor Q′ is then converted to a FIXED binary data item of the required precision and scale factor.

TABLE I-1. FIXED/DECIMAL Conversions

| Number of decimal digits N | Number of bits CEIL (N*3.322) | Number of bits N | Number of decimal digits CEIL (N/3.322) |
|---|---|---|---|
| 1 | 4 | 1 - 3 | 1 |
| 2 | 7 | 4 - 6 | 2 |
| 3 | 10 | 7 - 9 | 3 |
| 4 | 14 | 10 - 13 | 4 |
| 5 | 17 | 14 - 16 | 5 |
| 6 | 20 | 17 - 19 | 6 |
| 7 | 24 | 20 - 23 | 7 |
| 8 | 27 | 24 - 26 | 8 |
| 9 | 30 | 27 - 29 | 9 |
| 10 | 34 | 30 - 33 | 10 |
| 11 | 37 | 34 - 36 | 11 |
| 12 | 40 | 37 - 39 | 12 |
| 13 | 44 | 40 - 43 | 13 |
| 14 | 47 | 44 - 46 | 14 |
| 15 | 50 | 47 - 49 | 15 |

**I.6.5 FROM FIXED TYPE OF DIFFERENT PRECISION.** The value of the FIXED data of precision P and scale factor Q is converted to a FIXED data item of precision P′ and scale factor Q′.

When Q is less than Q′, the data is shifted left to precision (P+Q′-Q) and scale factor Q′ and assigned to a FIXED data item of precision P′ and scale factor Q′. If (P+Q′-Q) is greater than P′ or the maximum precision for FIXED type, high order bits may be lost and overflow may occur. Zeros replace low order bits vacated during the shift.

When Q is greater than Q′, the data is shifted right to precision (P+Q′-Q) and scale factor Q′. If the last bit shifted out is a 1, the result is incremented by 1. The result is assigned to a FIXED data item of precision P′ and scale factor Q′. If (P+Q′-Q) is greater than P′, high order bits may be lost and overflow may occur.

## I.7 CONVERSION TO DECIMAL TYPE

INTEGER, LONGINT, REAL, and FIXED types may be converted to DECIMAL type. A DECIMAL type may be converted to a DECIMAL type of different precision and/or scale factor. Conversions of each of these types are described in the following paragraphs.

**I.7.1 FROM INTEGER TYPE.** The value of the INTEGER data is assigned to a DECIMAL data item with scale factor Q = 0.

**I.7.2 FROM LONGINT TYPE.** The value of the LONGINT data is assigned to a DECIMAL data item with scale factor Q = 0.

**I.7.3 FROM REAL TYPE.** The value of the REAL data is assigned to a DECIMAL data item with the required precision and scale factor. This conversion occurs in assignment operations only.

**I.7.4 FROM FIXED TYPE.** The value of the FIXED data of precision P and scale factor Q is converted to a DECIMAL data item of precision P′ and scale factor Q′. Values P′ and Q′ correspond to precision P and scale factor Q of the FIXED data, and are computed as follows:

P′ = 1 + CEIL(P/3.322)

When Q is a positive value:

$$Q' = CEIL(Q/3.322)$$

When Q is a negative value:

$$Q' = -CEIL(-Q/3.322)$$

When $Q = 0$, $Q' = 0$.

The values of CEIL for an appropriate range of numbers are listed in table I-1.

Function CEIL has the value of the next integer higher than the product of the number times 3.322, an approximation of the base 2 logarithm of 10. The DECIMAL data item of precision P′ and scale factor Q′ is then converted to a DECIMAL data item of the required precision and scale factor.

**I.7.5 FROM DECIMAL TYPE OF DIFFERENT PRECISION.** The value of the DECIMAL data of precision P and scale factor Q is converted to a DECIMAL data item of precision P′ and scale factor Q′.

When Q is less than Q′, the data is shifted left to precision (P+Q′-Q) and scale factor Q′ and assigned to a DECIMAL data item of precision P′ and scale factor Q′. If (P+Q′-Q) is greater than P′ or the maximum precision for DECIMAL type, high order digits may be lost and overflow may occur. Zeros replace low order digits vacated during the shift.

When Q is greater than Q′, the data is shifted right to precision (P+Q′-Q) and scale factor Q′. The result is assigned to a DECIMAL data item of precision P′ and scale factor Q′. If (P+Q′-Q) is greater than P′, high order digits may be lost and overflow may occur.

# ALPHABETICAL INDEX

# ALPHABETICAL INDEX

## INTRODUCTION

### HOW TO USE THE INDEX

The index, table of contents, list of illustrations, and list of tables are used in conjunction to obtain the location of the desired subject. Once the subject or topic has been located in the index, use the appropriate paragraph number, figure number, or table number to obtain the corresponding page number from the table of contents, list of illustrations, or list of tables. The table of contents does not contain four-level paragraph entries. Therefore, for four-level paragraph numbers such as 2.3.1.2, use the three-level number and the corresponding page number. In this case, the three-level number is 2.3.1.

### INDEX ENTRIES

The following index lists key words and concepts from the subject material of the manual together with the area(s) in the manual that supply major coverage of the listed concept. The numbers along the right side of the listing reference the following manual areas:

- Sections - References to Sections of the manual appear as "Section x" with the symbol x representing any numeric quantity.

- Appendixes - References to Appendixes of the manual appear as "Appendix y" with the symbol y representing any capital letter.

- Paragraphs - References to paragraphs of the manual appear as a series of alphanumeric or numeric characters punctuated with decimal points. Only the first character of the string may be a letter; all subsequent characters are numbers. The first character refers to the section or appendix of the manual in which the paragraph is found.

- Tables - References to tables in the manual are represented by the capital letter T followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the table). The second character is followed by a dash (-) and a number:

  Tx-yy

- Figures - References to figures in the manual are represented by the capital letter F followed immediately by another alphanumeric character (representing the section or appendix of the manual containing the figure). The second character is followed by a dash (-) and a number:

  Fx-yy

- Other entries in the Index - References to other entries in the index are preceded by the word "See"followed by the referenced entry.

Abnormal Termination Dump . . . . . . . . F14-2
  Analysis . . . . . . . . . . . . . . . . . . . . . 14.7.5
  Analysis, TX990 . . . . . . . . . . . . . . . . 14.7.7
  Analysis, Unformatted . . . . . . . . . . . . 14.7.6
  TX990 . . . . . . . . . . . . . . . . . . . . . . F14-5
  Unformatted . . . . . . . . . . . . . . . . . F14-4
ABP Command . . . . . . . . . . . . . . . . 14.6.1.1
ABS Function . . . . . . . . 5.2, 5.10, 5.11, 5.12
ACCESS:
  Declaration . . . . . . . . . . . . . . . . 8.2.6, 8.9
  Procedures, System Common . . . . . . . 15.3.6
  Routine . . . . . . . . . . . . . . . . . . . . . F8-5
  Variable . . . . . . . . . . . . . . . . F.5.1, F8-4
ACTIVATE Procedure . . . . . . . . . . . . 15.3.4.3
ADD Command . . . . . . . . . . . . . . . . 12.3.3.2
Adding Operators . . . . . . . . . . . . . . . . . . 4.2
ALTOBJ Command . . . . . . . . . . . . . . 12.3.8.4
Analysis:
  Abnormal Termination Dump . . . . . . 14.7.5
  Error . . . . . . . . . . . . . . . . . . . . . . 14.7.5
  Stack Frame . . . . . . . . . . . . . . . . . 14.7.5
  TX990 Abnormal Termination
    Dump . . . . . . . . . . . . . . . . . . . . 14.7.7
  Unformatted Abnormal Termination
    Dump . . . . . . . . . . . . . . . . . . . . 14.7.6
AND Operator . . . . . . . . . . . . . . . . . 4.2, 5.3
ARCTAN Function . . . . . . . . 5.10, 5.11, 5.12
Arithmetic Operation Type Conversions . . I.2.2
Array:
  Display . . . . . . . . . . . . . . . . . . . . . F.4.2
  Packed . . . . . . . . . . . . . . . . . . . . . 6.7.1
  Parameters, Dynamic . . . . . . . . . . . 8.5.2
  Type . . . . . . . . . . . . . . . . . . . 6.1, 6.7.3
    Function . . . . . . . . . . . . . . . . . . . . 6.1
    Procedures . . . . . . . . . . . . . . . . . 6.1.1
Arrays:
  Dynamic . . . . . . . . . . . . . . . . . . . . 6.1.2
  Multidimensional . . . . . . . . . . . . . . . . 6.1
ASCII Character Set . . . . . Appendix G, TG-1
Assembly Language Routines . . . . . . F.1, F.6
ASSERT Statement . . . . . . . . . . . . . . . . 5.4
ASSERTS Option . . . . . . . . . . . . . . . 9.4.4
Assign Breakpoint-Pascal Command . . . 14.6.1.1
Assignment Operation Type Conversions . . I.2.1
Assignment Statement . . . . . . . . . . . . . . 4.1

Backus-Naur Form . . . . . . . . . . . . . . . . 3.1
Backus-Normal Form . . . . . . . . . . . . . . . 3.1
Batch:
  CONFIG Execution, DX10 . . . . . . . F12-10
  SPLITPGM Execution . . . . . . . . . . . 12.4.4
  Stream, DX10 TIP . . . . . . . . . . . . . F14-1
  TIP Execution, DX10 . . . . . . . . . . . . 14.5
BID Procedure . . . . . . . . . . . . . . . . 15.3.4.1
Block:
  Structure . . . . . . . . . . . . . . . . . . . . . . 8.6
  Syntax Diagram . . . . . . . . . . . . . . . . F8-1
Blocks:
  Global Common . . . . . . . . . . . . . . . 15.6
  Local Common . . . . . . . . . . . . . . . . 15.6
BNF:
  Examples . . . . . . . . . . . . . . . . . . . . . 3.1
  Productions . . . . . . . . . . . . . . Appendix D

BOOLEAN Type . . . . . . . . . . . . . . 5.3, 6.7.3
  Operators . . . . . . . . . . . . . . . . . . . . . 5.3
BUILD PROCESS Command . . . . . . . 12.3.3.1

