UNISYS

OS 1100

Conversational Time Sharing (CTS)

Programming Guide

Previous Title: Time Sharing Guide For CTS Level 8R1 Users

Relative to Release Level 8R1

Priced Item

January 1988

Printed in U S America
UP-8118.2-A

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product and related material disclosed herein are only furnished pursuant and subject to the terms and conditions of a duly executed Program Product License or Agreement to purchase or lease equipment. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such License or Agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, indirect, special or consequential damages.

You should be very careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

Correspondence regarding this publication should be forwarded using the remark form in this manual, or remarks may be addressed directly to Unisys Corporation, Large Systems Product Information, P.O. Box 64942 MS: WE1A, St. Paul, Minnesota, 55164-0942, U.S.A.

Page Status Summary
Issue: UP-8118.2-A

Section	Pages	Update
Front Cover		A
Title Page/Disclaimer		A
User Comment Form		A
PSS	1	A
Contents	1 - 8	Orig.
Section 1	1 - 6	Orig.
Section 2	1 - 42	Orig.
Section 3	1 - 22	Orig.
Section 4	1 - 16	Orig.
Section 5	1 - 27	Orig.
Section 6	1 - 20	Orig.
Section 7	1 - 16	Orig.
Section 8	1 - 40	Orig.
Section 9	1 - 7	Orig.
Section 10	1 - 5	Orig.
Section 11	1 - 44	Orig.
Section 12	1 - 14	Orig.
Section 13	1 - 4	Orig.
Appendix A	1 - 3	Orig.
Appendix B	1 - 13	Orig.
Index	1 - 10	A
Back Cover		A

Section	Pages	Update

1. Introduction

1.1. Scope of Manual

This guide teaches how to create and execute programs from remote terminals using the SPERRY UNIVAC Series 1100 Conversational Time Sharing (CTS) System. If more detailed explanation of a given CTS command is necessary, consult SPERRY UNIVAC Series 1100 Conversational Time Sharing (CTS) System, Programmer Reference, UP-7940 (current version).

Sections 2 through 6 of this guide show sequentially how to create, save, display, edit, and execute a program using CTS. These sections provide enough information to build most programs. Sections 7 through 14 discuss more complex CTS functions. Try to understand and use the concepts of the first half of the guide before going on to these sections.

1.2. Style Conventions

Anything displayed on a terminal screen by the system will appear in this guide in boldface type. Input typed at the terminal will be in *italic* type.

For most examples, verbatim copies from terminal screens have been used. Because many terminals print only uppercase letters, the conventional symbolism for some CTS names, such as the working area, f, and the program name, d, cannot be used.

Optional command parameters are enclosed by brackets ([ ]).

1.3. Calling CTS

The Conversational Time Sharing (CTS) System is a processor operating on the Series 1100 System. This processor is activated by typing the @CTS control statement that has the format:

@CTS. *options file-name.*

where:

options may be:

F This option specifies that a unique identifier is to be used instead of run-id wherever the run-id would be used as part of a file name. This option allows the establishment of a unique CTS$FILE and default assumed save file F in an environment where the actual run-id may not be known.

I This option forces initialization. All of the steps described in 1.3.1.2 are taken even if the recovery file CTS$FILE exists and is usable.

N This option causes a faster initialization by skipping steps 4, 5, and 6 as described in 1.3.1.2.

P This option specifies that the initial mode of the working area will be Fieldata. If neither the P nor the Q option is specified, the Q option is assumed.

Q This option specifies that the initial mode of the working area will be ASCII. If neither the P nor the Q option is specified, the Q option is assumed.

L This option disallows the XCTS command. The user is locked into CTS.

file-name. is a file name in the operand of the control card recognized by CTS as being the assumed file F. If no file name is specified, a file with the name of the user's run-id, or identifier if the F option is specified, will be created. In either case, this file name is recognized only when CTS initializes. If CTS does not initialize, the name of the save file is restored (like the other operating conditions) from the recovery file.

Example:

```
@CTS,IN  MYFILE.
CTS  8R1   12 FEB 81 AT 08:47:24
THE ASSUMED MODE IS ASCII
```

The assumed save file (F) is MYFILE. If there is a system news file or if the user has a USER$ subroutine, it is ignored because of the N option.

1.3.1. Initialization of CTS

CTS initialization can be done in two ways, depending on the F option on the processor call. If the F option is used, the initialization procedure described in 1.3.1.1 is performed. If it is not used, the initialization procedure described in 1.3.1.2 is performed.

1.3.1.1. Initialization With the F Option

CTS initialization begins with the following two steps:

1. If the I option is also specified on the @CTS command, a unique identifier is requested from the user. This identifier may be one to six characters in length, and must consist of characters allowed in a valid run-id. Using this identifier, the file name for the save file is created, as well as the file name for the auto recovery file CTS$FILE.

Contents

Page Status Summary

Contents

1. Introduction 1-1

1.1. Scope of Manual 1-1

1.2. Style Conventions 1-1

1.3. Calling CTS 1-1
1.3.1. Initialization of CTS 1-2
1.3.1.1. Initialization With the F Option 1-2
1.3.1.2. Initialization with the I Option 1-3
1.3.1.3. Reentering CTS 1-4
1.3.1.3.1. Normal 1-4
1.3.1.3.2. After a System Crash 1-4
1.3.2. System News File 1-4
1.3.3. Initialization Subroutine – USER$ 1-5

1.4. Exiting from CTS – XCTS 1-5

1.5. Changing Control Characters = 1-6

1.6. Interrupting the System – @@X CIO 1-6

2. Creating a New Program 2-1

2.1. General 2-1

2.2. The Contents of the Working Area – f 2-3
2.2.1. Specifying Part of a Line – ASSUME COLUMN 2-4
2.2.2. Specifying Part of an Edited Line – ASSUME ECOLUMN 2-4
2.2.3. Specifying Part of the Working Area for Siting – ASSUME OCOLUMN 2-5
2.2.4. Specifying Part of the Line for Printing – ASSUME PCOLUMN 2-5
2.2.5. Specifying Part of a Line to Search – ASSUME SCOLUMN 2-5
2.2.6. Line Numbers 2-6
2.2.7. Line Pointer – p 2-6
2.2.8. Line Number Specifications 2-7
2.2.8.1. Specifying a Sequence of Line Numbers 2-7

2.2.8.2. Specifying a Range of Line Numbers – L 2-7
2.2.9. Setting the Character Mode – ASSUME ASCII 2-9
2.2.10. Protecting f from a Loss Due to System Stop – ASSUME AUTO 2-9

2.3. Creating Lines of Data in f 2-10
2.3.1. Controlling the Solicitation Sequence – ASSUME POLL 2-11
2.3.2. Automatic Line Number Generation – NUMBER 2-12
2.3.3. Termination of Automatic Line Numbering – MANUAL 2-15
2.3.4. Defining Tab Stops and Character – TAB 2-16
2.3.5. Restricting Certain CTS Commands – ASSUME EDIT 2-16

2.4. Prescan Modules 2-17
2.4.1. BASIC 2-17
2.4.1.1. Scanning for American National Standard BASIC—ANSI 2-19
2.4.2. FORTRAN 2-19
2.4.2.1. Automatic Formatting 2-21
2.4.2.2. Continuation Lines 2-22
2.4.2.3. Abbreviated Key Words 2-23
2.4.2.4. Automatic Global Syntax Analysis (BFOR Only) 2-24
2.4.2.5. Global Syntax Analysis – CHECK 2-26
2.4.2.6. Controlling Automatic Global Syntax Analysis (BFOR Only) 2-26
2.4.3. COBOL 2-27
2.4.3.1. Operational Description 2-27
2.4.3.2. Modes of Operation 2-31
2.4.3.2.1. Edit Mode 2-31
2.4.3.2.2. Conversational Mode 2-32
2.4.3.2.3. Program File Mode 2-37
2.4.3.3. Summary 2-37
2.4.4. APL 1100/CTS 2-37
2.4.4.1. Access to APL 1100 2-38
2.4.4.2. Processor Options 2-38
2.4.4.3. Statements 2-39
2.4.5. Other Processors 2-40
2.4.6. Controlling Prescan Local Syntax Checking – SYNTAX 2-41
2.4.7. Terminating Prescan Control – CLEAR 2-42

3. Saving and Retrieving Programs 3-1

3.1. Specifying a Program Element or Data File 3-1

3.2. Making a Permanent Copy – SAVE 3-1
3.2.1. Saving f as a Program 3-2
3.2.2. Saving f as a Data File 3-7
3.2.3. Setting the Maximum Length of a Saved Line – ASSUME SAVELENGTH 3-9

3.3. Updating a Copy – REPLACE 3-9

3.4. Discarding a Copy – UNSAVE 3-11

3.5. Retrieving a Copy – OLD 3-13

3.6. Combining a Copy with f – MERGE 3-17
3.6.1. Resolution of Line Number Conflicts 3-18
3.6.2. MERGE Examples 3-20

3.7. Selecting Data Mode – DATA 3-21

4. Displaying and Printing Programs 4-1

4.1. Printing and Listing at Terminals 4-1
4.1.1. Displaying of f – PRINT 4-1
4.1.2. Compact Display of f – QUICK 4-5
4.1.3. Spacing Images in CTS Output Listing – SKIP 4-5
4.1.4. LIST 4-6
4.1.4.1. Displaying f – LIST L 4-6
4.1.4.2. Displaying Names of Saved Elements – LIST SAVED 4-6
4.1.4.3. Displaying the Names of Assigned Files – LIST INUSE 4-9

4.2. Sending the Output to Another Device 4-10
4.2.1. Sending Output to an Onsite Device – SITE 4-10
4.2.2. Output to Punched Cards – CARDS 4-11
4.2.3. Output to Paper Tape – PUNCH 4-12
4.2.3.1. Paper Tape Input – PTI 4-13
4.2.3.2. Setting the Line Length – ASSUME INPUTWIDTH 4-13

4.3. Setting Defaults for Printing 4-14
4.3.1. Defining Terminal Line Length – ASSUME PRINTWIDTH 4-14
4.3.2. Compressing Output – ASSUME QUICK 4-14
4.3.3. Defining an Onsite Device – ASSUME SITE 4-14
4.3.4. Specifying a Default Heading – ASSUME HEADING 4-15
4.3.5. Specifying a Default Return-to Message – ASSUME RETURN 4-15
4.3.6. Setting the Number of Copies – ASSUME COPY 4-15
4.3.7. Controlling the Line Number Display – ASSUME TYPE 4-15
4.3.8. Sending Print to an Alternate File – ASSUME BREAKPOINT 4-16

5. Editing and Modifying Programs 5-1

5.1. Locating Information in f to be Modified 5-1
5.1.1. Finding a String – LOCATE 5-1
5.1.2. Finding a String – FIND 5-5
5.1.3. Controlling the Display of Matched Lines – ASSUME BRIEF 5-7
5.1.4. Controlling the Display of Line Numbers of Matched Lines – ASSUME LINES 5-8
5.1.5. Reprinting of Lines Keyed into CTS – ASSUME ECHO 5-8
5.1.6. Setting the FILLER Default for LOCATE – ASSUME FILLER 5-9
5.1.7. Setting the SPACER Default for LOCATE – ASSUME SPACER 5-9
5.1.8. Setting the STRING Default – ASSUME STRING 5-9
5.1.9. Controlling the Printing of the NUMBER OF OCCURRENCES Message – ASSUME OCCURRENCES 5-10 5-10

5.2. Modifying Lines of f 5-11
5.2.1. Discarding Part of f – DELETE 5-11
5.2.2. Replacing Strings – CHANGE 5-14
5.2.3. Editing a Line – INLINE 5-18
5.2.4. Inserting Strings – INSERT 5-19