Call:
  Function . . . . . . . . . . . . . . . . . . . . . . 8.4
  Procedure . . . . . . . . . . . . . . . . . . . . . 8.4
    Supervisor . . . . . . . . . . . . . . . . . 15.3.8
Calling, Routine . . . . . . . . . . . . . . . . F.5.2
CASE Statement . . . . . . . . . . . . . . 5.8, 5.8.1
CAT PROCESS Command . . . . . . . . 12.3.3.3
Categories, Routine . . . . . . . . . . . . . . . F.3
CEIL Values . . . . . . . . . . . . . . . . . . . TI-1
Changing LUNO For OUTPUT . . . . . . . 15.5
CHAR Type . . . . . . . . . . . . . . . . 5.7.1, 6.7.3
Character Set . . . . . . . . . . . . . . . . . . . . 3.2
  ASCII . . . . . . . . . . . . . Appendix G, TG-1
Checks, Runtime . . . . . . . . . . . . . . . . 14.7.1
CHR Function . . . . . . . . . . . . . . . 5.2, 5.7.1
Circuit Evaluation, Short . . . . . . . . . . . . 5.9
CKINDEX Option . . . . . . . . . . . . . . . 9.5.1
CKOVER Option . . . . . . . . . . . . . . . 9.5.2
CKPREC Option . . . . . . . . . . . . . . . 9.5.3
CKPTR Option . . . . . . . . . . . . . . . . 9.5.4
CKSET Option . . . . . . . . . . . . . . . . 9.5.5
CKSUB Option . . . . . . . . . . . . . . . . 9.5.6
CKTAG Option . . . . . . . . . . . . . . . . 9.5.7
CLOSE Procedure . . . . . . . . . . . . . . . . . 6.5
Code, Condition . . . . . . . . . . . . . . . . 14.4.2
CODEGEN . . . . . . . . . . . . . . . . . . . . 11.4
Codes:
  Device Type . . . . . . . . . . . . . . . . . T15-1
  DX10:
    Condition . . . . . . . . . . . . . . . . . . . 11.5
    Device Type . . . . . . . . . . . . . . . 15.3.1.5
  Runtime Error . . . . . . . . . . . . . . . . . E.3
  TEXTFILE Error . . . . . . . . . . . . . . . 14.7.2
COLLECT Command . . . . . . . . . . . . . 12.3.6
COLUMN Function . . . . . . . . . . . . . 6.5.2
Command:
  ABP . . . . . . . . . . . . . . . . . . . . . . 14.6.1.1
  ADD . . . . . . . . . . . . . . . . . . . . . . 12.3.3.2
  ALTOBJ . . . . . . . . . . . . . . . . . . . . 12.3.8.4
  Assign Breakpoint-Pascal . . . . . . . . . 14.6.1.1
  BUILD PROCESS . . . . . . . . . . . . . . 12.3.3.1
  CAT PROCESS . . . . . . . . . . . . . . 12.3.3.3
  COLLECT . . . . . . . . . . . . . . . . . . . 12.3.6
  COMPILE . . . . . . . . . . . . . . . . . . . 12.3.4.1
  Conditional Flag . . . . . . . . . . . . . . 12.3.6.3
  DBP . . . . . . . . . . . . . . . . . . . . . . 14.6.1.2
  DEFAULT OBJECT . . . . . . . . . . . 12.3.8.7
  DEFAULT SOURCE . . . . . . . . . . . 12.3.8.6
  DELETE . . . . . . . . . . . . . . . . . . . 12.3.7.1
  Delete and Proceed From Breakpoint-
    Pascal . . . . . . . . . . . . . . . . . . . 14.6.1.3
  Delete Breakpoint-Pascal . . . . . . . . . 14.6.1.2
  DISPLAY . . . . . . . . . . . . . . . . . . . 12.3.7.3
  DPBP . . . . . . . . . . . . . . . . . . . . . 14.6.1.3
  EDIT . . . . . . . . . . . . . . . . . . . . . 12.3.9.1
  Example, Process Configuration . . . . 12.3.3.4
  Examples, Flag . . . . . . . . . . . . . . . 12.3.6.4
  EXIT . . . . . . . . . . . . . . . . . . . . . 12.3.4.3
  Flag . . . . . . . . . . . . . . . . . . . . . . 12.3.6.2

INSERT . . . . . 12.3.9.2
LBP . . . . . 14.6.1.5
LIBRARY . . . . . 12.3.8.2
LIST . . . . . 12.3.5.1
List Breakpoints-Pascal . . . . . 14.6.1.5
List Pascal Stack . . . . . 14.6.1.7
LISTDOC . . . . . 12.3.5.2
LISTORDER . . . . . 12.3.5.3
LPS . . . . . 14.6.1.7
MASTER . . . . . 12.3.8.1
MOVE . . . . . 12.3.7.2
OBJLIB . . . . . 12.3.8.3
PBP . . . . . 14.6.1.4
Proceed From Breakpoint-Pascal . . . 14.6.1.4
REPLACE . . . . . 12.3.9.3
SETFLAG . . . . . 12.3.6.1
SETLIB . . . . . 12.3.8.5
Show Pascal Stack . . . . . 14.6.1.6
SPLIT OBJECT . . . . . 12.3.4.2
Split Program . . . . . 12.4.1
SPS . . . . . 14.6.1.6
TXXLE SCI . . . . . 14.3.5
USE . . . . . 12.3.7.5
USE OBJECT . . . . . 12.3.7.4
USE PROCESS . . . . . 12.3.3.5
XCODE SCI . . . . . 11.5
XCONFIG SCI . . . . . 12.3.11
XCONFIGI SCI . . . . . 12.3.11
XHT . . . . . 14.3.4
XNESTER SCI . . . . . 10.3.5
XPT SCI . . . . . 12.4.4, 14.3.2
XRASS SCI . . . . . 13.3
XSILT SCI . . . . . 11.5
XSPLIT SCI . . . . . 12.5
XT . . . . . 14.3.4
XTIP SCI . . . . . 11.5
XTS . . . . . 14.3.4

Commands:
Debug . . . . . 14.6.1
NESTER . . . . . 10.3.4

Comment:
NESTER Option . . . . . 10.3.2
Option . . . . . 9.2

Comments . . . . . 3.3

Common:
Blocks:
Global . . . . . 15.6
Local . . . . . 15.6
Declaration . . . . . 8.2.5, 8.9

Comparison Type Conversions . . . . . I.2.2

Comparisons:
FORTRAN . . . . . I.3
PL/I . . . . . I.4

Compatibility, Type . . . . . 6.8

Compilation, Separate . . . . . 12.1, 12.3.4
Examples . . . . . 12.3.4.4
Flowchart . . . . . F12-1
Guidelines . . . . . 12.3.4.1
Requirements . . . . . 12.2

COMPILE Command . . . . . 12.3.4.1

Compiler:
Error Messages . . . . . E.1
Errors . . . . . 11.6
Execution . . . . . 11.5
Files . . . . . T11-1
Option . . . . . 4.1, 6.2.2
Options . . . . . 9.1, T9-1
Phases . . . . . 11.1
SYSMSG Display . . . . . F11-2

Components, Structured Type . . . . . 2.1.2

Compound Statement . . . . . 5.5

Concepts, Data Type . . . . . 2.2

Condition:
Code . . . . . 14.4.2
Codes, DX10 . . . . . 11.5

Conditional Flag Command . . . . . 12.3.6.3

Conditional Statements . . . . . 5.6

CONFIG . . . . . 12.3
Description . . . . . 12.3.1
Execution . . . . . 12.3.11
DX10 Batch . . . . . F12-10
Files . . . . . 12.3.10, T12-2
Libraries . . . . . 12.3.8

Configuration Module, TX990 . . . . . 14.3.7

Configuration Processor . . . . . 12.3
Description . . . . . 12.3.1
Files . . . . . 12.3.10, T12-2
Libraries . . . . . 12.3.8

Considerations, Task Execution . . . . . 14.4

CONST Declaration . . . . . 8.2.2

Constant:
DECIMAL . . . . . 5.12
Expressions . . . . . 5.9
Integer . . . . . 5.9
FIXED . . . . . 5.11
Hexadecimal . . . . . 3.2
INTEGER . . . . . 3.2
Manifest . . . . . 5.7.3, 5.8.1, 6.1.2, 6.4, 8.5.2
REAL . . . . . 3.2

Constants, String . . . . . 3.2.3

Conventions, Debug Command . . . . . 14.6.1

Conversion:
DECIMAL Type Precision . . . I.2.2.2, I.7.5
DECIMAL-FIXED Type . . . . . I.6.4
DECIMAL-INTEGER Type . . . . . I.3.4
DECIMAL-LONGINT Type . . . . . I.4.4
DECIMAL-REAL Type . . . . . I.5.4
FIXED Type Precision . . . . . I.2.2.2, I.6.5
FIXED-DECIMAL Type . . . . . I.7.4
FIXED-INTEGER Type . . . . . I.3.3
FIXED-LONGINT Type . . . . . I.4.3
FIXED-REAL Type . . . . . I.5.3
INTEGER-DECIMAL Type . . . . . I.7.1
INTEGER-FIXED Type . . . . . I.6.1
INTEGER-LONGINT Type . . . . . I.4.1
INTEGER-REAL Type . . . . . I.5.1
LONGINT-DECIMAL Type . . . . . I.7.2
LONGINT-FIXED Type . . . . . I.6.2
LONGINT-INTEGER Type . . . . . I.3.1
LONGINT-REAL Type . . . . . I.5.2
REAL Type Precision . . . . . I.2.2.1, I.5.5
REAL-DECIMAL Type . . . . . I.7.3
REAL-FIXED Type . . . . . I.6.3
REAL-INTEGER Type . . . . . I.3.2
REAL-LONGINT Type . . . . . I.4.2

Conversion To:
DECIMAL Type . . . . . I.7
FIXED Type . . . . . I.6

INTEGER Type . . . . . . . . . . . . . . . . . . . I.3
LONGINT Type . . . . . . . . . . . . . . . . . I.4
REAL Type . . . . . . . . . . . . . . . . . . . . I.5
Conversions:
Arithmetic Operation Type . . . . . . . . . I.2.2
Assignment Operation Type . . . . . . . . . I.2.1
Comparison Type . . . . . . . . . . . . . . . . I.2.2
Implicit Type . . . . . . . . . . . . . . . . . . . I.2
I/O Operation Type . . . . . . . . . . . . . I.2.3
Mathematical Function Type . . . . . . . . I.2.4
Type . . . . . . . . . . . . . . . . . . . . . . . . I.1
COS Function . . . . . . . . . . . 5.10, 5.11, 5.12
CRU Routines . . . . . . . . . . . . . . . . . . 15.2