5.3. Manipulation in f 5-22
5.3.1. Erasing and Naming f – NEW 5-22
5.3.2. Reorganizing Line Numbers – RESEQUENCE 5-23
5.3.3. Nondestructive Line Copy – DITTO 5-24

5.3.4. Destructive Line Copy – MOVE 5-25
5.3.5. Changing the Name of f – RENAME 5-26
5.3.6. Resolving Line Number Conflicts – ASSUME RESEQUENCE 5-26

6. Execution and Creation of Object Programs 6-1

6.1. General 6-1
6.1.1. Methods Used 6-1
6.1.2. Operating System Aspects of Compilation, Collection, and Execution 6-2
6.1.3. Compilation, Collection, and Execution Under CTS 6-3

6.2. Compiling, Collecting, and Executing in one Operation – RUN 6-4
6.2.1. Setting the Assumed Compiler – ASSUME COMPILER 6-9
6.2.2. Changing the Save and Object File – ASSUME FILE 6-10
6.2.3. Changing the Object File – ASSUME OBJECT 6-11
6.2.4. Changing the Save File – ASSUME PROGRAM 6-11
6.2.5. Changing the Name of the Relocatable Element – ASSUME RELOCATABLE 6-13

6.3. Executing, Naming, and Saving Absolute Elements 6-13
6.3.1. Executing an Absolute Element — XQT 6-14
6.3.2. Naming the Absolute Element — ASSUME XQT 6-15

6.4. Creating Relocatable and Absolute Elements 6-16
6.4.1. Creating Relocatable Elements — COMPILE 6-16
6.4.2. Creating an Absolute Element — MAP 6-17
6.4.2.1. Specifying the Main Program — ASSUME MAIN 6-18
6.4.2.2. Specifying Additional Libraries — ASSUME LIBRARIES 6-18
6.4.2.3. Specifying MAP Directives — ASSUME MAP 6-19

6.5. Initiating a Processor Call — PXQT 6-19

7. File Handling 7-1

7.1. Mass Storage Files 7-1
7.1.1. Mass Storage Files in the Series 1100 Operating System 7-1
7.1.2. Use of Mass Storage Files by CTS 7-2

7.2. Permanent and Temporary Files 7-3

7.3. Drum, Disk, and Tape Files 7-3

7.4. Security 7-7

7.5. Manipulating File Contents 7-7
7.5.1. CREATE 7-7
7.5.2. PURGE 7-8
7.5.3. RELEASE 7-9
7.5.4. COPY 7-10
7.5.5. USE 7-11
7.5.6. PACK 7-12
7.5.7. ADD 7-13
7.5.8. ERASE 7-14

7.6. Submitting Operating System Control Statements – CSF 7-14

7.7. Examples of File Usage 7-15
7.7.1. FORTRAN 7-15
7.7.2. BASIC 7-16
7.7.3. Alternate Program File 7-16

8. Subroutines 8-1

8.1. General 8-1

8.2. Building a Subroutine 8-2
8.2.1. SAVE 8-3
8.2.2. SUBROUTINE 8-3
8.2.3. PROC 8-6

8.3. Programming a Subroutine 8-7
8.3.1. Variables 8-7
8.3.2. SET 8-7
8.3.3. QUERY 8-8
8.3.4. TYPE 8-9
8.3.5. JUMP 8-9
8.3.5.1. ERROR 8-11
8.3.5.2. FOUND 8-11
8.3.5.3. BRANCH 8-12
8.3.6. Variable Substitution in CTS Commands 8-13
8.3.7. Miscellaneous Commands 8-16
8.3.7.1. ENTRY 8-16
8.3.7.2. RETURN 8-17
8.3.7.3. END 8-18
8.3.7.4. GENERATE 8-18
8.3.7.5. Setting the Line Pointer – GO 8-20
8.3.7.6. Commentary Information 8-21
8.3.7.6.1. REMARK 8-21
8.3.7.6.2. Percent-Sign (%) 8-22
8.3.7.7. Leaving CTS Mode – EXIT 8-22
8.3.8. Removing a Variable or Subroutine — DROP 8-23

8.4. Calling a Subroutine 8-24
8.4.1. CALL 8-24
8.4.1.1. ASSUME CALL FILE 8-25
8.4.2. CALL Parameter 8-25
8.4.3. Subroutine Debugging 8-27
8.4.3.1. ASSUME SBUG 8-27
8.4.3.2. Subroutine Trace 8-28
8.4.3.3. ASSUME TRACE 8-29
8.4.3.4. ASSUME JUMP 8-30
8.4.3.5. Miscellaneous Conditions 8-31
8.4.3.6. Displaying Variables 8-33
8.4.3.7. Subroutine Nesting 8-33

8.5. Saving Subroutines Between CTS Sessions 8-34
8.5.1. Saving a Subroutine as an Omnibus Element — SSUB 8-34
8.5.2. Replacing a Saved CTS Subroutine Element — RSUB 8-35

8.6. Examples 8-36
8.6.1. Selective Execution 8-36
8.6.2. Programmable Editor 8-37
8.6.3. Starting Batch Runs 8-39

9. Operating Information and Assistance 9-1

9.1. File Information 9-1
9.1.1. LIST CATALOG 9-1
9.1.2. LIST FILE 9-2
9.1.3. CTS Internal File Names 9-2

9.2. Miscellaneous Operating Information 9-3
9.2.1. NEWS File 9-3
9.2.2. Number of Lines in f – LENGTH 9-3
9.2.3. DATE 9-4
9.2.4. Central Processor Time – CPTIME 9-4
9.2.5. STATUS 9-4

9.3. Online Assistance 9-5
9.3.1. Command Information – HELP 9-5
9.3.2. Error Message Information – EXPLAIN 9-6

10. User Communications 10-1

10.1. General 10-1

10.2. User/Operator Communications 10-1
10.2.1. Operating System Message – @MSG 10-1
10.2.2. CTS Message – OPR 10-3

10.3. User/User Communications 10-4
10.3.1. MAIL 10-4
10.3.2. LOOK 10-5

11. Debugging Techniques 11-1

11.1. Program Debugging 11-1
11.1.1. Examining Processor Output—SCAN 11-2
11.1.2. Terminating SCAN Mode—EDIT 11-3

11.2. Debugging Source Code 11-4
11.2.1. Debug Mode – ASSUME DEBUG 11-4
11.2.2. BASIC 11-4
11.2.2.1. PAUSE 11-4
11.2.2.2. BREAK 11-7
11.2.2.3. TRACE 11-9
11.2.3. FTN 11-12
11.2.3.1. Debug Facility 11-12
11.2.3.1.1. DEBUG 11-13
11.2.3.1.2. AT 11-15
11.2.3.1.3. TRACE ON 11-16
11.2.3.1.4. TRACE OFF 11-16
11.2.3.1.5. DISPLAY 11-17
11.2.3.1.6. Debug Facility Example 11-17

11.2.3.2. Eliminating Program Collection – ASSUME CHECKOUT 11-18
11.2.3.3. Interactive Debugging Mode in the Checkout Compiler 11-18
11.2.3.3.1. Entering Interactive Debug Mode 11-19
11.2.3.3.2. Soliciting Input 11-20
11.2.3.4. Debug Commands 11-20
11.2.3.4.1. BREAK 11-21
11.2.3.4.2. CALL 11-22
11.2.3.4.3. CLEAR 11-23
11.2.3.4.4. DUMP 11-24
11.2.3.4.5. EXIT 11-25
11.2.3.4.6. GO 11-25
11.2.3.4.7. HELP 11-26
11.2.3.4.8. LINE 11-26
11.2.3.4.9. LIST 11-26
11.2.3.4.10. PROG 11-26
11.2.3.4.11. RESTORE 11-27
11.2.3.4.12. SAVE 11-28
11.2.3.4.13. SET 11-29
11.2.3.4.14. SETBP 11-30
11.2.3.4.15. SNAP 11-31
11.2.3.4.16. STEP 11-31
11.2.3.4.17. TRACE 11-31
11.2.3.4.18. WALKBACK 11-32
11.2.3.4.19. Interactive Debugging Example 11-32
11.2.3.5. Contingencies and Restrictions in Checkout Mode 11-34
11.2.3.6. Walkback and the Interactive Postmortem Dump 11-35
11.2.4. RFOR 11-35
11.2.4.1. PAUSE 11-35
11.2.4.2. BREAK 11-38
11.2.4.3. TRACE 11-39

12. Desk Calculator 12-1

12.1. Expressions 12-1
12.1.1. Integer Constants 12-1
12.1.2. Real Constants 12-1
12.1.3. String Constants 12-2
12.1.4. Functions 12-2
12.1.5. CTS Variables 12-9
12.1.6. Operators 12-10

12.2. Variable Definition – SET 12-10

12.3. Evaluating and Printing Expressions – TYPE 12-11

12.4. Iterative Expression Evaluation – DISPLAY 12-12

12.5. Iterative Expression Summation – SUM 12-13

12.6. Removing a Variable or Subroutine – DROP 12-14

13. Batch Mode 13-1

13.1. General 13-1

13.2. Starting a Batch Job from the Terminal 13-2
13.2.1. In Executive Mode – @START 13-2
13.2.2. In CTS Mode – CSF 'START' 13-2
13.2.3. In Either Mode – @@START 13-4

13.3. Batch/Time Sharing Compatibility 13-4

Appendix A. Transparent Control Statements A-1

Appendix B. Explanation of CTS Messages B-1

B.1. Miscellaneous Messages B-1

B.2. CTS Diagnostic Messages B-3

Index

User Comment Sheet

Tables

Table 2-1. APL 1100 Options 2-38
Table 3-1. Assumed Compiler and Options for OLD Command 3-14
Table 12-1. Numeric Functions 12-3
Table 12-2. String Functions 12-5

For example, if the user answers:

ENTER A UNIQUE FILE IDENTIFIER > *JONES*

the default save file F will be JONES, regardless of the user run-id.

2. An attempt is made to restore the CTS operating environment from the recovery file CTS$FILE. This file will exist if the previous CTS terminal session was terminated by a system crash, or if CTS has been called at least once during this terminal session. For a system crash recovery, CTS prints the message:

   CTS RESTART

   If a system crash did occur, then an F option without an I option will cause the unique identifier to be requested so recovery can be made. If no crash occurred, and the I option is not given, then no request for the identifier is made, if CTS$FILE already exists.

   The remaining steps are the same as steps 3 through 7 in 1.3.1.2.

1.3.1.2. Initialization with the I Option

CTS initialization begins with the following sequence:

1. If a save file F exists, it is assigned. If it does not exist, it is created. If this file is not specified in the control statement (see 1.3.1) then the run-id is used.

2. CTS creates a recovery file, CTS$FILE. This file retains the contents of the working area f intact after the user exits from CTS.

3. A sign-on line consisting of the CTS version, current date, and time is printed.

4. If a system news file exists, a solicitation asks whether or not the news should be printed (see 1.3.2).

5. The following reminder is printed:

   IF YOU NEED ASSISTANCE TYPE *HELP

6. If the user has a USER$ subroutine, it is automatically executed at this time (see 1.3.3).

7. The mode of f will be displayed as follows:

   THE ASSUMED MODE IS ASCII

   or

   THE ASSUMED MODE IS FIELDATA

1.3.1.3. Reentering CTS

1.3.1.3.1. Normal

After an exit, CTS retains the contents of the working area f until a @FIN statement is encountered, so it is possible to recover CTS with the working area the same as it was just prior to the exit from CTS. This is accomplished by entering the @CTS control statement (see 1.3) without any options and without a file name, since these were specified in the initial CTS call and are still in the recovery file.