Data:
Scope . . . . . . . . . . . . . . . . . . . . . . . . 2.1
Stack Frame . . . . . . . . . . . . . . . . . F.4.3
Type:
Concepts . . . . . . . . . . . . . . . . . . . . . 2.2
Packed . . . . . . . . . . . . . . . . . . . . . . 6.7
Types . . . . . . . . . . . . . . . . . . . . . . . . 2.2
DATE:
Procedure . . . . . . . . . . . . . . . . . . . . A.3
Procedures . . . . . . . . . . . . . . . . . . 15.3.3
DBP Command . . . . . . . . . . . . . . . . 14.6.1.2
Debug:
Commands . . . . . . . . . . . . . . . . . . . 14.6.1
Mode . . . . . . . . . . . . . . . . . . . . . . . . 14.6
Debug Command:
Conventions . . . . . . . . . . . . . . . . . . 14.6.1
Error Messages . . . . . . . . . . . . . . . . . E.4
Debugging, TX990 . . . . . . . . . . . . . . . . 14.6.2
DEC Function . . . . . . . . 5.2, 5.10, 5.11, 5.12
DECIMAL:
Constant . . . . . . . . . . . . . . . . . . . . . . 5.12
Type . . . . . . . . . . . . . . . . . . . . . 5.12, 6.7.3
Conversion To . . . . . . . . . . . . . . . . . I.7
Functions . . . . . . . . . . . . . . . . . . . . 5.12
Operators . . . . . . . . . . . . . . . . . . . . 5.12
DECIMAL Type Precision
Conversion . . . . . . . . . . . . . . I.2.2.2, I.7.5
DECIMAL-FIXED Type Conversion . . . . I.6.4
DECIMAL-INTEGER Type Conversion . . I.3.4
DECIMAL-LONGINT Type Conversion . . I.4.4
DECIMAL-REAL Type Conversion . . . . . I.5.4
Declaration:
ACCESS . . . . . . . . . . . . . . . . . 8.2.6, 8.9
COMMON . . . . . . . . . . . . . . . . 8.2.5, 8.9
CONST . . . . . . . . . . . . . . . . . . . . . 8.2.2
EXTERNAL . . . . . . . . . . . . . . . . . 8.2.7.2
FORWARD . . . . . . . . . . . . . . 8.2.7.1, 8.7
LABEL . . . . . . . . . . . . . . . . . . . 7.2, 8.2.1
Portion . . . . . . . . . . . . . . . . . . . . . . . . 8.2
Routine . . . . . . . . . . . . . . . . . . . . . 8.2.7
TYPE . . . . . . . . . . . . . . . . . . . . . . . 8.2.3
VAR . . . . . . . . . . . . . . . . . . . . . . . . 8.2.4
Declarations . . . . . . . . . . . . . . . . . . . . . . 2.3
FORTRAN Routine . . . . . . . . . . . . . 8.2.7.2
DECODE Procedure . . . . . . . . . . . 6.5.4, 6.5.6
DEFAULT OBJECT Command . . . . . 12.3.8.7
DEFAULT SOURCE Command . . . . . 12.3.8.6
Deferred Processing, OUTPUT File:
Full Compilation . . . . . . . . . . . . . . . F12-4
Partial Compilation . . . . . . . . . . . . . F12-7
DELAY Procedure . . . . . . . . . . . . . . 15.3.3.3
Delete and Proceed From Breakpoint-
Pascal Command . . . . . . . . . . . . . . 14.6.1.3
Delete Breakpoint-Pascal Command . . . 14.6.1.2
DELETE Command . . . . . . . . . . . . . 12.3.7.1
Description:
CONFIG . . . . . . . . . . . . . . . . . . . . 12.3.1
Configuration Processor . . . . . . . . . . 12.3.1
Device I/O, DX10 . . . . . . . . . . . . . . . . . C.3
Device Type Codes . . . . . . . . . . . . . . . T15-1
Device Type Codes, DX10 . . . . . . . . . 15.3.1.5
DEV$TYPE Function . . . . . . . . . . . . 15.3.1.5
Diagram, Syntax . . . . . . . . . 3.1, Appendix D
Block . . . . . . . . . . . . . . . . . . . . . . . . F8-1
Example . . . . . . . . . . . . . . . . . . . . . . . 3.1
Expression . . . . . . . . . . . . . . . . . . . . F5-1
Routine . . . . . . . . . . . . . . . . . . . . . . F8-2
Statement . . . . . . . . . . . . . . . . . . . . F8-3
Symbols . . . . . . . . . . . . . . . . . . . . . F3-1
Display:
Array . . . . . . . . . . . . . . . . . . . . . . . F.4.2
Command . . . . . . . . . . . . . . . . . . 12.3.7.3
Compiler SYSMSG . . . . . . . . . . . . . . F11-2
PROBER Option . . . . . . . . . . . . . . . . F9-2
PROBES Option . . . . . . . . . . . . . . . . F9-2
DISPOSE Procedure . . . . . . . . . . . 6.6.1, 8.7
DIV Operator . . . . . . . . . . . . . . . . . . 4.2, 5.2
Documentation Section . . . . . . . 12.3.2, 12.3.5
Documents, Related . . . . . . . . . . . . . . Preface
DPBP Command . . . . . . . . . . . . . . . 14.6.1.3
DRCT Flag . . . . . . . . . . . . . . . . . . . . . 9.3.3
DSPLY$ Procedure . . . . . . . . . . . . . . . . 15.7
Dummy Main Routine . . . . . . . . . . . . . . 15.4
Dump:
Abnormal Termination . . . . . . . . . . . F14-2
Analysis:
Abnormal Termination . . . . . . . . . . 14.7.5
TX990 Abnormal Termination . . . . . 14.7.7
Unformatted Abnormal
Termination . . . . . . . . . . . . . . . . 14.7.6
TX990 Abnormal Termination . . . . . . F14-5
Unformatted Abnormal Termination . . F14-4
DX10:
Batch:
CONFIG Execution . . . . . . . . . . . F12-10
TIP Execution . . . . . . . . . . . . . . . . 14.5
Condition Codes . . . . . . . . . . . . . . . . 11.5
CRU Routines . . . . . . . . . . . . . . . . . . 15.2
Device:
I/O . . . . . . . . . . . . . . . . . . . . . . . . C.3
Type Codes . . . . . . . . . . . . . . . . 15.3.1.5
Information . . . . . . . . . . . . . . . . . . . . C.1
Random Files . . . . . . . . . . . . . . . . . C.2.3
Sequential Files . . . . . . . . . . . . . . . . C.2.1
TEXTFILES . . . . . . . . . . . . . . . . . . C.2.2
TIP:
Batch Stream . . . . . . . . . . . . . . . . F14-1
Files . . . . . . . . . . . . . . . . . . . . . . . C.2
DX10 Execution Option . . . . . . . . . . . . 14.2.1
DX10 LUNO:
Execution . . . . . . . . . . . . . . . . . . . . 14.3.4
Linking . . . . . . . . . . . . . . . . . . . . . . 14.3.3

DX10 With SCI:
  Execution . . . . . 14.3.2
  Linking . . . . . 14.3.1
Dynamic:
  Array Parameters . . . . . 8.5.2
  Arrays . . . . . 6.1.2
  Set Parameters . . . . . 8.5.2
  Sets . . . . . 6.4
  Variable . . . . . 6.6.1

EDIT Command . . . . . 12.3.9.1
Editing, Text . . . . . 12.3.9
Effects, Side . . . . . 8.8
Elements, Syntax . . . . . 3.2
Empty Statement . . . . . 5.5
ENCODE Procedure . . . . . 6.5.4, 6.5.6
ENDPROG Option . . . . . 14.3.5
Entire Variable . . . . . 6.1.2
Enumeration:
  Type Functions . . . . . 2.2.1.1
  Types . . . . . 2.2.1.1, 5.1
EOF Function . . . . . 6.5, 6.5.5, 6.5.6
EOLN Function . . . . . 6.5.2, 6.5.3, 6.5.6
Error:
  Analysis . . . . . 14.7.5
  Codes, Runtime . . . . . E.3
  Listing Example . . . . . F11-3
  Messages . . . . . Appendix E
    Compiler . . . . . E.1
    Debug Command . . . . . E.4
    NESTER . . . . . 10.3.6
    Runtime . . . . . E.2
  Termination . . . . . 14.7.4
Errors:
  Compiler . . . . . 11.6
  I/O . . . . . 14.7.2
  Memory Space . . . . . 14.7.3
  NESTER . . . . . T10-2
  Runtime . . . . . 14.7
Escape:
  Label . . . . . 7.1, 8.2
  Statement . . . . . 7.1
Evaluation, Short Circuit . . . . . 5.9
Example:
  Error Listing . . . . . F11-3
  Input . . . . . 12.4.2, F12-11
  Linking Map . . . . . F14-3
  LISTDOC . . . . . F12-9
  NESTER . . . . . 10.3.7, F10-2
    Statement . . . . . F10-1
  PROBER Option . . . . . F9-3
  PROBES Option . . . . . F9-3
  Process Configuration Command . . . 12.3.3.4
  Program . . . . . 2.5, F2-1
    GENERATE . . . . . FB-4
    QUAD . . . . . FB-2
    RANDOM . . . . . FB-3
    ROOTS . . . . . FB-1
  Programs . . . . . Appendix B
  RASS . . . . . 13.4, F13-1
  Reverse Assembler . . . . . 13.4, F13-1
  Semaphore . . . . . F15-1
  Source Listing . . . F9-1, F9-3, F11-1, F12-8
  Syntax Diagram . . . . . 3.1

Examples:
  BNF . . . . . 3.1
  Flag Command . . . . . 12.3.6.4
  Separate Compilation . . . . . 12.3.4.4
  Source Listing . . . . . 12.3.5.4
Execution:
  Batch SPLITPGM . . . . . 12.4.4
  Compiler . . . . . 11.5
  CONFIG . . . . . 12.3.11
  Considerations, Task . . . . . 14.4
  DX10:
    Batch CONFIG . . . . . F12-10
    Batch TIP . . . . . 14.5
  DX10 LUNO . . . . . 14.3.4
  DX10 With SCI . . . . . 14.3.2
  Minimal Runtime . . . . . 14.3.12
  NESTER . . . . . 10.3.5
  RASS . . . . . 13.3
  Reverse Assembler . . . . . 13.3
  RX990 . . . . . 14.3.10
  SPLIT . . . . . 12.5
  Split Object Utility . . . . . 12.5
  Split Program . . . . . 12.4.4
  SPLITPGM . . . . . 12.4.4
  Stand-Alone . . . . . 14.3.14
  TIP Program . . . . . 14.1, 14.3
  TX990 . . . . . 14.3.6
EXIT Command . . . . . 12.3.4.3
EXP Function . . . . . 5.10, 5.11, 5.12
Expression Syntax Diagrams . . . . . F5-1
Expressions . . . . . 4.2, 5.9
  Constant . . . . . 5.9
  Integer Constant . . . . . 5.9
  Mixed Mode . . . . . 5.9
EXTEND Procedure . . . . . 6.5, 6.5.5, 14.3.1
Extensions, Pascal . . . . . 1.2
Extent . . . . . 8.7
EXTERNAL Declaration . . . . . 8.2.7.2

Field, Tag . . . . . 6.2.2
Fields:
  File Descriptor . . . . . 14.7.4
  Process Record . . . . . 14.7.4
  Stack Frame . . . . . 14.7.4
File:
  INPUT . . . . . 6.5
  OUTPUT . . . . . 6.5
  SHRPROC . . . . . 14.3.15
  Type . . . . . 6.5, 6.7.3
File Descriptor Fields . . . . . 14.7.4
Files:
  Compiler . . . . . T11-1
  CONFIG . . . . . 12.3.10, T12-2
  Configuration Processor . . . . 12.3.10, T12-2
  DX10:
    Random . . . . . C.2.3
    Sequential . . . . . C.2.1
    TIP . . . . . C.2
  RANDOM . . . . . 6.5, 6.5.5
  RASS . . . . . 13.2
  Reverse Assembler . . . . . 13.2
  Sequential . . . . . 6.5, 6.5.1
  Split Program . . . . . 12.4.3
  SPLITPGM . . . . . 12.4.3

FIND$SYN Procedure . . . . . . . . . . . . 15.3.1.1
FIX Function . . . . . . . . . 5.2, 5.10, 5.11, 5.12
FIXED:
  Constant . . . . . . . . . . . . . . . . . . . . . . . 5.11
  Type . . . . . . . . . . . . . . . . . . . . 5.11, 6.7.3
    Conversion To . . . . . . . . . . . . . . . . . I.6
    Functions . . . . . . . . . . . . . . . . . . . . 5.11
    Operators . . . . . . . . . . . . . . . . . . . . 5.11
FIXED Type Precision
  Conversion . . . . . . . . . . . . . . I.2.2.2, I.6.5
FIXED-DECIMAL Type Conversion . . . . I.7.4
FIXED-INTEGER Type Conversion . . . . . I.3.3
FIXED-LONGINT Type Conversion . . . . I.4.3
FIXED-REAL Type Conversion . . . . . . . I.5.3
Flag:
  Command . . . . . . . . . . . . . . . . . . . 12.3.6.2
    Examples . . . . . . . . . . . . . . . . . . 12.3.6.4
  DRCT . . . . . . . . . . . . . . . . . . . . . . . 9.3.3
Flags, Process Configuration . . . . 12.3.6, T12-1
FLOAT Function . . . . . . 5.2, 5.10, 5.11, 5.12
Floating Point Instructions . . . . . . . . . . TE-1
Flowchart, Separate Compilation . . . . . . F12-1
FOR Statement . . . . . . . . . . . . 5.8, 5.8.2, 8.2
FORINDEX Option . . . . . . . . . . . . . . 9.6.2
Form:
  Backus-Naur . . . . . . . . . . . . . . . . . . . . 3.1
  Backus-Normal . . . . . . . . . . . . . . . . . . 3.1
Format:
  Program . . . . . . . . . . . . . . . . . . . . . . . 8.1
  Source Module . . . . . . . . . . . . . . . . 12.3.2
Formatted I/O . . . . . . . . . . . . . . . . . . 6.5.3
FORTRAN Comparisons . . . . . . . . . . . . . I.3
FORTRAN Routine Declarations . . . . 8.2.7.2
FORWARD Declaration . . . . . . . 8.2.7.1, 8.7
Frame:
  Analysis, Stack . . . . . . . . . . . . . . . 14.7.5
  Stack . . . . . . . . . . . . . . . . . . . . . . . . . 8.7
Full Compilation:
  Deferred Processing, OUTPUT File . . . F12-4
  Initial Run, OUTPUT File . . . . . . . . F12-2
  Source Listing . . . . . . . . . . . . . . . . F12-3
Function:
  ABS . . . . . . . . . . . . . 5.2, 5.10, 5.11, 5.12
  ARCTAN . . . . . . . . . . . . 5.10, 5.11, 5.12
  Array Type . . . . . . . . . . . . . . . . . . . . . 6.1
  Call . . . . . . . . . . . . . . . . . . . . . . . . . . 8.4
  CHR . . . . . . . . . . . . . . . . . . . . 5.2, 5.7.1
  COLUMN . . . . . . . . . . . . . . . . . . . . . 6.5.2
  COS . . . . . . . . . . . . . . . 5.10, 5.11, 5.12
  DEC . . . . . . . . . . . . 5.2, 5.10, 5.11, 5.12
  DEV$TYPE . . . . . . . . . . . . . . . . . 15.3.1.5
  EOF . . . . . . . . . . . . . . . . 6.5, 6.5.5, 6.5.6
  EOLN . . . . . . . . . . . . . . 6.5.2, 6.5.3, 6.5.6
  EXP . . . . . . . . . . . . . . . 5.10, 5.11, 5.12
  FIX . . . . . . . . . . . . . 5.2, 5.10, 5.11, 5.12
  FLOAT . . . . . . . . . . 5.2, 5.10, 5.11, 5.12
  LINT . . . . . . . . . . . . . . . . . . . . . . . . . 5.2
  LN . . . . . . . . . . . . . . . . 5.10, 5.11, 5.12
  LOCATION . . . . . . . . . . . . . . . . . 5.2, 6.6
  LROUND . . . . . . . . . . . . . . . . . . . . . . A.2
  LTRUNC . . . . . . . . . . . . . . . . . . . . . . A.2
  ODD . . . . . . . . . . . . . . . . . . . . . . . . . 5.2
  ORD . . . . . . . . . . . . . 2.2.1.1, 5.7.1, 5.7.2
  Parameters . . . . . . . . . . . . . . . . . . . . 8.5.3
  PRED . . . . . . . . . . . . . . . . . . 2.2.1.1, 5.7.1
  ROUND . . . . . . . . . . . . . . 5.10, 5.11, 5.12
  SIN . . . . . . . . . . . . . . . . . 5.10, 5.11, 5.12
  SIZE . . . . . . . . . . . . . . . . . . . . . . . . 6.7.4
  SQR . . . . . . . . . . . . . 5.2, 5.10, 5.11, 5.12
  SQRT . . . . . . . . . . . . . . . 5.10, 5.11, 5.12
  STATIONID . . . . . . . . . . . . . . . . . 15.3.2.2
  STATUS . . . . . . . . . . . . . . . . . . . . . . . 6.5
  SUCC . . . . . . . . . . . . . . . . . . 2.2.1.1, 5.7.1
  TASKID . . . . . . . . . . . . . . . . . . . . 15.3.2.1
  TRUNC . . . . . . . . . . . 5.2, 5.10, 5.11, 5.12
  UB . . . . . . . . . . . . . . . . . . . 6.1, 6.4, 6.7.1
  $TB . . . . . . . . . . . . . . . . . . . . . . . . 15.2.5
Functions:
  DECIMAL Type . . . . . . . . . . . . . . . . . 5.12
  Enumeration Type . . . . . . . . . . . . . . 2.2.1.1
  FIXED Type . . . . . . . . . . . . . . . . . . . . 5.11
  Identification . . . . . . . . . . . . . . . . . . 15.3.2
  INTEGER Type . . . . . . . . . . . . . . . . . . . 5.2
  LONGINT Type . . . . . . . . . . . . . . . . . . 5.2
  REAL Type . . . . . . . . . . . . . . . . . . . . 5.10
  Standard . . . . . . . . . . . . . . . . . . . . . . . A.2
  User Defined . . . . . . . . . . . . . . . . . . . . 8.8

GENERATE Example Program . . . . . . . FB-4
GETMSG Procedure . . . . . . . . . . . . . 15.3.5.2
Global Common Blocks . . . . . . . . . . . . . 15.6
GLOBALS Option . . . . . . . . . . . . . . . 9.6.3
GOTO Statement . . . . . . . . . . . . . . . . . . 7.2
Guidelines:
  Separate Compilation . . . . . . . . . . . 12.3.4.1
  Source Code . . . . . . . . . . . . . . . . . . . 10.2

HALT Procedure . . . . . . . . . . . . . . . . . A.3
Heap Manager Option . . . . . . . . . . . . 14.3.11
Hexadecimal:
  Constant . . . . . . . . . . . . . . . . . . . . . . . 3.2
  I/O . . . . . . . . . . . . . . . . . . . . . . . . . 6.5.3

IDATE Procedure . . . . . . . . . . . . . . . 15.3.3.2
Identification Functions . . . . . . . . . . . . 15.3.2
Identifiers . . . . . . . . . . . . . . . . . . . . . . 3.2.1
IF Statement . . . . . . . . . . . . . . . . . . . 5.6.1
Implicit Type Conversions . . . . . . . . . . . . I.2
IN Operator . . . . . . . . . . . . . . . . . . 4.2, 6.4
Information, DX10 . . . . . . . . . . . . . . . . C.1
Initial Run, OUTPUT File:
  Full Compilation . . . . . . . . . . . . . . . F12-2
  Partial Compilation . . . . . . . . . . . . . F12-5
Input:
  Example . . . . . . . . . . . . . . 12.4.2, F12-11
  File . . . . . . . . . . . . . . . . . . . . . . . . . 6.5
INSERT Command . . . . . . . . . . . . . . 12.3.9.2
Installation, Program . . . . . . . . . . . . . . . H.5
Instructions, Floating Point . . . . . . . . . . TE-1
INTEGER:
  Constant . . . . . . . . . . . . . . . . . . . . . . . 3.2
    Expressions . . . . . . . . . . . . . . . . . . . . 5.9
  Type . . . . . . . . . . . . . . . . . . . . . 5.2, 6.7.3
    Conversion To . . . . . . . . . . . . . . . . . I.3
    Functions . . . . . . . . . . . . . . . . . . . . . 5.2
    Operators . . . . . . . . . . . . . . . . . . . . . 5.2
INTEGER-DECIMAL Type Conversion . . I.7.1
INTEGER-FIXED Type Conversion . . . . . I.6.1

INTEGER-LONGINT Type Conversion . . I.4.1
INTEGER-REAL Type Conversion . . . . . I.5.1
Integers . . . . . 3.2.2
Interactive Mode, NESTER . . . . . 10.3.3
Internal Representation of Types . . . . . 6.7.3
Introduction . . . . . 1.1
I/O:
  DX10 Device . . . . . C.3
  Errors . . . . . 14.7.2
  Formatted . . . . . 6.5.3
  Hexadecimal . . . . . 6.5.3
  Routines . . . . . 15.3.1
  Stand Alone . . . . . 15.7
I/O Operation Type Conversions . . . . . I.2.3
IOTERM Procedure . . . . . 6.5
ITIME Procedure . . . . . 15.3.3.1

LABEL:
  Declaration . . . . . 7.2, 8.2.1
  Escape . . . . . 7.1, 8.2
LBP Command . . . . . 14.6.1.5
Level, Static Nesting . . . . . F.2
Libraries:
  CONFIG . . . . . 12.3.8
  Configuration Processor . . . . . 12.3.8
  Runtime . . . . . 14.2, T14-1
LIBRARY:
  Command . . . . . 12.3.8.2
  Split Program . . . . . 12.4.3
  SPLITPGM . . . . . 12.4.3
Line Width, Source Listing . . . . . 9.3.2
Linking:
  DX10 LUNO . . . . . 14.3.3
  DX10 With SCI . . . . . 14.3.1
  Minimal Runtime . . . . . 14.3.11
  Overlay . . . . . H.4
  RX990 . . . . . 14.3.9
  Shared Procedure . . . . . 14.3.15
  Stand-Alone . . . . . 14.3.13
  TIP . . . . . 14.1, 14.3
  TX990 . . . . . 14.3.5
    Multiple Task . . . . . 14.3.7, 14.3.8
    Overlay . . . . . H.6
Linking Map Example . . . . . F14-3
LINT Function . . . . . 5.2
LIST:
  Command . . . . . 12.3.5.1
  Option . . . . . 9.3.1
List Breakpoints-Pascal Command . . . . 14.6.1.5
List Pascal Stack Command . . . . . 14.6.1.7
LISTDOC:
  Command . . . . . 12.3.5.2
  Example . . . . . F12-9
Listing Example, Error . . . . . F11-3
LISTORDER Command . . . . . 12.3.5.3
LN Function . . . . . 5.10, 5.11, 5.12
Local Common Blocks . . . . . 15.6
LOCATION Function . . . . . 5.2, 6.6
Logical Operators . . . . . 4.2
LOGMSG Option . . . . . 14.3.11
LONGINT Type . . . . . 5.2, 6.7.3
  Conversion To . . . . . I.4
  Functions . . . . . 5.2
  Operators . . . . . 5.2

LONGINT-DECIMAL Type Conversion . . I.7.2
LONGINT-FIXED Type Conversion . . . . I.6.2
LONGINT-INTEGER Type Conversion . . I.3.1
LONGINT-REAL Type Conversion . . . . . I.5.2
LPS Command . . . . . 14.6.1.7
LROUND Function . . . . . A.2
LTRUNC Function . . . . . A.2
LUNO For OUTPUT, Changing . . . . . 15.5
LUNO I/O:
  Option . . . . . 14.2.2
  Program Considerations . . . . . 15.5
  Termination Routines . . . . . F.7.2
LU$MSG Module . . . . . 15.5

Main Routine, Dummy . . . . . 15.4
Manifest Constant . . . . . 5.7.3, 5.8.1, 6.1.2, 6.4, 8.5.2
Manual Organization . . . . . 1.5
MAP Option . . . . . 9.3.3, F9-1
MASTER Command . . . . . 12.3.8.1
Mathematical Function Type
  Conversions . . . . . I.2.4
Medium Category Routines . . . . . F.3
Memory Space Errors . . . . . 14.7.3
Message Handling Procedures . . . . . 15.3.5
MESSAGE Procedure . . . . . A.3
Messages:
  Compiler Error . . . . . E.1
  Debug Command Error . . . . . E.4
  Error . . . . . Appendix E
  NESTER Error . . . . . 10.3.6
  Runtime Error . . . . . E.2
Minimal Runtime:
  Execution . . . . . 14.3.12
  Linking . . . . . 14.3.11
  Option . . . . . 14.2.7
  Program Considerations . . . . . 15.8
Mixed Mode Expressions . . . . . 5.9
MOD Operator . . . . . 4.2, 5.2
Mode:
  Debug . . . . . 14.6
  NESTER Interactive . . . . . 10.3.3
Modification, Process Configuration . . . . 12.3.7
Modifications, Pascal . . . . . 1.2
Module:
  Format, Source . . . . . 12.3.2
  LU$MSG . . . . . 15.5
  Routine . . . . . F.5
  Standard Category Routine . . . . . FF-2
MOVE Command . . . . . 12.3.7.2
Multidimensional Arrays . . . . . 6.1
Multiple Task:
  Linking, TX990 . . . . . 14.3.7, 14.3.8
  Program Considerations . . . . . 15.6
Multiplying Operators . . . . . 4.2
Multitask DX10 Option . . . . . 14.2.3
Multitask TX990 Option . . . . . 14.2.5

NESTER . . . . . 10.3
  Commands . . . . . 10.3.4
  Error Messages . . . . . 10.3.6
  Errors . . . . . T10-2
  Example . . . . . 10.3.7, F10-2
  Execution . . . . . 10.3.5