1.3.1.3.2. After a System Crash

When a system crash occurs, the CTS recovery file may or may not have been retained intact. CTS protects against loss of data on an abnormal CTS run termination, but it will not necessarily protect against system crashes. For example, if a system crash occurs when a disk write is being performed on CTS$FILE, some of the pointers set up in CTS may have been changed before the crash but some may not. Thus, when a restart is attempted, errors may occur because some pointers are incorrect.

1.3.2. System News File

If a news file has been established by the site, CTS will automatically solicit a response from the user after the sign-on line is printed. The following is an example of what is received when the news is requested:

```
-> @CTS,I
CTS   8R1   07 FEB 81 AT 07:53:53
THE NEWS IS DATED 07 FEB 81 AT 07:50:02
WOULD YOU LIKE THE NEWS?> YES
1
2          FOR A BRIEF OVERVIEW OF THE NEW FEATURES IN CTS, TYPE IN:
3               CALL CTS-COMMANDS
4          PLEASE REFER TO THE APPROPRIATE DOCUMENTATION FOR A FULLER
5          DESCRIPTION OF THE NEW COMMANDS.
6
IF YOU NEED ASSISTANCE TYPE  *HELP
FOR NEW FEATURES TYPE  *CALL CTS-COMMANDS
THE ASSUMED MODE IS ASCII
->
```

The time and date of the news file is given so the user can see if the news has changed. A *NO* response to the solicitation:

```
WOULD YOU LIKE THE NEWS?
```

would cause the news printing to be skipped.

1.3.3. Initialization Subroutine – USER$

A subroutine can be built (see Section 8) that will automatically be called during initialization. This subroutine must be an element called USER$. It must be saved in the assumed save file F. The rules for establishing the name of the assumed save file are described in 1.3.1. The USER$ subroutine can be used to set operating conditions different from the usual defaults, to check for mail (see 10.3.2, LOOK command), etc. Please note that the USER$ subroutine is not executed if the N option is used during initialization.

For example, a possible USER$ element would be:

```
110 ASSUME AUTO 20 P
120 ASSUME FILE JUDY*JUDY
130 LOOK
140 TAB ; 10,20,30,40
150 ASSUME SITE PR1
```

1.4. Exiting from CTS – XCTS

The XCTS command causes an exit from the CTS processor. This puts the system in "control mode," allowing the user to execute any valid Series 1100 Executive control statement. An exit from CTS via an XCTS command does not alter the working area, f. The working area is retained until the terminal session is terminated by a @FIN control statement. A reentry to CTS via the @CTS control statement will recover the area, as described in 1.3.1.

NOTE:

The L option on the @CTS control statement disallows the XCTS command.

If the working area has been edited using any of the following commands, but not saved or replaced prior to exiting CTS, a warning message is printed. The commands are:

CHANGE	DELETE	DITTO	GENERATE
INLINE	INSERT	MERGE	MOVE
NEW	NUMBER	RENAME	RESEQUENCE
Entering a data line			

For example:

```
-> 10 ABC
-> 20 DEF
-> XCTS
WARNING - THE WORK AREA WAS NOT SAVED/REPLACED
IN EXEC MODE
-> @CTS
-> SAVE ELT
-> XCTS
IN EXEC MODE
```

1.5. Changing Control Characters =

Syntax: = A, B, C

Abbreviation: None

Function: To change the CTS control characters.

The CTS control characters normally are the following symbols:

- * The asterisk is the command control character which is used to indicate a command when in number mode (see 2.3.2). When so used, the "*" will terminate number mode.
- % The percent sign is the delimiter for comments and for substitution of variables (see 8.3.6).
- ' The single quote is the string delimiter (see 5.2.2).

The character specified by A becomes the new CTS command control character rather than the "*". The character specified by B becomes the new variable delimiter rather than the "%" character. The character specified by C becomes the new string delimiter rather than the single quote.

All characters specified for A, B, and C must be special, i.e., not alphanumeric.

Examples:

- =-,; changes the CTS command control character to "-" and the string delimiter to semicolon, while the variable delimiter is not changed from its former definition.
- -=* restores the CTS command control character to the asterisk. Notice that the current command control character precedes the command. This is, of course, optional.

Changing the command control character allows entering data lines in number mode which have an asterisk as the first character. Normally the asterisk would terminate number mode. The command control character cannot be set to "@" since this conflicts with Executive command syntax.

NOTE:

Certain CTS commands depend on the existence of prewritten CTS subroutines. These subroutines, in turn, assume that the variable delimiter and string delimiter are "%" and " ' ", respectively. If these delimiter characters are changed, the commands in question may no longer work, nor, in fact, may other subroutines constructed for personal use. Hence, the = command should be employed with caution. The same variable delimiter must be used throughout a subroutine and it must be the same variable delimiter as when the subroutine definition was made (see 8.3.6).

1.6. Interrupting the System – @@X CIO

Some program errors require cancelling a current program activity. For example, an infinite loop in a program could tie up a terminal and processor. The @@X CIO statement can be used to accomplish this.

NOTE:

Never use the T option of the @@X statement to terminate CTS commands.

2. Creating a New Program

2.1. General

Programs are built by creating a set of data images which conform to the rules of syntax and semantics of a particular programming language. In CTS, data sets of all kinds, including those that happen to be programs, are created in the working area file which is called f. CTS creates f using a predetermined size and assigns it a use-name of CTS$FILE. It is possible to create and assign f to your run before calling @CTS,I. This method will override the default size used by CTS and is useful when excessively large elements or files are to be entered.

Once a program is created in f, it may be saved as an element of a program file, usually F, so it may be used again. (Section 3 discusses the saving and retrieving of programs.) The working area file, f, is thus central to the creation, editing, and use of programs under CTS. This file has a format unique to CTS, i.e., it is neither a program file nor a data file (see 7.1.1). It has several attributes which make it useful for creating, editing, and running programs. These are:

- Contents

  The contents of f consist of data images, each of which is called a line, and each of which has a unique line number. The contents may be nonexistent, in which case f is said to be empty.

- Line Pointer

  This pointer is a nonnegative integer stored by CTS which defines the current line number. The line pointer always exists.

- Name

  This is the name of the contents of f, although for brevity it is sometimes called by the name of f. The name may be nonexistent.

- Assumed Compiler

  This is the operating system processor associated with the data in f. It also includes the options to be used if this processor is used to compile the contents of f. The assumed compiler may be nonexistent.

Most of the commands in CTS deal directly with f. Some of them change its contents, line numbers, name, or assumed compiler explicitly. Others change them indirectly.

There are different ways of getting images into f. They may be entered directly, as if they were data. They may be entered under control of one of the prescan modules of CTS which checks the format of each line as it is entered. The information may be retrieved from a file. The material retrieved from the file need not have been created under CTS.

Once the purpose and method of a program are known and a language and compiler are chosen, creating the program with the help of CTS involves several steps:

a. Enter the lines of code conforming to the chosen compiler language.

b. Remove the obvious errors – typographical errors, formatting errors, etc.

c. Compile the program to locate global errors.

d. Remove the errors uncovered in compilation.

e. When steps c and d have been repeated until no more errors are discovered during compilation, save the program.

f. Test the program by executing it with data designed to verify that it is operating correctly.

g. Modify the program to remove any errors discovered, and return to step c.

h. When steps c through g have been repeated until no more errors are uncovered, the program is considered to be error free and ready to be used. CTS is designed specifically to assist in each of the above steps, allowing quick procession through them.

In step a, for example, CTS makes it easy to enter lines of data formatted to the needs of any compiler. The required column positions and tab characters may be defined to line up margins for convenience (in ALGOL, for example, to line up nested blocks) or of necessity (to comply with FORTRAN or COBOL line formats). In addition, the NUMBER command (see 2.3.2) can request that CTS explicitly enter the line numbers. CTS will then supply them automatically and, if desired, leave room between successive lines for insertion of correction lines or lines which were inadvertently skipped.

Abbreviations or special symbols may be used for long names or expressions which occur frequently, to make keying in the program easier. To do this, use the symbol or abbreviation when typing the program. No special definition is necessary. Then, when the entire program is entered, a single CHANGE command (see 5.2.2) will change all occurrences of one special symbol or abbreviation into the full name or expression throughout the program.

If the chosen compiler has a prescan module in CTS (e.g., BASIC, ASCII FORTRAN, COBOL), the task of entering lines initially is even easier. The formatting may be done by the prescan module. The prescan module may also provide a convenient short form for special words of the language (in FORTRAN; for example, D:N is expanded into DIMENSION). The prescan modules provide for automatic or simplified implementation of a continuation line when a statement is too long for one line. Also, prescan modules do a noncontext syntax scan of each line as it is entered. If a discrepancy is detected, a message tells what is wrong, and the module waits for the statement to be corrected before accepting it. In this way, step b is performed as the program is built.

CTS provides a comprehensive set of commands to help locate and remove errors in the program. Simply listing the program will make many formatting errors obvious. Others may be detected more readily by listing only a part of each line – those columns beyond column 72, for example. If a prescan module is not being used, step b may be implemented by finding the obvious errors and removing them with the editing commands designed to modify lines easily (see 5.1). If a prescan module is being used, this step will have been performed during the keying in of the program. The corrections would have involved using the same editing commands. The editing commands will also be used

even more extensively in steps d and g, not only to make corrections, but also to help locate errors.

CTS makes step c (compiling) quicker and easier, not only by providing a simpler command form (it is usually sufficient to enter COM), but by arranging for the somewhat voluminous compiler output listing to be directed not to the terminal, as would be the normal case, but to a file. If requested,

CTS does a diagnostic scan of this output file after the compilation to display all the diagnostic messages. Alternatively, the SCAN command (see 11.1.1) can be used to retrieve parts of the file and editing commands can be used to search for key words such as ERROR. All of it can be listed or parts can be accessed randomly while locating errors. When this file is no longer needed, control can be returned to the program to make corrections. This technique is also useful in step f. Prescan modules have the advantage that they are oriented towards finding errors by doing only the syntax-scan part of a compilation. This is especially efficient in step c, where the purpose of the compilation is to find errors. Prescan modules produce somewhat more comprehensive and explicit diagnostic messages than their batch compiler counterparts, which makes them particularly useful in step c.

If the program is complex, it may take several sessions at the terminal before a checked-out program exists. CTS makes it convenient to save the contents of f at the end of one session and restore it at the start of the next, so work may be continued. In any case, it is a good idea to save the program occasionally as a backup in case f is accidentally destroyed. The saving and retrieving of programs under CTS are discussed in Section 3.

In summary, CTS facilitates every step of program creation by providing facilities such as:

- aids for entering data into f;
- editing features and commands;
- simplified compilation system;
- prescan modules; and
- saving and restoring of programs.

2.2. The Contents of the Working Area – f

The unit of information in f is the line. In turn, the unit of information in the line is the character. Character positions in a line are numbered from the left, beginning with 1. The line length is usually limited by the input device, but a longer line may be created by editing an existing line in such a way that more characters are inserted than deleted. The longest line that can be created in this way has 132 character positions. Be careful, when using such features for creating programs, to avoid accidently extending a line beyond the limit accepted by the processor (compiler) to be used. For example, some processors have a limit of 80 character positions; others, 72.

The working area file is not lost due to a system stop. When CTS is called after a system stop, the working area contains the lines that existed when the last automatic save was done. An automatic working area save is done after each OLD command (see 3.5), before any program execution command (see Section 6), and periodically if requested by an ASSUME AUTO command (see 2.2.6).

2.2.1. Specifying Part of a Line – ASSUME COLUMN

Syntax: ASSUME COLUMN [(c1,c2)]

Abbreviation: A COL

Function: To establish a new global default column parameter, or to reestablish the standard one.

Occasionally it may be useful to refer to a part of a line (e.g., print or change part of a line). This is done by specifying a range of column numbers. Several of the CTS commands afford this option, which is specified by a parameter of the form: (c1,c2) where c1 is the leftmost column to be considered, and c2 the rightmost. If either is left blank, it is normally assumed to be the end of the line in that direction. Thus, (,20) means the part of a line from the beginning, column 1, through column 20, inclusive. If only one number is specified in the parentheses, it is assumed to be c1, and c2 is assumed to be the rightmost character. If this parameter is omitted, the normal default is (1,132). However, the ASSUME COLUMN command is available to change this default assumption. It also changes the default values for c1 or c2 individually.

The ASSUME COLUMN command specifies default column limits for the ASSUME ECOLUMN, OCOLUMN, PCOLUMN, and SCOLUMN commands. This produces a global column limit for all commands using column limits.

Parameters c1 or c2 may be strings. In this case a search determines the column limits for each line. If the column limits are not found on a particular image, no operation is performed on that line. For example, ('3', '0') would specify columns 6 through 17 for the line:

```
PI = 3.1415926660
```

because the character "3" is positioned in column 6, and the character "0" is positioned in column 17.

2.2.2. Specifying Part of an Edited Line – ASSUME ECOLUMN

Syntax: ASSUME ECOLUMN [(c1, c2)]

Abbreviation: A ECOL

Function: To establish a new default column parameter (or to reestablish the standard one).

The ASSUME ECOLUMN command specifies default column limits for the following CTS commands: INSERT, DELETE. If (c1,c2) is not specified, then the default value of (1,132) is assumed.

Example:

```
->D  A
-> 10 ABCDEFGHABC
->ASSUME ECOL  (1,6)
->INSERT X RJ#
10 #####XGHABC
```

2.2.3. Specifying Part of the Working Area for Siting – ASSUME OCOLUMN

Syntax: ASSUME OCOLUMN [(c1,c2)]

Abbreviation: A OCOL

Function: To establish a new default column parameter, or to reestablish the standard one.

The ASSUME OCOLUMN command specifies default column limits for the CTS commands SITE and CARDS. If (c1,c2) is not specified, then the default value of (1,132) is assumed.

2.2.4. Specifying Part of the Line for Printing – ASSUME PCOLUMN

Syntax: ASSUME PCOLUMN [(c1,c2)]

Abbreviation: A PCOL

Function: To establish a new default column parameter, or to reestablish the standard one.

The ASSUME PCOLUMN command specifies default column limits for the following CTS commands: LIST, PRINT, PUNCH, QUICK. If (c1,c2) is not specified, then the default value of (1,132) is assumed.

Example:

```
-> 10 ABCDEF
->ASSUME PCOL (1,4)
->P 10
10 ABCD
```

2.2.5. Specifying Part of a Line to Search – ASSUME SCOLUMN

Syntax: ASSUME SCOLUMN [(c1,c2)]

Abbreviation: A SCOL

Function: To establish a new default column parameter, or to reestablish the standard one.

The ASSUME SCOLUMN command specifies default column limits for the following CTS commands: FIND, LOCATE, CHANGE. IF (c1,c2) is not specified, then the default of (1,132) is assumed.

Example:

```
-> 10 ABCDEFABC
->ASSUME SCOL (1,3)
->LOC 'DEF' 10
*NOT FOUND
->ASSUME SCOL (1,10)
->LOC 'DEF' 10
10 ABCDEFABC
```

2.2.6. Line Numbers

Each line has a line number which is always a positive integer. This number must be specified when a line is keyed in. CTS can enter the line numbers automatically (see 2.3.2), in which case CTS (optionally) displays the line number before the data is entered. The line number is not actually part of the line, but only a tag by which it is identified.

Lines may be entered into f in any order, but they are sequenced according to ascending line numbers. Therefore, if lines 100 and 110 exist in f, and a line is entered with number 105, it is inserted between 100 and 110. If a line is entered with a line number which already exists in f, the old line is discarded and the new one takes its place.

The smallest allowable line number is 1. The largest is 262,141. When creating a program, space the line numbers to permit easy insertion. Should it be necessary, however, the RESEQUENCE command (see 5.3.2) will change the line numbers of all or a part of f, maintaining the sequence relationship of the changed lines to one another.

2.2.7. Line Pointer – p

CTS maintains a line pointer, p, which is always set to a nonnegative integer. This integer is the line number of the current line. Many commands use p as the default specification when a line number parameter is omitted. To take advantage of this, the value of p must be known. Any CTS command which references a line explicitly or implicitly will change the line pointer.

Commands for saving and restoring programs usually set p to 0. When keying lines of data into f, p is set to the most recent line entered.

When editing commands are used, p is left equal to the line number of the last line edited. (Many lines may be edited with a single command.) However, if a line number specification in an editing command causes the display of the message:

```
TOP OF FILE
```

p is set to 0. If it causes the display of the message:

```
END OF FILE
```

p is set to 0. If it causes the message:

```
<21> LINE n DOES NOT EXIST
```

p is unchanged.

The line pointer, p, may be set to a specific existing line number or moved ahead or back with the GO command (see 8.3.7.5).

To find the value of p, use the statement

```
->TYPE P()
```

This combination of the TYPE command and the function P( ) (see Table 12-1) will cause the value of p to be displayed on the following line.

2.2.8. Line Number Specifications

There are two kinds of line number specifications. The first specifies a sequence of line numbers. The second, more common, specifies a range of numbers. Some editing commands use both types in different parameter positions.

2.2.8.1. Specifying a Sequence of Line Numbers

Some of the editing commands generate a sequence of line numbers which are given to successive new lines they have created, either by moving them from elsewhere in f (MOVE, see 5.3.4, and DITTO, 5.3.3), acquiring them from the element of a program file or from a data file (MERGE, see 3.6), or accepting them from the terminal (NUMBER, see 2.3.2, and GENERATE, 8.3.7.4). The form of this type of parameter is:

i, j

where i is the initial line number and j is the increment to be added to form each successive line number. Thus, if:

100,10

is specified for such a parameter, the successive line numbers generated are:

100,110,120,130,...

The first subparameter, i, must be a legal line number, and the increment, j, must be a positive integer.

2.2.8.2. Specifying a Range of Line Numbers – L

This type of parameter, called L, is used to identify what part of f (or a saved program in the case of MERGE and OLD) is to be included in the operation of a command. There are many forms of L. It may be a single line number, two line numbers separated by a comma (defining the endpoints of the range of line numbers), All (meaning all line numbers in f), and many others. Many of the CTS commands permit L to specify a sequence of line numbers which are decreasing in magnitude. Others do not. When an L parameter specification contains endpoints, the line numbers specifying the endpoints are included in the range.

Depending on the command, the default assumption when L is omitted is either A (all of the lines), *+ (all following lines), or the value of p (the current line). The various forms of L are:

n	meaning line number n.
+0 or ,	meaning the current line.
*	meaning the line following the current line.
+	meaning the current line and all following lines.
*+	meaning all lines following the current line.
*+i	meaning i lines following the current line.
*i	meaning i lines following the current line.

-	meaning the current line and all preceding lines.
*-	meaning all lines preceding the current line.
*-i	meaning i lines preceding the current line.
n1,n2	meaning all lines of f from n1 through n2 inclusive.
*,n2	meaning the line following the current line if n2 is greater than the current line number (or the line preceding the current line if n2 is less than the current line number) through line n2.
n1,*	meaning n1 through the line preceding the current line or through the line following the current line, depending on whether n1 is less or greater than the current line number.
n+i n-i	meaning line number n and the next i lines which follow (for +i) or precede (for -i) line number n.
A or ALL	meaning all the lines. (This automatically turns on the R (repeat) option for those commands that have it. 0+ also means all lines but the R option is not turned on.)
n+ n-	meaning line number n and all following lines (for n+) or all preceding lines (for n-).
+i -i	meaning the current line and i lines following the current line (for +i), or i lines preceding the current line (for -i).
!-	denotes all lines in reverse order.
!-i	denotes the last i lines of f in reverse order.
!,n	denotes lines from the end of f through line n.
!,*	denotes all lines from the end of f through the line following the current line.
!-1	denotes the last line.
0+	denotes all the lines.

If a single line number is specified and this line number does not exist, an error message is printed. However, in the case where a group of lines is specified, the beginning and ending line numbers are bounds. Thus, if a beginning line number is specified which is not in f, the next higher existing line number is implied. If the ending line number is not in f, the next lower existing line number than that specified is implied. If no lines exist within the range an error message is printed.

2.2.9. Setting the Character Mode – ASSUME ASCII

Syntax: ASSUME ASCII [ON/OFF]

Abbreviation: A ASC

Function: To set the character mode of CTS.

CTS can operate in either ASCII character mode or Fieldata character mode. Unless the P option is specified on the processor call card, CTS will be initialized as an ASCII processor (ASSUME ASCII ON). If neither ON nor OFF is specified in the command, ON is assumed as the default.

Modes may be switched without converting the operating environment. Internal characteristics such as tab characters, variables, FILLER, and SPACER are translated by CTS. All Fieldata alphabetics will be converted to uppercase ASCII characters. Special ASCII characters which do not have a Fieldata counterpart will be converted to a Fieldata "?".

NOTE:

Fieldata code 077 (≠) is a special stop code for the Executive and, hence, should not be used in source input.

The OLD command may change the mode if the mode of the element or file is different from the current mode and a prescanner is not active. A call to a prescanner may change the mode and the mode cannot be changed once a prescanner is active. A message will be printed if the mode is altered by either of these methods.

2.2.10. Protecting f from a Loss Due to System Stop – ASSUME AUTO

Syntax: ASSUME AUTO [i [k]]

Abbreviation: A AUT

Function: To control the frequency of automatic checkpoint saves by CTS.

CTS automatically does periodic checkpoint saves to record its status as insurance against the occurrence of an abnormal run termination. Only the most recent checkpoint save is available, since each one overwrites the previous one. When a new run with the same run-id and project-id is logged on after an abnormal termination, CTS attempts a recovery. If successful, it displays the message:

CTS RESTART

When a run terminates normally, the checkpoint file is discarded.

The ASSUME AUTO command controls the frequency of these saves and directs CTS to announce when a checkpoint save is taken. The parameter i, a positive integer greater than 4, specifies the number of input lines which trigger a checkpoint. The lower limitation of 5 is to avoid excessive overhead. If i is 0, an immediate save is done.

If k is not empty, the message:

AUTO

is displayed each time a checkpoint is taken.

Omitting both parameters disables the checkpointing feature, and no automatic saves are subsequently produced.

With or without the ASSUME AUTO feature, a checkpoint save is done after each OLD command (see 3.5), before any program execution command (see Section 6), and before any command which causes an exit.

2.3. Creating Lines of Data in f

A program is usually created under CTS by keying lines of data into the system. CTS puts these lines into f. If they are to be a program, they must conform to the rules of syntax of the compiler to be used to compile the program. Of course, if the program is to perform the intended task, the lines must conform to the semantics of the compiler language as well. Unless these lines are entered under the control of a prescan module, CTS does not concern itself with the contents of the lines. As far as CTS is concerned, they are lines of data. More specifically, each line is a string of characters. For purposes of identification and ordering, each line also has associated with it (but not part of it) a line number, as described earlier in this section.