Interactive Mode . . . . . . . . . . . . . . . 10.3.3
Option Comment . . . . . . . . . . . . . . . 10.3.2
Options . . . . . . . . . . . . . . . . . . . . . T10-1
Rules . . . . . . . . . . . . . . . . . . . . . . 10.3.1
Statement Example . . . . . . . . . . . . . . F10-1
Nesting Level, Static . . . . . . . . . . . . . . F.2
NEW Procedure . . . . . . . . . . . . . . 6.6.1, 8.7
Nonsharable Runtime Library
Routines . . . . . . . . . . . . . . . . . . . . . T14-2
NOT Operator . . . . . . . . . . . . . . . . . 4.2, 5.3
Notation, Syntax . . . . . . . . . . . . . . . . . . 3.1
NULLBODY Option . . . . . . . . . . . . . . 9.4.2
Numbers . . . . . . . . . . . . . . . . . . . . . 3.2.2
Overlay . . . . . . . . . . . . . . . . . . . . . . H.7
Real . . . . . . . . . . . . . . . . . . . . . . . 3.2.2

OBJECT Option . . . . . . . . . . . . . . . . . 9.4.1
OBJLIB Command . . . . . . . . . . . . . 12.3.8.3
ODD Function . . . . . . . . . . . . . . . . . . . . 5.2
Operator:
AND . . . . . . . . . . . . . . . . . . . . . . 4.2, 5.3
DIV . . . . . . . . . . . . . . . . . . . . . . . 4.2, 5.2
IN . . . . . . . . . . . . . . . . . . . . . . . . 4.2, 6.4
MOD . . . . . . . . . . . . . . . . . . . . . . 4.2, 5.2
NOT . . . . . . . . . . . . . . . . . . . . . . 4.2, 5.3
OR . . . . . . . . . . . . . . . . . . . . . . . 4.2, 5.3
Operators . . . . . . . . . . . . . . . . . . . . . . . 4.2
Adding . . . . . . . . . . . . . . . . . . . . . . . 4.2
BOOLEAN Type . . . . . . . . . . . . . . . . . 5.3
DECIMAL Type . . . . . . . . . . . . . . . . 5.12
FIXED Type . . . . . . . . . . . . . . . . . . . 5.11
INTEGER Type . . . . . . . . . . . . . . . . . . 5.2
Logical . . . . . . . . . . . . . . . . . . . . . . . 4.2
LONGINT Type . . . . . . . . . . . . . . . . . 5.2
Multiplying . . . . . . . . . . . . . . . . . . . . 4.2
Pointer Type . . . . . . . . . . . . . . . . . . . 6.6
REAL Type . . . . . . . . . . . . . . . . . . . 5.10
Relational . . . . 2.2.1.1, 4.2, 5.11, 5.12, 6.7.1
SET Type . . . . . . . . . . . . . . . . . . . . . 6.4
String . . . . . . . . . . . . . . . . . . . . . . 6.7.1
Option:
ASSERTS . . . . . . . . . . . . . . . . . . . . 9.4.4
CKINDEX . . . . . . . . . . . . . . . . . . . 9.5.1
CKOVER . . . . . . . . . . . . . . . . . . . . 9.5.2
CKPREC . . . . . . . . . . . . . . . . . . . . 9.5.3
CKPTR . . . . . . . . . . . . . . . . . . . . . 9.5.4
CKSET . . . . . . . . . . . . . . . . . . . . . 9.5.5
CKSUB . . . . . . . . . . . . . . . . . . . . . 9.5.6
CKTAG . . . . . . . . . . . . . . . . . . . . . 9.5.7
Comment . . . . . . . . . . . . . . . . . . . . . 9.2
NESTER . . . . . . . . . . . . . . . . . . . 10.3.2
Compiler . . . . . . . . . . . . . . . . . . 4.1, 6.2.2
Display:
PROBER . . . . . . . . . . . . . . . . . . . . F9-2
PROBES . . . . . . . . . . . . . . . . . . . . F9-2
DX10 Execution . . . . . . . . . . . . . . . 14.2.1
ENDPROG . . . . . . . . . . . . . . . . . . 14.3.5
Example:
PROBER . . . . . . . . . . . . . . . . . . . . F9-3
PROBES . . . . . . . . . . . . . . . . . . . . F9-3
FORINDEX . . . . . . . . . . . . . . . . . . 9.6.2
GLOBALS . . . . . . . . . . . . . . . . . . . 9.6.3
Heap Manager . . . . . . . . . . . . . . . 14.3.11
LIST . . . . . . . . . . . . . . . . . . . . . . . 9.3.1
LUNO I/O . . . . . . . . . . . . . . . . . . . 14.2.2
MAP . . . . . . . . . . . . . . . . . . . 9.3.3, F9-1
Minimal Runtime . . . . . . . . . . . . . . 14.2.7
Multitask DX10 . . . . . . . . . . . . . . . 14.2.3
Multitask TX990 . . . . . . . . . . . . . . 14.2.5
NULLBODY . . . . . . . . . . . . . . . . . . 9.4.2
OBJECT . . . . . . . . . . . . . . . . . . . . 9.4.1
PAGE . . . . . . . . . . . . . . . . . . . . . . 9.3.4
PROBER . . . . . . . . . . . . . . . 9.5.8, 14.3.1
PROBES . . . . . . . . . . . . . . . 9.5.9, 14.3.1
ROUND . . . . . . . . . . . . . . . . . . . . . 9.6.4
RX990 Execution . . . . . . . . . . . . . . 14.2.6
Shared Procedure . . . . . . . . . . . . . . 14.2.9
Stand-Alone Execution . . . . . . . . . . 14.2.8
TRACEBACK . . . . . . . . . . . . . . . . . 9.4.3
TX990 Execution . . . . . . . . . . . . . . 14.2.4
UNSAFEOPT . . . . . . . . . . . . . . . . . 9.6.5
WARNINGS . . . . . . . . . . . . . . . . . . 9.3.5
WIDELIST . . . . . . . . . . . . . . . 9.3.2, F9-1
72COL . . . . . . . . . . . . . . . . . . . . . . 9.6.1
Options:
Compiler . . . . . . . . . . . . . . . . . . . 9.1, T9-1
NESTER . . . . . . . . . . . . . . . . . . . . T10-1
Program Level . . . . . . . . . . . . . . . . . . 9.2
Routine Level . . . . . . . . . . . . . . . . . . 9.2
Runtime . . . . . . . . . . . . . . . . . . . . . . 14.2
Statement Level . . . . . . . . . . . . . . . . . 9.2
OR Operator . . . . . . . . . . . . . . . . . 4.2, 5.3
ORD Function . . . . . . . . . 2.2.1.1, 5.7.1, 5.7.2
Organization, Manual . . . . . . . . . . . . . . . 1.5
OUTPUT, Changing LUNO For . . . . . . . 15.5
OUTPUT File . . . . . . . . . . . . . . . . . . . . 6.5
Full Compilation:
Deferred Processing . . . . . . . . . . . . F12-4
Initial Run . . . . . . . . . . . . . . . . . . F12-2
Partial Compilation:
Deferred Processing . . . . . . . . . . . . F12-7
Initial Run . . . . . . . . . . . . . . . . . . F12-5
Overflow, Symbol Table . . . . . . . . . . . 14.4.1
Overlay:
Linking . . . . . . . . . . . . . . . . . . . . . . H.4
TX990 . . . . . . . . . . . . . . . . . . . . . . H.6
Numbers . . . . . . . . . . . . . . . . . . . . . H.7
Structure, TIP . . . . . . . . . . . . . . . . . H.2
Overlays, TIP . . . . . . . . . . . . . . . . . . . H.1
OVLY$ Procedure . . . . . . . . . . . . . . . . H.3

PACK Procedure . . . . . . . . . . . . . . . . 6.1.1
Packed:
Array . . . . . . . . . . . . . . . . . . . . . . 6.7.1
Data Type . . . . . . . . . . . . . . . . . . . . 6.7
Records . . . . . . . . . . . . . . . . . . . . 6.7.2
PAGE:
Option . . . . . . . . . . . . . . . . . . . . . 9.3.4
Procedure . . . . . . . . . . . . . . . . 6.5.2, 6.5.6
Parameter Rules . . . . . . . . . . . . . . . . 8.5.1
Parameters . . . . . . . . . . . . . . . . . . . . . 8.5
Dynamic:
Array . . . . . . . . . . . . . . . . . . . . . 8.5.2
Set . . . . . . . . . . . . . . . . . . . . . . . 8.5.2
Function . . . . . . . . . . . . . . . . . . . . 8.5.3
Procedure . . . . . . . . . . . . . . . . . . . 8.5.3
Program . . . . . . . . . . . . . . . . . . . . . . 2.1