The most elementary way to enter a line of data is to key in the line number, followed by an optional single space, followed by the string of characters which constitute the contents of the line.

For example, if, after the solicitation character, the following is keyed in:

```
-> 100 Δ THIS Δ IS Δ A Δ LINE.
->
```

The string of characters:

THIS ΔIS ΔA ΔLINE.

has been entered as line 100. Note that for better visibility of blanks, the character Δ is used in the example where blanks would appear on a terminal. If the single blank after the line number was omitted, the contents of the line would be the same. The first character is ignored if it is a blank. This allows separating the data from the line number for readability. If, after the line number, two blanks are used instead of one, the second blank becomes part of the contents of the line. If the above line is followed by the line:

```
-> 110 ΔΔ THIS Δ IS Δ ANOTHER Δ LINE.
->
```

Line 110 contains the string:

ΔTHIS Δ IS ΔANOTHER ΔLINE.

Note the leading blank this time.

Continuing in this fashion, one line may be entered after another, giving each a unique line number, until the entire program has been entered into f. If a line has been left out, simply key in the omitted line specifying a line number which is between the two line numbers of the surrounding lines (assuming room is left for it). No matter in what order the lines are entered, CTS sequences them in order of increasing line numbers. If a line which has been entered has errors, it may be replaced by simply keying in the line again (with identifying line number identical to the line number of the incorrect line). This deletes the old line and replaces it with the new one. The replacement and insertion of lines in a data set constitutes perhaps the most elementary form of editing.

The following example illustrates some of the above points:

```
->50 Δ THIS Δ IS Δ THE Δ FIRST Δ  LINE.
->100 Δ THIS Δ IS Δ THE Δ THE.
->100 Δ THIS Δ IS Δ THE Δ THIRD.
->75 Δ THIS Δ IS Δ THE Δ SECOND Δ  LINE.
->PRINT Δ ALL
50 ΔTHIS ΔIS ΔTHE ΔFIRST ΔLINE.
75 ΔTHIS ΔIS ΔTHE ΔSECOND ΔLINE.
100 ΔTHIS ΔIS ΔTHE ΔTHIRD.
END ΔOF ΔFILE
->
```

Line 100 was entered incorrectly and out of sequence. It was then replaced with the corrected line. Line 75 was then inserted between lines 50 and 100. The PRINT command (4.1.1) was then used to display the contents in their updated form.

2.3.1. Controlling the Solicitation Sequence – ASSUME POLL

Syntax: ASSUME POLL [ON/OFF]

Abbreviation: A POL

Function: To eliminate or reestablish the display of the first character of the 2-character solicitation sequence.

The first character of the solicitation sequence normally tells what mode CTS is in. It is a "–" for element mode, unless a prescan module is in effect, in which case a ">" is used. An "*" is used for data mode.

ASSUME POLL OFF suspends the display of this character. ASSUME POLL ON or ASSUME POLL will reinstate the standard CTS behavior.

2.3.2. Automatic Line Number Generation – NUMBER

Syntax: NUMBER [i] [,j] [k]

Abbreviation: N

Function: To direct CTS to supply a specified sequence of line numbers for lines of data being entered into f.

To relieve the tedium of keying in line numbers when entering a program or data set, the NUMBER command conditions CTS to supply them. The parameters i and j specify the sequence of line numbers, and the parameter k determines whether CTS displays the line numbers or not. Lines entered in this mode are placed in f as data lines until the mode is terminated by entering an asterisk (*) as the first character after the solicitation sequence.

The sequence of line numbers produced by CTS in response to a NUMBER command is:

i, i+j, i+2j, i+3j, ...,

CTS assigns these line numbers to the lines as they are entered. As lines are entered under control of the NUMBER command, p is always set to the most recent line entered during this process, even though CTS has generated (and perhaps displayed) the line number for the line being keyed in. If i is not specified, CTS uses 100. If j is not specified, CTS uses 10, unless i has been coded with an * (see following paragraph). Thus, the simplest NUMBER command (and probably the most common) is illustrated by the following sequence:

```
->N
100 >LINE 1.
110 >LINE 2.
120 >
```

If i is coded with an *, i is taken to be the last line number produced by a previous NUMBER command plus the increment j from the present number command. If, in addition to coding i with an *, j is omitted, the j from the most recent NUMBER command is used. Note the difference from the case where i is not coded with an *, in which an omitted j parameter is assumed to be 10. This is useful in resuming the NUMBER command after an interruption to do some editing or to enter a line which was omitted. Thus, if the sequence of the above example is terminated before generating any data for line number 120, and the * option is used on the next NUMBER statement, the following sequence would result:

```
->N *
120 >LINE 3.
130 >LINE 4.
140 >
```

Here 10 is taken for j because it was the value of j in the previous NUMBER command. If this sequence is terminated as before and the * is used for i but with an explicit j, this sequence would result:

```
->N *,5
135 >LINE 5.
140 >LINE 6.
145 >
```

If an ! is coded for the i parameter, numbering begins at the highest line in the working area plus the increment j. This will resume numbering at a different line than the * if the last use of NUMBER was to generate lines that were not the highest in working area or if manual data input has been done. This is useful in initiating the number mode to append new data lines to the working area.

The behavior of CTS in generating line numbers is not modified by what is done between terminating one NUMBER command and issuing another. The contents of f may have been completely changed or nonexistent when a NUMBER command with an asterisk is issued, but the sequence will still continue as indicated above. If line numbers are generated which already exist in f, the old lines will be replaced as the new ones are submitted.

If the i or j parameter is coded with a string which is not a number, the standard values of 100 and 10 are assumed, since this string is interpreted as k.

The parameter k determines whether CTS displays the line numbers it is assigning to the lines. If k is coded with an N (or any string beginning with N), CTS does not display them. If k is coded with a P (or any string not beginning with N), CTS displays them. If k is not specified (left blank), the option is governed by the most recent NUMBER command in which k was specified. If k has never been specified, P is assumed, and the numbers are displayed. When CTS displays the line numbers, they become part of the solicitation sequence, as in the above examples. Notice that the character immediately after the > is the first character of the string which constitutes the contents of the line.

The above rules are illustrated by the following example:

```
->N 10,5 N
>LINE 1
>LINE 2
>LINE 3
>*PRINT A
10 LINE 1
15 LINE 2
20 LINE 3
END OF FILE
->N *
>LINE 4
>LINE 5
>*PRINT A
10 LINE 1
15 LINE 2
20 LINE 3
25 LINE 4
30 LINE 5
END OF FILE
->N 50 P
50 >LINE 6
60 >LINE 7
70 >*PRINT 25+
```

```
25 LINE 4
30 LINE 5
50 LINE 6
60 LINE 7
END OF FILE
->
```

The first NUMBER command in this example establishes the sequence and turns off the display of line numbers. The *PRINT commands (see 4.1.1) terminate the effect of the NUMBER command then in force (see the discussion later in this section).

The next NUMBER command continues the sequence because of the *. Because the parameter k is blank, the display of line numbers is governed by the previous NUMBER command. The third NUMBER command establishes a new sequence and turns the display of generated line numbers back on. Notice that this time the undefined j parameter was taken to be 10, but in the previous NUMBER command, the omitted j (in the presence of *) was taken from the preceding NUMBER command.

When the NUMBER command is not in effect, CTS expects a command to be entered. If the first character entered is a digit, CTS assumes that a line of data is being entered instead. The presence of a line number beginning in the first column overrides the normal sequence of events – the interpretation of a command.

On the other hand, when the NUMBER command is in effect, CTS expects a line of data to begin in the first column after the solicitation sequence. If a CTS instruction is entered, CTS has no way of determining that this string of characters is not another data line. To break into the sequence of input established by a NUMBER command, and submit a CTS command rather than a line of data, simply prefix the CTS command by an *. The * in the first column after the > performs the same function when CTS is under control of a NUMBER command as the line number does when CTS is not. In the case of the *, however, the effect of the NUMBER command is terminated. To reestablish it, a new NUMBER command must be submitted. (However, the special situation which arises when a prescan module is in control is explained in the text which follows.)

The previous example illustrated the termination of NUMBER commands with *PRINT commands. If the * appears immediately following the >, the effect of the NUMBER command is terminated, as in the following example:

```
->N
100 >LINE 1
110 >LINE 2
120 >*XYZ
<87> INVALID COMMAND
->
```

The return to the normal solicitation sequence signals the end of automatic line number generation.

If an illegal line number is generated by CTS while it is automatically generating line numbers, it prints a warning at the time the line number is generated and before soliciting the input for that line. The following example illustrates the point:

```
->N  262130,5
262130 >LINE 1
262135 >LINE 2
262140 >LINE 3
<102> LINE NUMBER LIMIT EXCEEDED
->
```

All three lines of data were entered into f. The line pointer is set to 262140 (the last line entered) and automatic line number generation is terminated.

The number within the "<>" characters preceding the error messages in the examples identifies the message. These numbers may be used with the EXPLAIN command (see 9.3.2) to request more information about the cause of the error and suggested action.

The NUMBER command also operates when a prescan module is in control. In this case, however, there are some differences. When a program line is entered while a prescan module is in effect, the module normally checks each line as it is entered for local errors in syntax (errors which do not depend on relationships between statements). If such an error is detected in a line which was submitted while a NUMBER command is in effect, the prescan modules temporarily suspend the NUMBER command, display a diagnostic message, return the solicitation character, and wait for the line to be corrected. The line may be corrected either by typing the line number and the correct line or by an editing command such as CHANGE (see 5.2.2). The corrected line is accepted, entered into f, and the NUMBER command is reinstated to generate the next line number. If the error is not corrected immediately, but other CTS commands are entered, the NUMBER command is terminated.

The following example illustrates suspension and reinstatement of a NUMBER command:

```
->BASIC
BBASIC 9R1
>>NEW ABC
>>N
100 >A=1.2
110 >B=2.3
120 >C=AB
120 AN EXPRESSION CONTAINS AN IMPROPER ITEM BEGINNING [AB] IN 120.
>>120 C=A+B
130 >
```

After the automatic numbering was initiated under control of the BASIC prescan module, an erroneous line was entered (line 120). The line was rejected and automatic numbering was suspended. After the line was corrected, it was accepted, placed into f, and the automatic numbering resumed.

2.3.3. Termination of Automatic Line Numbering – MANUAL

Syntax: MANUAL

Abbreviation: MAN

Function: To terminate the effect of a NUMBER command.

The NUMBER command (see 2.3.2) conditions CTS to generate line numbers automatically and to expect a line of data as the normal input from the terminal. This condition can be removed by submitting a line with an asterisk immediately following the ">", followed by a CTS command. If the CTS command submitted in this way is a MANUAL command, the removal of the automatic numbering is the only effect. Any other CTS command, in addition to removing the condition, performs its function. In this sense, the MANUAL command may be thought of as a dummy CTS command.

2.3.4. Defining Tab Stops and Character – TAB

Syntax: TAB [b1 c1,c2,c3,...[b2 c1,c2,...[b3...]]]

Abbreviation: None

Function: To define characters to be interpreted on data input as tab characters and the columns in which tab stops are set and to disable the values established by the previous TAB command. Tab characters are also recognized by the FIND and GENERATE commands.

When entering data lines after a TAB command, an occurrence of the characters defined by the b parameters will cause a skip to the next column with a tab stop defined, inserting spaces in the columns skipped. If the tab character occurs past the last tab stop, the tab character is replaced with a blank, and a warning message is printed. Up to four tab characters may be defined with up to twelve tab stops distributed among them.

If all parameters are omitted, the tab feature is disabled.

The following example illustrates the use of the TAB command:

```
->TAB # 10,20,30
->10 1#2#3#4
->PRINT
10 1          2          3          4
->TAB
->20 1#2#3#4
->PRINT
20 1#2#3#4
->TAB ; 5 : 10 # 15 @ 20 ! 3,5,7
<39> TAB TABLE IS FULL
->
```

The last TAB command attempted to define too many (five) tab characters, causing the diagnostic and disabling the TAB feature.

2.3.5. Restricting Certain CTS Commands – ASSUME EDIT

Syntax: ASSUME EDIT [ON/OFF]

Abbreviation: A EDI

Function: To restrict CTS commands for Text Editor (@ED processor) users.

The ASSUME EDIT command can be used to aid former users of the Text Editor (@ED processor) by restricting certain commands or syntax. (See SPERRY UNIVAC Series 1100 Text Editor (ED Processor), Programmer Reference, UP-8723 (current version).) ASSUME EDIT ON causes the following to occur:

1. A blank data line (a line number followed only by blanks) cannot be entered.

   NOTE:

   A line number followed by a tab character will result in a blank data line.

2. A blank line terminates number mode. See the note in item 1.

3. The R and R* abbreviation for RUN may not be used in command mode. The abbreviations are still allowed in a subroutine.

ASSUME EDIT or ASSUME EDIT OFF reinstates the original condition of allowing these items.

2.4. Prescan Modules

For convenience, a prescan module should be used whenever creating a program in a language having an associated prescan module. Currently, prescan modules are available for BASIC, FORTRAN, and COBOL.

When a prescan module is called, it sets the assumed compiler (see 2.1) to the correct compiler and options. It checks CTS commands to see that their results are consistent with the language which is in effect. For example, if an OLD command attempts to bring into f an element with a type not consistent with the language of the prescan module, it will suspend the command, explain the situation, and ask if this element is to be loaded. An affirmative response will allow the OLD command to continue, but will keep the assumed compiler set to its own standard.

Prescan modules offer line–by–line local syntax checking as lines are entered, global checking before compiling or saving a program, and various aids to the creation of a program, such as formatting, automatic line continuation, and abbreviation of long key words. All of the prescan modules do not offer all of these features. Each prescan module implements features advantageous to creation of programs in its own language.

A prescan module is invoked by the CTS statement which calls it. It is terminated by the CLEAR command (see 2.4.8), DATA command (see 3.7), or by invoking another prescan module. Most of the CTS commands may be used while under control of a prescan module, but the operation of some of them may be restricted or modified by particular prescan modules.

Lines which are marked in error by a prescan module and not corrected are printed with an asterisk preceding them. In all other ways they are the same as other lines. The asterisk is just a reminder that the line had a syntax error.

A prescanner may have commands which are unique to it. If one of these commands is entered when the prescanner is not active, CTS will print the error: <87> INVALID COMMAND.

2.4.1. BASIC

Syntax: BASIC

Abbreviation: BAS

Function: To place the BASIC prescan module, BBASIC, in control.

As a line of a BASIC program is entered, CTS performs a syntax check on that line of code. This is a noncontext syntax check. That is, it looks at the single line of code, and that line only, and tries to find all possible things wrong with it.

This syntax checking is invoked by informing CTS that the BASIC prescan is to be used on the current program. This is done by the following command:

```
->BASIC
BBASIC 9R1
>>
```

Notice in this example that the input solicitation following the BASIC statement has changed from the hyphen (-) to a greater than symbol (>). This is a reminder that future lines of code are under the control of a prescan module. Any new line of code entered will now be checked by the appropriate syntax analyzer (in this case BBASIC).

The BASIC command not only will invoke the syntax analysis by the BASIC syntax checker, but it will also indicate that the BASIC compiler is to be used when this program is subsequently run.

Suppose the system is automatically numbering lines of code via the NUMBER command and a line of code which is in error is entered to BASIC. The prescan module will respond with a diagnostic indicating the source of the error. It will then solicit, not with the new line number, but with the greater than (>) symbol. This is a reminder to use an editing command, say the CHANGE command, to correct the previous error line before entering any new lines of code.

Once the line in error is corrected, CTS will continue with the line number solicitation in the proper sequence.

For example:

```
120 >PRNT A,B
120 THE STATEMENT CONTAINS NO RECOGNIZABLE INSTRUCTION IN 120.
>>CHANGE 'RN'RIN'
120 PRINT A,B
130 >END
```

This example shows the CHANGE command being used to correct a typing error. Notice that the second line is a diagnostic message printed by the syntax analyzer. The third line is the CHANGE command issued by the programmer. The fourth line is the visual check which shows to the programmer the actual correction as it was made by the system. The fifth line shows that the system has gone back into the number mode, picking up with the next available number.

Notice in the preceding example that on the CHANGE command no line number was specified. This is because the default value for the line number in the CHANGE command is the current line. When getting a diagnostic from the prescan analyzer and correcting the line in error with the CHANGE command, the CHANGE command does not need the line number since it pertains to the error line.

There is another way to correct errors detected by the syntax analyzer.

For example:

```
120 >C=AB
120 AN EXPRESSION CONTAINS AN IMPROPER ITEM BEGINNING [AB] IN 120.
>>120 C=A+B
130 >
```

In this example the first line was typed in error. Line two contains the diagnostic from the BASIC syntax analyzer, and the editing command has been solicited in line three with the greater than symbol (>). In this case, the line is so short that it is easy to retype. CTS knows this to be a line of code because it starts with a line number. It will cause the syntax analyzer to check this line and then, as illustrated in line four, CTS returns to the line number mode, soliciting with the next available line number.

The preceding two examples illustrate two different means of correcting errors in lines of code that have been checked by the syntax analyzer. One is simply to retype the line, the other is to use an editing command to effect the correction in the line. In both cases, CTS will return to the automatic line number mode.

2.4.1.1. Scanning for American National Standard BASIC—ANSI

Syntax: ANSI [ON/OFF]

Function: Turn the scanner and corresponding compiler on for American National Standard Minimal BASIC.

If the ANSI ON command is used, both the BASIC prescanner and the BASIC compiler will scan and execute programs with the American National Standard Minimal BASIC standard being enforced. This may not be completely compatible with existing BASIC programs. For differences see SPERRY UNIVAC Series 1100 UBASIC/BBASIC, Programmer Reference, UP-7925 (current version).

If the ANSI OFF command is used, the prescanner and compiler remain completely compatible with current programs and older versions of BASIC.

The default for BBASIC level 9R1 is ANSI OFF.

2.4.2. FORTRAN

Syntax: FORTRAN [ASCII/FIELDATA]

Abbreviation: FOR [A/F]

Function: To place a FORTRAN prescan module in control.

The FORTRAN command activates the ASCII FORTRAN prescanner (BFTN) or the Fieldata FORTRAN prescanner (BFOR), changes the assumed compiler to FTN or RFOR, respectively, and, if necessary, changes the mode of the working area to ASCII ON or ASCII OFF, respectively. The prescanner performs line-by-line local syntax checking as lines are entered or edited, global checking (BFOR only), compiling or saving a program, line formatting, automatic line continuation, and expansion of abbreviated keywords. ASCII images may be prescanned by BFTN. Fieldata images will be converted to ASCII for BFTN. Only Fieldata images may be prescanned by BFOR. Because they are different prescanners and they are checking for different syntax, they will behave differently at times.

If the keyword ASCII or FIELDATA is omitted, the default is ASCII. ASCII may be abbreviated A and FIELDATA may be abbreviated F.

CTS will perform a syntax check on each line of code written in the FORTRAN language. This is a noncontext syntax check.

FORTRAN involves continuation lines. Thus, the syntax check by the FORTRAN analyzer (BFTN) is performed on a single statement which may actually comprise several lines of information. That is to say, more than one line may compose a single FORTRAN statement.

The FORTRAN prescanner provides automatic formatting of source statements. It also allows abbreviations of key words and performs an analysis on a whole program at appropriate times.

The syntax analyzer scans to find all possible things that can be wrong with a single FORTRAN statement.

This syntax checking is accomplished by informing CTS that the FORTRAN prescan is to be used on the current program. This is done by the following command:

```
->FORTRAN ASCII
ASCII FORTRAN PRESCAN 2R1A
>>
```

or if the desired mode is Fieldata:

```
->FORTRAN FIELDATA
FD FORTRAN PRESCAN 5R1
>>
```

The FORTRAN statement changes the input solicitation character from the hyphen (-) to the greater than symbol(>). This is a reminder that future lines of code are under the control of a prescan module.

The FORTRAN command not only will invoke the syntax analysis by the FORTAN syntax checker, but will also indicate which FORTRAN compiler is to be used when this program is subsequently run. This compiler is ASCII FORTRAN (FTN) or Reentrant FORTRAN (RFOR).

If the wrong syntax is being checked, typing ASCII or FORTRAN FIELDATA will activate the other syntax analyzer.

```
->FORTRAN FIELDATA
->ASSUME ASCII OFF
FD FORTRAN 5R1
>>ASCII
ASSUME ASCII ON
ASCII FORTRAN PRESCAN 2R1A
>>FORTRAN FIELDATA
ASSUME ASCII OFF
FD FORTRAN 5R1
>>
```

Note that the mode of the working area was automatically changed to the mode required by the syntax analyzer. This is indicated by the message ASSUME ASCII ON or ASSUME ASCII OFF.

If a mistake is made in a line of code while the system is automatically numbering lines, the prescan module will respond with a diagnostic indicating the source of the error. CTS will then solicit, not with the new line number, but with the greater than symbol (>). This is a reminder to use an editing command, perhaps the CHANGE command, to correct the previous error line before entering any new lines of code.

Once the line in error is corrected, CTS will continue with line number solicitation in the proper sequence.

For example:

```
->FOR
ASSUME ASCII ON
ASCII FORTRAN PRESCAN 2R1A
>>NUM 110
110 >PRNT 10,A
REJECTED:  STATEMENT IS OF UNRECOGNIZABLE TYPE
>>C 'RN'RIN'
110       PRINT 10,A
120 >
```

In this example the second line is a diagnostic message printed by the syntax analyzer. The third line is the CHANGE command (abbreviated by "*C*") issued by the programmer. The fourth line is the visual check which shows the programmer the correction made by the system. Note that automatic formatting has occurred (see 2.4.2.1). The fifth line shows that the system has gone back into the number mode, picking up with the next available number.

In the preceding example, the CHANGE command has not listed a line number. This is because the default value for the line number in the CHANGE command is the current line. When a diagnostic from the prescan analyzer is printed and that line is being corrected with a CHANGE command, the line number is not necessary.

The line could also be entirely retyped. The procedure for this method of correction is explained in 2.4.1.

Either of these two methods is available to correct errors in lines of code. One method is simply to retype the line. The other is to use an editing command to correct the line. In both cases, CTS will return to the automatic line numbering mode.

2.4.2.1. Automatic Formatting

In batch programming, card columns 1 to 5 of a FORTRAN statement may contain a statement number, column 6 is used as a continuation column, and column 7 begins the actual FORTRAN statement. CTS retains the concept of these columns, but it aids the FORTRAN programmer by tabbing automatically to column 7 and beginning the FORTRAN statement there. It does this if the first character typed is not a number, or is an exclamation point (!) or an ampersand (&).

If the first character is a number, it must be a statement number, and it will be left justified in columns 1 to 5.

If it is an ampersand (&), it must be part of a continuation statement. This is discussed in 2.4.2.2.

If the first character typed is an exclamation point, the statement is taken as a comment, and the exclamation point will be replaced by an askerisk (*) in column 1. The exclamation point in column 1 replaces the letter C in column 1 indicating that the statement is a comment.