Reference . . . . . . . . . . . . . . . . . . . . . . 8.5
Routines Without . . . . . . . . . . . . . . . 8.5.4
Value . . . . . . . . . . . . . . . . . . . . . . . . . 8.5
Partial Compilation:
Deferred Processing, OUTPUT File . . . F12-7
Initial Run, OUTPUT File . . . . . . . . F12-5
Source Listing . . . . . . . . . . . . . . . . . F12-6
Pascal:
Extensions . . . . . . . . . . . . . . . . . . . . . . 1.2
Modifications . . . . . . . . . . . . . . . . . . . 1.2
PBP Command . . . . . . . . . . . . . . . . 14.6.1.4
Phases, Compiler . . . . . . . . . . . . . . . . . 11.1
PL/I Comparisons . . . . . . . . . . . . . . . . . 1.4
Pointer Type . . . . . . . . . . . . . . . . 6.6, 6.7.3
Operators . . . . . . . . . . . . . . . . . . . . . . 6.6
Portion, Declaration . . . . . . . . . . . . . . . . 8.2
Precision Conversion:
DECIMAL Type . . . . . . . . . . I.2.2.2, I.7.5
FIXED Type . . . . . . . . . . . . I.2.2.2, I.6.5
REAL Type . . . . . . . . . . . . I.2.2.1, I.5.5
PRED Function . . . . . . . . . . . . 2.2.1.1, 5.7.1
Preparation, Source Code . . . . . . . . . . . . 15.1
PRGMSG Procedure . . . . . . . . . . . . . 15.3.5.3
PROBER Option . . . . . . . . . . . . 9.5.8, 14.3.1
Display . . . . . . . . . . . . . . . . . . . . . . F9-2
Example . . . . . . . . . . . . . . . . . . . . . F9-3
PROBES Option . . . . . . . . . . . . 9.5.9, 14.3.1
Display . . . . . . . . . . . . . . . . . . . . . . F9-2
Example . . . . . . . . . . . . . . . . . . . . . F9-3
Procedure:
ACTIVATE . . . . . . . . . . . . . . . . . 15.3.4.3
BID . . . . . . . . . . . . . . . . . . . . . . 15.3.4.1
Call . . . . . . . . . . . . . . . . . . . . . . . . . 8.4
CLOSE . . . . . . . . . . . . . . . . . . . . . . . 6.5
DATE . . . . . . . . . . . . . . . . . . . . . . . A.3
DECODE . . . . . . . . . . . . . . . . 6.5.4, 6.5.6
DELAY . . . . . . . . . . . . . . . . . . . 15.3.3.3
DISPOSE . . . . . . . . . . . . . . . . . 6.6.1, 8.7
DSPLY$ . . . . . . . . . . . . . . . . . . . . . 15.7
ENCODE . . . . . . . . . . . . . . . . 6.5.4, 6.5.6
EXTEND . . . . . . . . . . . . . 6.5, 6.5.5, 14.3.1
FIND$SYN . . . . . . . . . . . . . . . . . 15.3.1.1
GETMSG . . . . . . . . . . . . . . . . . . 15.3.5.2
HALT . . . . . . . . . . . . . . . . . . . . . . . A.3
IDATE . . . . . . . . . . . . . . . . . . . . 15.3.3.2
IOTERM . . . . . . . . . . . . . . . . . . . . . . 6.5
ITIME . . . . . . . . . . . . . . . . . . . . 15.3.3.1
MESSAGE . . . . . . . . . . . . . . . . . . . . A.3
NEW . . . . . . . . . . . . . . . . . . . . 6.6.1, 8.7
OVLY$ . . . . . . . . . . . . . . . . . . . . . . . H.3
PACK . . . . . . . . . . . . . . . . . . . . . . 6.1.1
PAGE . . . . . . . . . . . . . . . . . . 6.5.2, 6.5.6
Parameters . . . . . . . . . . . . . . . . . . . 8.5.3
PRGMSG . . . . . . . . . . . . . . . . . . 15.3.5.3
PUTMSG . . . . . . . . . . . . . . . . . . 15.3.5.1
P$SYN . . . . . . . . . . . . . . . . . . . . . 14.4.1
Q$SYN . . . . . . . . . . . . . . . . . . . . . 14.4.1
READ . . . . . . . . . . . 6.5, 6.5.1, 6.5.2, 6.5.3,
6.5.5, 6.5.6
READLN . . . . . . . . . 6.5, 6.5.2, 6.5.3, 6.5.6
RESET . . . . . . . . . . . 6.5, 6.5.1, 6.5.2, 6.5.5
RESETSEMAPHORE . . . . . . . . . . . 15.3.7.1
REWRITE . . . . . . . . . . . . . . . . 6.5, 6.5.5
RLSCOM . . . . . . . . . . . . . . . . . . . 15.3.6.2
SETLUNO . . . . . . . . . . . 6.5, 15.3.1.4, 15.5
SETMEMBER . . . . . . . . . . . . . . . . . . . 6.5
SETNAME . . . . . . . . . . . . . . . . . . 6.5, 15.5
SET$ACNM . . . . . . . . . . . . . . . . . 15.3.1.3
SKIPFILES . . . . . . . . . . . . . . . . . . . 6.5.1
STORE$SYN . . . . . . . . . . . . . . . . 15.3.1.2
Supervisor Call . . . . . . . . . . . . . . . . 15.3.8
SUSPEND . . . . . . . . . . . . . . . . . . 15.3.4.2
SVC$ . . . . . . . . . . . . . . . . . . . . . . . 15.3.8
SYSCOM . . . . . . . . . . . . . . . . . . . 15.3.6.1
TESTANDSET . . . . . . . . . . . . . . . 15.3.7.2
TIME . . . . . . . . . . . . . . . . . . . . . . . . . A.3
UNPACK . . . . . . . . . . . . . . . . . . . . . 6.1.1
WRITE . . . . . . . . . . 6.5, 6.5.1, 6.5.2, 6.5.3,
6.5.5, 6.5.6
WRITEEOF . . . . . . . . . . . . . . . . . . . 6.5.1
WRITELN . . . . . . . . . . . . . 6.5, 6.5.2, 6.5.6
$LDCR . . . . . . . . . . . . . . . . . . . . . . 15.2.1
$SBO . . . . . . . . . . . . . . . . . . . . . . . 15.2.2
$SBZ . . . . . . . . . . . . . . . . . . . . . . . 15.2.3
$STCR . . . . . . . . . . . . . . . . . . . . . . 15.2.4
Procedures:
Array Type . . . . . . . . . . . . . . . . . . . . 6.1.1
Date . . . . . . . . . . . . . . . . . . . . . . . 15.3.3
Message Handling . . . . . . . . . . . . . . 15.3.5
Semaphore . . . . . . . . . . . . . . . . . . . 15.3.7
Standard . . . . . . . . . . . . . . . . . . . . . . A.3
System Common Access . . . . . . . . . . 15.3.6
Task Control . . . . . . . . . . . . . . . . . 15.3.4
Time . . . . . . . . . . . . . . . . . . . . . . . 15.3.3
Proceed From Breakpoint-Pascal
Command . . . . . . . . . . . . . . . . . . 14.6.1.4
Process Configuration . . . . . . . . . . . . . 12.3.3
Command Example . . . . . . . . . . . 12.3.3.4
Flags . . . . . . . . . . . . . . . . . . 12.3.6, T12-1
Modification . . . . . . . . . . . . . . . . . 12.3.7
Process Record Fields . . . . . . . . . . . . . 14.7.4
Productions, BNF . . . . . . . . . . . Appendix D
Program:
Example . . . . . . . . . . . . . . . . . . . 2.5, F2-1
Format . . . . . . . . . . . . . . . . . . . . . . . . 8.1
GENERATE Example . . . . . . . . . . . . FB-4
Installation . . . . . . . . . . . . . . . . . . . . H.5
Parameters . . . . . . . . . . . . . . . . . . . . . 2.1
QUAD Example . . . . . . . . . . . . . . . FB-2
RANDOM Example . . . . . . . . . . . . . FB-3
ROOTS Example . . . . . . . . . . . . . . . FB-1
Structure . . . . . . . . . . . . . . . . . . . . . . 2.1
Program Considerations:
LUNO I/O . . . . . . . . . . . . . . . . . . . . 15.5
Minimal Runtime . . . . . . . . . . . . . . . . 15.8
Multiple Task . . . . . . . . . . . . . . . . . . 15.6
Stand Alone . . . . . . . . . . . . . . . . . . . 15.7
Program Level Options . . . . . . . . . . . . . . 9.2
Programs, Example . . . . . . . . . . Appendix B
PUTMSG Procedure . . . . . . . . . . . . 15.3.5.1
P$SYN Procedure . . . . . . . . . . . . . . . 14.4.1

QUAD Example Program . . . . . . . . . . . FB-2
Q$SYN Procedure . . . . . . . . . . . . . . . 14.4.1

RANDOM:
  Example Program . . . . . . . . . . . . . . FB-3
  Files . . . . . . . . . . . . . . . . . . . . . 6.5, 6.5.5
    DX10 . . . . . . . . . . . . . . . . . . . . C.2.3
RASS . . . . . . . . . . . . . . . . . . . . . . 13.1
  Example . . . . . . . . . . . . . . . . 13.4, F13-1
  Execution . . . . . . . . . . . . . . . . . . . . 13.3
  Files . . . . . . . . . . . . . . . . . . . . . . 13.2
READ Procedure . . . . .6.5, 6.5.1, 6.5.2, 6.5.3, 6.5.5, 6.5.6
READLN Procedure . . . 6.5, 6.5.2, 6.5.3, 6.5.6
REAL:
  Constant . . . . . . . . . . . . . . . . . . . . . .3.2
  Numbers . . . . . . . . . . . . . . . . . . . . 3.2.2
  Type . . . . . . . . . . . . . . . . . . . .5.10, 6.7.3
    Conversion To . . . . . . . . . . . . . . . . . I.5
    Functions . . . . . . . . . . . . . . . . . . . . 5.10
    Operators . . . . . . . . . . . . . . . . . . . . 5.10
REAL Type Precision
  Conversion . . . . . . . . . . . . . . I.2.2.1, I.5.5
REAL-DECIMAL Type Conversion . . . . . I.7.3
REAL-FIXED Type Conversion . . . . . . . I.6.3
REAL-INTEGER Type Conversion . . . . . I.3.2
REAL-LONGINT Type Conversion . . . . . I.4.2
Record Type . . . . . . . . . . . . 6.2, 6.2.1, 6.7.3
Records, Packed . . . . . . . . . . . . . . . . . 6.7.2
Recursive Routine . . . . . . . . . . . . . . . . . .8.7
Reference Parameters . . . . . . . . . . . . . . . .8.5
References, Shared Procedure . . . . . . . . . 15.6
Related Documents . . . . . . . . . . . . . . Preface
Relational Operators . . . . . . . . . . 2.2.1.1, 4.2, 5.11, 5.12, 6.7.1
REPEAT Statement . . . . . . . . . . . . . . . 5.6.3
Repetitive Statements . . . . . . . . . . . . . . . .5.6
REPLACE Command . . . . . . . . . . . .12.3.9.3
Representation of Types, Internal . . . . . . 6.7.3
Requirements:
  Separate Compilation . . . . . . . . . . . . . 12.2
  Storage . . . . . . . . . . . . . . . . . . . . . . 2.2.3
Reserved Words . . . . . . . . . . . . . . . . . . .3.2
RESET Procedure . . . . . 6.5, 6.5.1, 6.5.2, 6.5.5
RESETSEMAPHORE Procedure . . . . .15.3.7.1
Reverse Assembler . . . . . . . . . . . . . . . . . 13.1
  Example . . . . . . . . . . . . . . . . 13.4, F13-1
  Execution . . . . . . . . . . . . . . . . . . . . 13.3
  Files . . . . . . . . . . . . . . . . . . . . . . 13.2
REWRITE Procedure . . . . . . . . . . 6.5, 6.5.5
RLSCOM Procedure . . . . . . . . . . . . . .15.3.6.2
ROOTS Example Program . . . . . . . . . . . FB-1
ROUND:
  Function . . . . . . . . . . . . . 5.10, 5.11, 5.12
  Option . . . . . . . . . . . . . . . . . . . . . . 9.6.4
Routine:
  Access . . . . . . . . . . . . . . . . . . . . . . .F8-5
  Calling . . . . . . . . . . . . . . . . . . . . . . F.5.2
  Categories . . . . . . . . . . . . . . . . . . . . . F.3
  Declaration . . . . . . . . . . . . . . . . . . . 8.2.7
  Dummy Main . . . . . . . . . . . . . . . . . . . 15.4
  Level Options . . . . . . . . . . . . . . . . . . .9.2
  Module . . . . . . . . . . . . . . . . . . . . . . . F.5
    Standard Category . . . . . . . . . . . . . FF-2
  Recursive . . . . . . . . . . . . . . . . . . . . . .8.7
  Scope . . . . . . . . . . . . . . . . . . . . . . . .2.1
  Structure . . . . . . . . . . . . . . . . . . . . . .2.1
  Syntax Diagram . . . . . . . . . . . . . . . . .F8-2
  Workspace . . . . . . . . . . . . . . . . . . . F.4.1
Routines:
  Assembly Language . . . . . . . . . . . F.1, F.6
  DX10 CRU . . . . . . . . . . . . . . . . . . . . 15.2
  I/O . . . . . . . . . . . . . . . . . . . . . . . . .15.3.1
  LUNO I/O Termination . . . . . . . . . . F.7.2
  Medium Category . . . . . . . . . . . . . . . . F.3
  Nonsharable Runtime Library . . . . . . T14-2
  Short Category . . . . . . . . . . . . . . . . . F.3
  Standard . . . . . . . . . . . . . . . . Appendix A
    Category . . . . . . . . . . . . . . . . . . . . F.3
    Termination . . . . . . . . . . . . . . . . . F.7.1
  System Interface . . . . . . . . . . . . . . . . 15.3
  User Termination . . . . . . . . . . . . . . . . F.7
  Without Parameters . . . . . . . . . . . . . 8.5.4
Rules:
  NESTER . . . . . . . . . . . . . . . . . . . .10.3.1
  Parameter . . . . . . . . . . . . . . . . . . . . 8.5.1
Runtime:
  Checks . . . . . . . . . . . . . . . . 9.5, 14.7.1
  Error:
    Codes . . . . . . . . . . . . . . . . . . . . . . E.3
    Messages . . . . . . . . . . . . . . . . . . . . E.2
  Errors . . . . . . . . . . . . . . . . . . . . . . . 14.7
  Libraries . . . . . . . . . . . . . . . . 14.2, T14-1
  Library Routines, Nonsharable . . . . . . T14-2
  Options . . . . . . . . . . . . . . . . . . . . . . 14.2
  Size . . . . . . . . . . . . . . . . . 14.2, C.4, TC-1
RX990:
  Execution . . . . . . . . . . . . . . . . . . . 14.3.10
  Execution Option . . . . . . . . . . . . . . . 14.2.6
  Linking . . . . . . . . . . . . . . . . . . . . . 14.3.9