Notice the input of the following program:

```
->FOR
ASCII FORTRAN PRESCAN 2R1A
>>NEW DEF
>>N
100 >A=1.2
110 >PRINT 10,A
120 >10 FORMAT (E14.8)
130 >END
140 >
```

The following is a printing of the program DEF. Notice that all of the statements begin in column 7, with the statement number in line 120 appearing left-justified in columns 1 to 5. Note also the appearance of the line numbers, with a blank following them. Again, these are CTS line numbers, which are not part of the line image. They have no specific relationship to the FORTRAN statement number.

```
>>PRINT ALL
100       A=1.2
110       PRINT 10,A
120 10    FORMAT (E14.8)
130       END
END OF FILE
>>
```

2.4.2.2. Continuation Lines

FORTRAN allows the use of more than one line in a single statement. This is traditionally denoted in batch processing by a nonblank punch in column 6 of the continuation card; that is, the second, or succeeding card in the statement. Continuation statements may be entered by placing an ampersand (&) as the last character of the line to be continued, and an ampersand as the first character of the continuation line in input mode.

For example:

```
100 >A=1.23&
-CONTINUE- (REMEMBER THE '&')
110 >&456
120 >
```

The prescan module will strip the ampersand from the end of the line to be continued and will place it in column 6 of the continuation line.

For example:

```
180 >D=4&
-CONTINUE- (REMEMBER THE '&')
190 >&5.6&
-CONTINUE- (REMEMBER THE '&')
200 >&78
210 >*P  180,200
180       D=4
190      &5.6
200      &78
>>
```

After a line to be continued, CTS displays the message –CONTINUE– (REMEMBER THE '&'), and then solicits with the next available line number. This is a reminder that the previous line is to be continued. If each line to be continued is followed by a continuation line, the entire statement will be checked for syntax.

Since lines do not have to be entered in order in CTS, it is not necessary to follow lines to be continued with continuation lines. This is allowed, but the syntax analyzer gives up at that point.

For example:

```
210 >E=5.6&
-CONTINUE- (REMEMBER THE '&')
220 >78
CONTINUATION EXPECTED -- PREVIOUS INCOMPLETE STATEMENT NOT SCANNED
WARNING-LINE CONTAINS ONLY STATEMENT LABEL
230 >*P 210,220
210      E=5.6
220 78
>>
```

In this example an ampersand (&) should have preceded the 78 in line 220. Since it did not, line 220 cannot be assumed to be a continuation of the previous lines. Therefore, the syntax analyzer has given the diagnostic that a continuation was expected and that it is not going to scan the previous statement. Line 220 is taken as a separate statement, and the 78 is regarded by the syntax analyzer as a statement number. The syntax analyzer has also issued a warning that the line contains only a statement label. Statement 220 is printed with 78 appearing in columns 1 and 2.

2.4.2.3. Abbreviated Key Words

Many of the key words of FORTRAN can be abbreviated.

Any key word of six or more characters may be abbreviated by typing:

first letter :*last letter*

except for the following key words:

DELETE,DECODE (D:E is DEFINE)
ENDFILE, ENCODE (E:E is EQUIVALENCE)
START EDIT, STOP EDIT
DOUBLE PRECISION (D:N is DIMENSION, D:P is DOUBLE PRECISION)

The following are acceptable abbreviations:

A:L = ABNORMAL*		F:T = FORMAT	
A:N = ASSIGN		I:C = INTRINSIC**	
B:A = BLOCK DATA		I:E = INCLUDE	
B:E = BACKSPACE		I:R = INTEGER	
C:E = CONTINUE		I:T = IMPLICIT	
C:N = COMMON		L:L = LOGICAL	
C:R = COMPILER		N:T = NAME LIST	
C:X = COMPLEX		P:M = PROGRAM**	
D:E = DEFINE		P:R = PARAMETER	
D:N = DIMENSION		R:D = REWIND	
D:Y = DISPLAY**		R:N = RETURN	
D:P = DOUBLE PRECISION		S:E = SUBROUTINE	
E:E = EQUIVALENCE		T:F = TRACE OFF**	
E:L = EXTERNAL		T:N = TRACE ON**	
F:N = FUNCTION			

** – BFOR only*
*** – BFTN only*

Expansion of the abbreviations is performed at the time of entry of the line into f.

For example:

```
->FOR ASCII
ASCII FORTRAN PRESCAN 2R1A
>>N
100 >A(10)
REJECTED : STATEMENT IS OF UNRECOGNIZABLE TYPE
>>C /A/D:N A/
100         DIMENSION A(10)
110 >D:N (10)
<571> REJECTED : ( APPEARS WHERE A VARIABLE NAME IS NEEDED
>>C /(/B(/
110         DIMENSION B(10)
120 >
```

2.4.2.4. Automatic Global Syntax Analysis (BFOR Only)

Fieldata FORTRAN syntax analysis goes a step beyond the statement-by-statement checking described in 2.4.1. It will perform an analysis on the entire program. This is called a global syntax analysis.

BFOR scans the entire program for errors which may occur due to conflicts between two or more statements, or between a statement and the entire program.

Examples of the types of errors which this scan can discover are unreferenced variables, unassigned variables, missing or redundant END cards, and statements which cannot be reached.

This scan will be automatically activated after a SAVE, REPLACE, or RUN command.

For example:

```
>>FOR FIELDATA
FD FORTRAN 5R1
>>NEW DEF
>>N
100 >A=B
110 >D:N C(10)
120 >GO TO 20
130 >D=1.2
140 >30 END
150 >STOP
160 >*SAVE
DO YOU WANT A GLOBAL SCAN?  >YES
<611> ALLOWED(130) : THIS STATEMENT CANNOT BE REACHED.
<635> ALLOWED(140) : END CARD APPEARS BEFORE TRUE END OF DECK.
<615> ALLOWED(150) : MISSING END CARD.
<606> ALLOWED(100) : 'B' IS REFERENCED BUT NEVER ASSIGNED A VALUE.
<608> REFUSED(120) : LABEL '20' IS NOT DEFINED.
>>P A
100       A=B
110       DIMENSION C(10)
120       GO TO 20
130       D=1.2
140 30    END
150       STOP
END OF FILE
>>
```

Note the question, DO YOU WANT A GLOBAL SCAN? A positive answer will activate the global scan. It may be bypassed in cases where it is relatively certain there are no global errors, or in cases where a scan is not wanted (e.g., saving a partial program).

Notice when the program is listed that no alteration to the program has been made, i.e., no statements have been eliminated. Thus, the lines must be revised to eliminate the error conditions.

BFTN will scan the entire program, checking each line for syntax errors but not for global errors. BFTN does not do a full program syntax check for SAVE, REPLACE, or RUN commands unless the command GLOBAL has been entered. If it has, the query:

```
DO YOU WANT ALL LINES SCANNED? >
```

will be printed. This question is different than the one given by BFOR to emphasize that a global scan is not done.

The special command:

```
>>NODIAG
```

turns off the local syntax scanning. The statement command:

```
>>DIAG
```

turns it on again.

The special command:

```
>> ECHO
```

causes each statement to be displayed in its reformatted form with abbreviations expanded. The command:

```
>> NOECHO
```

discontinues this line-by-line display.

NOTE:

These four commands - DIAG, NODIAG, ECHO and NOECHO - are part of the FORTRAN syntax analyzer. They are not available to other prescan modules, or to CTS when the prescanner is not active.

2.4.2.5. Global Syntax Analysis - CHECK

The FORTRAN prescanner command CHECK causes a line-by-line syntax check (BFTN) or a global syntax check (BFOR). This command can be entered whenever the FORTRAN prescanner is active and the results are identical to the automatic global syntax analysis described in 2.4.2.4.

2.4.2.6. Controlling Automatic Global Syntax Analysis (BFOR Only)

The BFOR prescanner commands GLOBAL and NOGLOBAL, respectively, turn on and turn off automatic global syntax analysis.

```
-> FOR F
FD FORTRAN 5R1
>> NOGLOBAL
>> SAVE TEST
>>
```

The SAVE command did not result in the global scan query since the NOGLOBAL command was entered.

NOTE:

Whenever a prescanner is called, the DIAG, NODIAG, ECHO, NOECHO, GLOBAL, and NOGLOBAL command modes are reset to their default values.

See 2.4.7 for a description of the SYNTAX command.

2.4.3. COBOL

Syntax: COBOL

Abbreviation: COB

Function: To place CTS under control of the COBOL prescan module, BCOB.

BCOB is the COBOL syntax prescan module. Its function is to aid the time sharing user in constructing, editing, and syntax debugging COBOL programs from a demand terminal.

The syntax analysis is compatible as a user option with the ASCII COBOL compiler and also supports the ASCII COBOL Data Manipulation Languages.

BCOB operates in three modes: conversational mode for development of new programs, program file mode for syntax debugging existing programs, and edit mode for inserting or changing individual lines.

BCOB provides automatic line formatting which conforms to the Margin A and Margin B requirements of COBOL, and also provides abbreviation expansion to facilitate use of a COBOL shorthand. A system-defined shorthand is provided with the BCOB processor. In addition, the system abbreviation set may be optionally replaced or enhanced by user-defined replacement sets. The use of these two features greatly reduces the quantity of information the user inputs from the terminal.

2.4.3.1. Operational Description

BCOB performs two syntax scans. The first is active at the time insertions, updates or corrections are interactively supplied to the COBOL program being developed. A limited line-by-line type analysis is performed as each image is encountered. During this local scan, the line is examined out of context, checking for misspelled words, keywords, improper use of reserved words, etc. The second scan, which is entered when a total global scan of a COBOL program is requested, is a full contextual syntax analysis providing diagnostics comparable to the ASCII COBOL compiler.

Either an ASCII COBOL or ASCII with DML COBOL syntax scan can be chosen by the following BCOB commands:

ASCII [*DML*]

The assumed compiler (see 2.1) is set to ACOB,SBE. If the compiler to be used is known to the operating system by another name, change the assumed compiler either with the CTS ASSUME COMPILER command (see 6.2.1) or the BCOB command:

COBOL k

where *k* is the name of the compiler and the options.

For example:

```
>>COBOL      ACOB,SBE
```

BCOB permits the use of the NUMBER command (see 2.3.2). In the absence of this command, it will generate line numbers starting with 100, with an increment of 10. The increment can be changed with the BCOB command:

```
STEP i
```

where *i* is the increment. In any case, the COBOL line number will be the same as the CTS line number.

In addition to the CTS commands, BCOB has its own command set. These include options for COBOL line sequencing (columns 1–6 of each image) and automatic line continuation when an inout line extends beyond column 72. Extended input may also be optionally truncated beyond column 72. The extent of the error list can be controlled with BCOB commands. For example, all remarks can be suppressed on request to see only the serious and fatal errors, etc. Of primary importance are the commands which control abbreviation expansions. There are three such commands: LOAD, ERASE, and replace or RPL. The RPL command has the form "*RPL *pseudo-text-1* BY *pseudo-text-2*". *Pseudo-text* may consist of any character string. If it does not consist of a single COBOL word or identifier it must be bounded by the pseudo-text delimiter "==". Whenever a match occurs between *pseudo-text-1* and the text, the corresponding *pseudo-text-2* is inserted into the program replacing the text corresponding to *pseudo-text-1*. The LOAD command causes BCOB to load an element consisting entirely of BCOB commands. For purposes of defining a COBOL shorthand, this element could consist of a series of replace (RPL) commands. Initially, BCOB loads the system defined shorthand. A LOAD command then adds to this standard abbreviation set. The ERASE command is used to erase the currently effective set of replacement directives (RPL commands), or a specified replacement command identified by *pseudo-text-1*.