Scalar Type . . . . . . . . . . . . . . . . 5.7.2, 6.7.3
Scope . . . . . . . . . . . . . . . . . . . . . . . . . .8.7
  Data . . . . . . . . . . . . . . . . . . . . . . . . .2.1
  Routine . . . . . . . . . . . . . . . . . . . . . . .2.1
Section, Documentation . . . . . . . 12.3.2, 12.3.5
Semaphore:
  Example . . . . . . . . . . . . . . . . . . . . . F15-1
  Procedures . . . . . . . . . . . . . . . . . . . 15.3.7
Separate Compilation . . . . . . . . . . . 12.1, 12.3.4
  Examples . . . . . . . . . . . . . . . . . . . .12.3.4.4
  Flowchart . . . . . . . . . . . . . . . . . . . . F12-1
  Guidelines . . . . . . . . . . . . . . . . . . .12.3.4.1
  Requirements . . . . . . . . . . . . . . . . . . . 12.2
Separators . . . . . . . . . . . . . . . . . . . . . . . .3.3
Sequential Files . . . . . . . . . . . . . . . 6.5, 6.5.1
  DX10 . . . . . . . . . . . . . . . . . . . . . . . C.2.1
Set:
  ASCII Character . . . . . . Appendix G, TG-1
  Character . . . . . . . . . . . . . . . . . . . . . .3.2
  Parameters, Dynamic . . . . . . . . . . . . 8.5.2
  Type . . . . . . . . . . . . . . . . . . . . . 6.4, 6.7.3
    Operators . . . . . . . . . . . . . . . . . . . . . .6.4
SETFLAG Command . . . . . . . . . . . . .12.3.6.1
SETLIB Command . . . . . . . . . . . . . . .12.3.8.5
SETLUNO Procedure . . . . .6.5, 15.3.1.4, 15.5
SETMEMBER Procedure . . . . . . . . . . . . .6.5
SETNAME Procedure . . . . . . . . . . . 6.5, 15.5
Sets, Dynamic . . . . . . . . . . . . . . . . . . . . .6.4

SET$ACNM Procedure . . . . . . . . . . . .15.3.1.3
Shared Procedure:
  Linking . . . . . . . . . . . . . . . . . . . . . 14.3.15
  Option . . . . . . . . . . . . . . . . . . . . . .14.2.9
  References . . . . . . . . . . . . . . . . . . . . 15.6
Short:
  Category Routines . . . . . . . . . . . . . . . F.3
  Circuit Evaluation . . . . . . . . . . . . . . . .5.9
Show Pascal Stack Command . . . . . . .14.6.1.6
SHRPROC File . . . . . . . . . . . . . . . . 14.3.15
Side Effects . . . . . . . . . . . . . . . . . . . . .8.8
SILT1 . . . . . . . . . . . . . . . . . . . . . . . . . 11.2
SILT2 . . . . . . . . . . . . . . . . . . . . . . . . . 11.3
Simple Types . . . . . . . . . . . . . . . . . . 2.2.1
SIN Function . . . . . . . . . . . . 5.10, 5.11, 5.12
Size:
  Function . . . . . . . . . . . . . . . . . . . . . 6.7.4
  Runtime . . . . . . . . . . . . . . 14.2, C.4, TC-1
SKIPFILES Procedure . . . . . . . . . . . . . 6.5.1
Source Code:
  Guidelines . . . . . . . . . . . . . . . . . . . . . 10.2
  Preparation . . . . . . . . . . . . . . . . . . . . 15.1
Source Listing . . . . . . . . . . . . . . . . . . 12.3.5
  Example . . . . . . . . F9-1, F9-3, F11-1, F12-8
  Examples . . . . . . . . . . . . . . . . . . .12.3.5.4
  Full Compilation . . . . . . . . . . . . . . . F12-3
  Line Width . . . . . . . . . . . . . . . . . . . 9.3.2
  Partial Compilation . . . . . . . . . . . . . F12-6
Source Module Format . . . . . . . . . . . . 12.3.2
Space:
  Errors, Memory . . . . . . . . . . . . . . . . 14.7.3
  Stack . . . . . . . . . . . . . . . . . . . . . . . . 11.5
SPLIT Execution . . . . . . . . . . . . . . . . . 12.5
SPLIT OBJECT:
  Command . . . . . . . . . . . . . . . . . .12.3.4.2
  Utility . . . . . . . . . . . . . . . . . . . . . . . 12.5
    Execution . . . . . . . . . . . . . . . . . . . . 12.5
Split Program:
  Command . . . . . . . . . . . . . . . . . . . . 12.4.1
  Execution . . . . . . . . . . . . . . . . . . . . 12.4.4
  Files . . . . . . . . . . . . . . . . . . . . . . . . 12.4.3
  Library . . . . . . . . . . . . . . . . . . . . . . 12.4.3
  Utility . . . . . . . . . . . . . . . . . . . . . . . 12.4
SPLIT Utility . . . . . . . . . . . . . . . . . . . . 12.5
SPLITPGM:
  Execution . . . . . . . . . . . . . . . . . . . . 12.4.4
    Batch . . . . . . . . . . . . . . . . . . . . . . 12.4.4
  Files . . . . . . . . . . . . . . . . . . . . . . . . 12.4.3
  Library . . . . . . . . . . . . . . . . . . . . . . 12.4.3
  Utility . . . . . . . . . . . . . . . . . . . . . . . 12.4
SPS Command . . . . . . . . . . . . . . . . .14.6.1.6
SQR Function . . . . . . . . 5.2, 5.10, 5.11, 5.12
SQRT Function . . . . . . . . . . 5.10, 5.11, 5.12
Stack . . . . . . . . . . . . . . . . . . . . . . . . . .8.7
Stack Frame . . . . . . . . . . . . . . . . . . 8.7, F.4
  Analysis . . . . . . . . . . . . . . . . . . . . . 14.7.5
  Data . . . . . . . . . . . . . . . . . . . . . . . . F.4.3
  Fields . . . . . . . . . . . . . . . . . . . . . . . 14.7.4
  Structure . . . . . . . . . . . . . . . . . . . . . FF-1
Stack Space . . . . . . . . . . . . . . . . . . . . . 11.5
Stand Alone:
  I/O . . . . . . . . . . . . . . . . . . . . . . . . 15.7
  Linking . . . . . . . . . . . . . . . . . . . . 14.3.14
  Program Considerations . . . . . . . . . . 15.7
Standard:
  Category:
    Routine Module . . . . . . . . . . . . . . . FF-2
    Routines . . . . . . . . . . . . . . . . . . . . . F.3
  Functions . . . . . . . . . . . . . . . . . . . . . . A.2
  Procedures . . . . . . . . . . . . . . . . . . . . . A.3
  Routines . . . . . . . . . . . . . . . . Appendix A
  Termination Routines . . . . . . . . . . . . F.7.1
Stand-Alone:
  Execution . . . . . . . . . . . . . . . . . . . 14.3.14
    Option . . . . . . . . . . . . . . . . . . . . . .14.2.8
  Linking . . . . . . . . . . . . . . . . . . . . . 14.3.13
Statement:
  ASSERT . . . . . . . . . . . . . . . . . . . . . . .5.4
  Assignment . . . . . . . . . . . . . . . . . . . . .4.1
  CASE . . . . . . . . . . . . . . . . . . . . 5.8, 5.8.1
  Compound . . . . . . . . . . . . . . . . . . . . .5.5
  Empty . . . . . . . . . . . . . . . . . . . . . . . . .5.5
  ESCAPE . . . . . . . . . . . . . . . . . . . . . . .7.1
  Example, NESTER . . . . . . . . . . . . . . F10-1
  FOR . . . . . . . . . . . . . . . . 5.8, 5.8.2, 8.2
  GOTO . . . . . . . . . . . . . . . . . . . . . . . .7.2
  IF . . . . . . . . . . . . . . . . . . . . . . . . . . 5.6.1
  REPEAT . . . . . . . . . . . . . . . . . . . . . 5.6.3
  Syntax Diagram . . . . . . . . . . . . . . . . .F8-3
  WHILE . . . . . . . . . . . . . . . . . . . . . . 5.6.2
  WITH . . . . . . . . . . . . . . . . . . . . . .6.3, 8.2
Statement Level Options . . . . . . . . . . . . . .9.2
Statements . . . . . . . . . . . . . . . . . . . .2.4, 8.3
  Conditional . . . . . . . . . . . . . . . . . . . . .5.6
  Repetitive . . . . . . . . . . . . . . . . . . . . . .5.6
Static Nesting Level . . . . . . . . . . . . . . . F.2
STATIONID Function . . . . . . . . . . . .15.3.2.2
STATUS Function . . . . . . . . . . . . . . . . . .6.5
Storage:
  Requirements . . . . . . . . . . . . . . . . . . 2.2.3
  System . . . . . . . . . . . . . . . . . . . . . . . F.4.2
STORE$SYN Procedure . . . . . . . . . . .15.3.1.2
Stream, DX10 TIP Batch . . . . . . . . . . . F14-1
String:
  Constants . . . . . . . . . . . . . . . . . . . . 3.2.3
  Operators . . . . . . . . . . . . . . . . . . . . 6.7.1
  Type . . . . . . . . . . . . . . . . . . . . . . . . 6.7.1
Structure:
  Block . . . . . . . . . . . . . . . . . . . . . . . . .8.6
  Program . . . . . . . . . . . . . . . . . . . . . . .2.1
  Routine . . . . . . . . . . . . . . . . . . . . . . .2.1
  Stack Frame . . . . . . . . . . . . . . . . . . FF-1
  TIP Overlay . . . . . . . . . . . . . . . . . . . H.2
Structured:
  Type Components . . . . . . . . . . . . . . 2.1.2
  Types . . . . . . . . . . . . . . . . . . . . . . . 2.1.2
Subrange Type . . . . . . . . . . . . . 5.7.3, 6.7.3
SUCC Function . . . . . . . . . . . . 2.2.1.1, 5.7.1
Supervisor Call Procedure . . . . . . . . . .15.3.8
SUSPEND Procedure . . . . . . . . . . . .15.3.4.2
SVC$ Procedure . . . . . . . . . . . . . . . 15.3.8
Symbols, Syntax Diagram . . . . . . . . . . F3-1
Synonym Table Overflow . . . . . . . . . . 14.4.1
Syntax . . . . . . . . . . . . . . . . . . Appendix D
  Diagram:
    Block . . . . . . . . . . . . . . . . . . . . . F8-1
    Example . . . . . . . . . . . . . . . . . . . . 3.1

Routine . . . . . . . . . . . . . . . . . . . . . F8-2
Statement . . . . . . . . . . . . . . . . . . . . F8-3
Symbols . . . . . . . . . . . . . . . . . . . . . F3-1
Diagrams . . . . . . . . . . . . 3.1, Appendix D
Expression . . . . . . . . . . . . . . . . . . . F5-1
Elements . . . . . . . . . . . . . . . . . . . . . . 3.2
Notation . . . . . . . . . . . . . . . . . . . . . . 3.1
SYSCOM Procedure . . . . . . . . . . . . . 15.3.6.1
SYSMSG Display, Compiler . . . . . . . . . F11-2
System Common Access Procedures . . . . 15.3.6
System Interface Routines . . . . . . . . . . . 15.3
System Storage . . . . . . . . . . . . . . . . . F.4.2

Table Overflow, Symbol . . . . . . . . . . . 14.4.1
Tag Field . . . . . . . . . . . . . . . . . . . . . 6.2.2
Task Control Procedures . . . . . . . . . . . 15.3.4
Task Execution Considerations . . . . . . . . 14.4
TASKID Function . . . . . . . . . . . . . 15.3.2.1
Termination:
Error . . . . . . . . . . . . . . . . . . . . . . . 14.7.4
Routines:
LUNO I/O . . . . . . . . . . . . . . . . . F.7.2
Standard . . . . . . . . . . . . . . . . . . F.7.1
User . . . . . . . . . . . . . . . . . . . . . . F.7
TESTANDSET Procedure . . . . . . . . . 15.3.7.2
Text Editing . . . . . . . . . . . . . . . . . . 12.3.9
TEXTFILE Error Codes . . . . . . . . . . . 14.7.2
Textfiles . . . . . . . . . . . . . . . 6.5, 6.5.2, 6.5.3
DX10 . . . . . . . . . . . . . . . . . . . . . . C.2.2
TIME:
Procedure . . . . . . . . . . . . . . . . . . . . A.3
Procedures . . . . . . . . . . . . . . . . . . 15.3.3
TIP:
Batch Stream, DX10 . . . . . . . . . . . . . F14-1
Execution, DX10 Batch . . . . . . . . . . . . 14.5
Files, DX10 . . . . . . . . . . . . . . . . . . . . C.2
Linking . . . . . . . . . . . . . . . . . . 14.1, 14.3
Overlay Structure . . . . . . . . . . . . . . . H.2
Overlays . . . . . . . . . . . . . . . . . . . . . H.1
TIP Program Execution . . . . . . . . . 14.1, 14.3
TRACEBACK Option . . . . . . . . . . . . . 9.4.3
Transfer, Type . . . . . . . . . . . . . . . . . . . 6.9
TRUNC Function . . . . . . 5.2, 5.10, 5.11, 5.12
TXXLE SCI Command . . . . . . . . . . . 14.3.5
TX990:
Abnormal Termination:
Dump . . . . . . . . . . . . . . . . . . . . . F14-5
Dump Analysis . . . . . . . . . . . . . . . 14.7.7
Configuration Module . . . . . . . . . . . . 14.3.7
Debugging . . . . . . . . . . . . . . . . . . . 14.6.2
Execution . . . . . . . . . . . . . . . . . . . 14.3.6
Option . . . . . . . . . . . . . . . . . . . . 14.2.4
Linking . . . . . . . . . . . . . . . . . . . . . 14.3.5
Multiple Task Linking . . . . . . 14.3.7, 14.3.8
Overlay Linking . . . . . . . . . . . . . . . . H.6
Type:
Array . . . . . . . . . . . . . . . . . . . 6.1, 6.7.3
BOOLEAN . . . . . . . . . . . . . . . . 5.3, 6.7.3
CHAR . . . . . . . . . . . . . . . . . 5.7.1, 6.7.3
Codes, DX10 Device . . . . . . . . . . . 15.3.1.5
Compatibility . . . . . . . . . . . . . . . . . . . 6.8
Components, Structured . . . . . . . . . . . 2.1.2
Concepts, Data . . . . . . . . . . . . . . . . . . 2.2
Conversion:
DECIMAL-FIXED . . . . . . . . . . . . . I.6.4
DECIMAL-INTEGER . . . . . . . . . . . I.3.4
DECIMAL-LONGINT . . . . . . . . . . . I.4.4
DECIMAL-REAL . . . . . . . . . . . . . . I.5.4
FIXED-DECIMAL . . . . . . . . . . . . . I.7.4
FIXED-INTEGER . . . . . . . . . . . . . I.3.3
FIXED-LONGINT . . . . . . . . . . . . . I.4.3
FIXED-REAL . . . . . . . . . . . . . . . . I.5.3
INTEGER-DECIMAL . . . . . . . . . . . I.7.1
INTEGER-FIXED . . . . . . . . . . . . . . I.6.1
INTEGER-LONGINT . . . . . . . . . . . I.4.1
INTEGER-REAL . . . . . . . . . . . . . . I.5.1
LONGINT-DECIMAL . . . . . . . . . . . I.7.2
LONGINT-FIXED . . . . . . . . . . . . . I.6.2
LONGINT-INTEGER . . . . . . . . . . . I.3.1
LONGINT-REAL . . . . . . . . . . . . . . I.5.2
REAL-DECIMAL . . . . . . . . . . . . . . I.7.3
REAL-FIXED . . . . . . . . . . . . . . . . I.6.3
REAL-INTEGER . . . . . . . . . . . . . . I.3.2
REAL-LONGINT . . . . . . . . . . . . . . I.4.2
Conversion To:
DECIMAL . . . . . . . . . . . . . . . . . . . . I.7
FIXED . . . . . . . . . . . . . . . . . . . . . . I.6
INTEGER . . . . . . . . . . . . . . . . . . . . I.3
LONGINT . . . . . . . . . . . . . . . . . . . . I.4
REAL . . . . . . . . . . . . . . . . . . . . . . I.5
Conversions . . . . . . . . . . . . . . . . . . . . I.1
Arithmetic Operation . . . . . . . . . . . . I.2.2
Assignment Operation . . . . . . . . . . . I.2.1
Comparison . . . . . . . . . . . . . . . . . . I.2.2
Implicit . . . . . . . . . . . . . . . . . . . . . I.2
I/O Operation . . . . . . . . . . . . . . . . I.2.3
Mathematical Function . . . . . . . . . . I.2.4
DECIMAL . . . . . . . . . . . . . . . . 5.12, 6.7.3
Declaration . . . . . . . . . . . . . . . . . . . 8.2.3
FILE . . . . . . . . . . . . . . . . . . . . 6.5, 6.7.3
FIXED . . . . . . . . . . . . . . . . . . 5.11, 6.7.3
Function, Array . . . . . . . . . . . . . . . . . 6.1
Functions:
DECIMAL . . . . . . . . . . . . . . . . . . . 5.12
Enumeration . . . . . . . . . . . . . . . . 2.2.1.1
FIXED . . . . . . . . . . . . . . . . . . . . . 5.11
INTEGER . . . . . . . . . . . . . . . . . . . 5.2
LONGINT . . . . . . . . . . . . . . . . . . . 5.2
REAL . . . . . . . . . . . . . . . . . . . . . . 5.10
INTEGER . . . . . . . . . . . . . . . . . 5.2, 6.7.3
LONGINT . . . . . . . . . . . . . . . . 5.2, 6.7.3
Operators:
BOOLEAN . . . . . . . . . . . . . . . . . . . 5.3
DECIMAL . . . . . . . . . . . . . . . . . . . 5.12
FIXED . . . . . . . . . . . . . . . . . . . . . 5.11
INTEGER . . . . . . . . . . . . . . . . . . . 5.2
LONGINT . . . . . . . . . . . . . . . . . . . 5.2
Pointer . . . . . . . . . . . . . . . . . . . . . 6.6
REAL . . . . . . . . . . . . . . . . . . . . . 5.10
SET . . . . . . . . . . . . . . . . . . . . . . . 6.4
Packed Data . . . . . . . . . . . . . . . . . . . 6.7
Pointer . . . . . . . . . . . . . . . . . . 6.6, 6.7.3
Procedures, Array . . . . . . . . . . . . . . 6.1.1
REAL . . . . . . . . . . . . . . . . . . 5.10, 6.7.3
Record . . . . . . . . . . . . . . . 6.2, 6.2.1, 6.7.3
Scalar . . . . . . . . . . . . . . . . . . 5.7.2, 6.7.3

SET . . . . . . . . . . 6.4, 6.7.3
String . . . . . . . . . . 6.7.1
Subrange . . . . . . . . . . 5.7.3, 6.7.3
Transfer . . . . . . . . . . 6.9

Types:
Data . . . . . . . . . . 2.2
Enumeration . . . . . . . . . . 2.2.1.1, 5.1
Internal Representation Of . . . . . . . . . . 6.7.3
Simple . . . . . . . . . . 2.2.1
Structured . . . . . . . . . . 2.1.2

UB Function . . . . . . . . . . 6.1, 6.4, 6.7.1
Unformatted Abnormal Termination:
Dump . . . . . . . . . . F14-4
Analysis . . . . . . . . . . 14.7.6
UNPACK Procedure . . . . . . . . . . 6.1.1
UNSAFEOPT Option . . . . . . . . . . 9.6.5
USE Command . . . . . . . . . . 12.3.7.5
USE OBJECT Command . . . . . . . . . . 12.3.7.4
USE PROCESS Command . . . . . . . . . . 12.3.3.5
User-Defined Functions . . . . . . . . . . 8.8
User Termination Routines . . . . . . . . . . F.7
Utility:
Execution, Split Object . . . . . . . . . . 12.5
SPLIT . . . . . . . . . . 12.5
Split Object . . . . . . . . . . 12.5
Split Program . . . . . . . . . . 12.4
SPLITPGM . . . . . . . . . . 12.4

Value Parameters . . . . . . . . . . 8.5
Values, CEIL . . . . . . . . . . TI-1
VAR Declaration . . . . . . . . . . 8.2.4
Variable:
Access . . . . . . . . . . F.5.1, F8-4
Dynamic . . . . . . . . . . 6.6.1
Entire . . . . . . . . . . 6.1.2
Variants . . . . . . . . . . 6.2.2

WARNINGS Option . . . . . . . . . . 9.3.5
WHILE Statement . . . . . . . . . . 5.6.2
WIDELIST Option . . . . . . . . . . 9.3.2, F9-1
WITH Statement . . . . . . . . . . 6.3, 8.2
Without Parameters, Routines . . . . . . . . . . 8.5.4
Words, Reserved . . . . . . . . . . 3.2
Workspace, Routine . . . . . . . . . . F.4.1
WRITE Procedure . . . . . 6.5, 6.5.1, 6.5.2, 6.5.3, 6.5.5, 6.5.6
WRITEEOF Procedure . . . . . . . . . . 6.5.1
WRITELN Procedure . . . . . . 6.5, 6.5.2, 6.5.6

XCODE SCI Command . . . . . . . . . . 11.5
XCONFIG SCI Command . . . . . . . . . . 12.3.11
XCONFIGI SCI Command . . . . . . . . . . 12.3.11
XHT Command . . . . . . . . . . 14.3.4
XNESTER SCI Command . . . . . . . . . . 10.3.5
XPT SCI Command . . . . . . . . . . 12.4.4, 14.3.2
XRASS SCI Command . . . . . . . . . . 13.3
XSILT SCI Command . . . . . . . . . . 11.5
XSPLIT SCI Command . . . . . . . . . . 12.5
XT Command . . . . . . . . . . 14.3.4
XTIP SCI Command . . . . . . . . . . 11.5
XTS Command . . . . . . . . . . 14.3.4

72COL Option . . . . . . . . . . 9.6.1
$LDCR Procedure . . . . . . . . . . 15.2.1
$SBO Procedure . . . . . . . . . . 15.2.2
$SBZ Procedure . . . . . . . . . . 15.2.3
$STCR Procedure . . . . . . . . . . 15.2.4
$TB Function . . . . . . . . . . 15.2.5
$$CC . . . . . . . . . . 11.5, 12.3.11, 14.4.2

CUT ALONG LINE

# USER'S RESPONSE SHEET

**Manual Title:** Model 990 Computer TI Pascal User's Manual (946290-9701)

**Manual Date:** 15 January 1979 **Date of This Letter:** ______

**User's Name:** ______ **Telephone:** ______

**Company:** ______ **Office/Department:** ______

**Street Address:** ______

**City/State/Zip Code:** ______

Please list any discrepancy found in this manual by page, paragraph, figure, or table number in the following space. If there are any other suggestions that you wish to make, feel free to include them. Thank you.

| Location in Manual | Comment/Suggestion |
|---|---|
| | |
| | |
| | |

NO POSTAGE NECESSARY IF MAILED IN U.S.A.
FOLD ON TWO LINES (LOCATED ON REVERSE SIDE), TAPE AND MAIL

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS PERMIT NO. 7284 DALLAS, TX

POSTAGE WILL BE PAID BY ADDRESSEE

TEXAS INSTRUMENTS INCORPORATED
DIGITAL SYSTEMS GROUP

ATTN: TECHNICAL PUBLICATIONS
P.O. Box 2909 M/S 2146
Austin, Texas 78769

FOLD

TEXAS INSTRUMENTS
INCORPORATED
DIGITAL SYSTEMS DIVISION
POST OFFICE BOX 2909 AUSTIN, TEXAS 78769