Replacement commands have no effect in program file mode.

The following is an example of a replacement set:

```
>>RPL P BY PICTURE
>>RPL EXAMINE BY INSPECT
>>RPL LP BY ==LINE PLUS==
>>RPL NGNP BY ==NEXT GROUP IS NEXT PAGE==
>>RPL JUSTIFIED BY == ==
```

This set would affect input to BCOB as follows:

Input:

```
>>100    02 ITEM-A P X(32).
>>110    02 ITEM-B P 9(2) JUSTIFIED.
>>120    EXAMINE FIL REPLACING
>>130  01 TYPE CF LP 3 NGNP.
```

Resulting text:

```
>> LIST
100 000100      02 ITEM-A PICTURE X(132).
110 000110      02 ITEM-B PICTURE 9(2).
120 000120      INSPECT FIL REPLACING
130 000130 01 TYPE CF LINE PLUS 3 NEXT GROUP IS NEXT PAGE.
END OF FILE
>>
```

Five levels of diagnostics exist in BCOB. In order of increasing severity they are REMARK, WARNING, MINOR, SERIOUS, and FATAL. To allow only certain levels of diagnostics, the command DIAG [k] [p] is available, where k specifies which level of diagnostics to allow:

k = ALL	All messages are printed.
k = W	WARNING, MINOR, FATAL and SERIOUS errors are flagged.
k = M	MINOR, FATAL, and SERIOUS errors are flagged.
k = F	FATAL and SERIOUS errors are flagged.
k = S	SERIOUS errors are flagged.
k = NONE	No errors are displayed.

The parameter p specifies in which phase to apply the constraint. If p is not present, the constraint applies to both phases 1 and 2.

BCOB will inhibit the compilation of any program with errors of severity MINOR or greater and print the message:

```
COMPILATION REFUSED BECAUSE OF ABOVE ERRORS
```

The commands ECHO, ECHO 2, and NOECHO may be used to see what line BCOB is currently scanning. NOECHO turns off this feature. ECHO and ECHO 2 turn it on for a phase 1 and phase 2 scan.

Formatting of input in BCOB is done with the FMT and NOFMT commands. The default is FMT. If automatic formatting is requested (with a FMT command) the general input format is:

lineno CAB *text*

The parameters are defined as:

lineno	A number associated with both the CTS line number and the COBOL line number (columns 1–6) of the COBOL image. It may be typed in directly if in the MANUAL mode, or generated and typed by CTS if in NUMBER mode.
C	If C is nonalphabetic and not a space, the text beginning in A is placed left justified in COBOL area A. The C character is placed in column 7. If C is a dollar sign ($), it is converted to an asterisk (*) and placed in column 7. If C is alphabetic, the text is placed left justified in COBOL area A. C must not be numeric.

A — If C is a space and A is not a space, then text beginning in A is placed left justified in COBOL area A (column 8).

B — If a space is found in C and A, then text beginning in B is placed left justified in area B (column 12).

BCOB will insert sequence numbers into the COBOL sequence-number field (columns 1–6). These numbers will be the same as the CTS line numbers. The commands controlling this function are CSEQ and NSEQ.

Automatic continuation is provided by the CHOP and NOCHOP commands. NOCHOP requests automatic continuation. If it is specified and an image extends beyond column 72 after formatting and/or replacement, BCOB will automatically create an additional line or lines for the remainder of the text. Each additional line will be numbered one greater than its predecessor, unless the increment is changed with the STEP command. Automatic continuation will be applied to all edit mode input except lines which end with an incomplete nonnumeric literal. The default is NOCHOP.

Automatic continuation is disabled by the CHOP command. If CHOP is requested and an image expands beyond column 72, the following message will be printed:

```
COLUMN 72, TEXT BEGINNING 'word' TRUNCATED.
```

where word was the first complete COBOL word, delimited by spaces, which did not fit on the line. All succeeding text characters are discarded.

If, however, the image would end with an incomplete nonnumeric literal, the message:

```
UNFINISHED LITERAL, CONTINUE AFTER 'string'
```

will be printed, where string is the contents of columns 65 to 72 of the edited image. The literal should be continued on the next line.

Complete control of continuation is gained by the command CHOP ASK. In this mode, right margin overflow will cause BCOB to print:

```
COLUMN 72, CONTINUE TEXT AFTER 'string'?
```

where string is the last 8 characters of the edited image (columns 65 – 72). There are three responses to this question:

ASIS — The image will be left as it is. The user must continue it.

CHOP — Characters beyond the last complete COBOL word will be erased, as in CHOP mode.

NOCHOP — This response, or a blank return, continues the image automatically, as in NOCHOP mode.

If an image ends with an open literal in edit mode, the following message will be printed:

```
LITERAL BEGINNING 'string' CLOSED BY BCOB AT END
```

where string is the first eight characters of the unfinished literal. BCOB appends a closing quote at the end of the input line and performs the normal edit and overflow actions.

If an input line contains an ampersand (&) as the last nonblank character, regardless of context, BCOB takes this as a request to extend this line. It solicits the extension with the message:

```
CONTINUE:
```

The extension of the line just typed is the expected response. The first character of the response will be written over the ampersand in the extended line and the remaining characters will follow it in sequence. If the extension itself ends with an ampersand, the process will be repeated until a complete line or a maximum of 400 characters has been collected. If the extended line exceeds 400 characters, a message will be printed:

```
MAXIMUM EXTENDED LINE - USED FIRST 400 CHARACTERS
```

and the first four hundred characters will be taken as a complete line.

If a carriage return is given as the response, the line will be complete as is.

2.4.3.2. Modes of Operation

2.4.3.2.1. Edit Mode

BCOB is always initially in edit mode. This mode assumes the individual lines of a program will be entered directly from the terminal. Automatic line continuation, line formatting and sequencing are assumed. Each line is scanned as it is entered. This operation, known as a phase 1 scan, is a mode in which lines are analyzed out of context with the rest of the program. Therefore, there is no recognized syntactic order for consecutive entries. For example, procedural statements could be followed by the Data Division entries required to define referenced working storage data items.

The following input could be entered into the working area:

```
>>210    02 INX USAGE IS INDEX.
>>400    OPEN INPUT IN-FILE
>>50   ENVIRONMENT DIVISION.
```

with the resulting text:

```
>>LIST
50 000050 ENVIRONMENT DIVISION.
210 000210      02 INX USAGE IS INDEX.
400 000400      OPEN INPUT IN-FILE
END OF FILE
>>
```

after adding the lines:

```
>>1   IDENTIFICATION DIVISION.
>>200   01 WORK-A PIC X(50).
>>450    EXIT.
```

The program element would be:

```
>>LIST
1 000001 IDENTIFICATION DIVISION.
50 000050 ENVIRONMENT DIVISION.
200 000200 01 WORK-A PIC X(50).
210 000210      02 INX USAGE IS INDEX.
400 000400      OPEN INPUT IN-FILE
450 000450      EXIT.
END OF FILE
>>
```

An entire program may be entered in this manner within edit mode or could be entered in conversational mode.

2.4.3.2.2. Conversational Mode

In conversational mode, BCOB solicits input directly and constructs the source images itself. BCOB treats the syntactic definition as a template, supplies the key words, and requests the blanks to be filled in item by item. If the clause indicated by the BCOB-supplied keyword is not wanted, transmitting a blank will send BCOB to the next space in the template. If the clause is required and a blank line is transmitted, BCOB will respond "REQUIRE , *clause?*" and repeat the keywords. The answers to the next several questions may be separated by asterisks (*), in which case BCOB will assimilate the additional information, and proceed to the logically next question. The key to conversational mode operation is that those portions of the COBOL program which tend to be lengthy, mechanical in nature, and prone to error can be developed quickly with a minimum of typing.

Conversational mode is entered whenever BCOB receives a CTS NEW command for development of the Identification and Environment Divisions.

The following is an example of BCOB user interaction on a NEW command:

```
>>NEW TEST
PROGRAM-ID? >TEST
AUTHOR? >JJJ
INSTALLATION? >UNIVAC
SECURITY? >NONE
SOURCE-COMPUTER? >
OBJECT-COMPUTER? >
MEMORY? >
COLLATING? >
SEGMENT-LIMIT? >
SPECIAL-NAMES? >
SELECT? >FILE-1
OPTIONAL? >
ASSIGN? >PRINTER
EXTERNAL NAME? >OUT-FILE
PROCESSING MODE? >
SELECT? >FILE-2
OPTIONAL? >
ASSIGN? >DISC
RESERVE? >
ORGANIZATION? >
ACCESS? >
FILE STATUS? >
SELECT? >*
NEXT LINE NUMBER IS 000280
```

The result of this input would appear in the user working area as:

```
>>LIST
100 000100 IDENTIFICATION DIVISION.
110 000110 PROGRAM-ID.  TEST.
120 000120 AUTHOR. JJJ.
130 000130 INSTALLATION.  UNIVAC.
140 000140 DATE-WRITTEN.  23 JUL 80 AT 10:12:26.
150 000150 DATE-COMPILED.  TODAY.
160 000160
170 000170 ENVIRONMENT DIVISION.
180 000180 CONFIGURATION SECTION.
190 000190 SOURCE-COMPUTER.  UNIVAC-1108.
200 000200 OBJECT-COMPUTER.  UNIVAC-1108.
210 000210
220 000220 INPUT-OUTPUT SECTION.
230 000230 FILE-CONTROL.
240 000240     SELECT FILE-1
250 000250         ASSIGN TO PRINTER OUT-FILE,
260 000260     SELECT FILE-2
270 000270         ASSIGN TO DISC.
END OF FILE
>>
```

A more experienced BCOB programmer would shorten the input:

```
>>NEW
NEW PROGRAM NAME? > TEST
PROGRAM-ID? >TEST*JJJ
INSTALLATION? >UNIVAC
SECURITY? >
SOURCE-COMPUTER? >
OBJECT-COMPUTER? >
SPECIAL-NAMES? >
SELECT? >FILE-1
OPTIONAL? >
ASSIGN? >PRINTER*OUT-FILE**SEQ**
SELECT? >*
NEXT LINE NUMBER IS 000270.
```

The resultant program would appear:

```
>>LIST
100 000100 IDENTIFICATION DIVISION.
110 000110 PROGRAM-ID.  TEST.
120 000120 AUTHOR. JJJ.
130 000130 INSTALLATION.  UNIVAC.
140 000140 DATE-WRITTEN.  23 JUL 80 AT 10:20:19.
150 000150 DATE-COMPILED.  TODAY.
160 000160
170 000170 ENVIRONMENT DIVISION.
180 000180 CONFIGURATION SECTION.
```

```
190 000190 SOURCE-COMPUTER.   UNIVAC-1108.
200 000200 OBJECT-COMPUTER.   UNIVAC-1108.
210 000210
220 000220 INPUT-OUTPUT SECTION.
230 000230 FILE-CONTROL.
240 000240     SELECT FILE-1
250 000250         ASSIGN TO PRINTER OUT-FILE,
260 000260         ORGANIZATION IS SEQUENTIAL.
END OF FILE
>>
```

The user can leave conversational mode and reenter edit mode at any time by responding with an asterisk (*). The program can be repositioned in conversational mode by specifying:

```
*ID    [n]  position to IDENTIFICATION DIVISION.
*ENV   [n]  position to ENVIRONMENT DIVISION.
*FILE  [n]  position to INPUT-OUTPUT SECTION FILE CONTROL.
*IO    [n]  position to I-O CONTROL.
*DATA[n]    position to DATA DIVISION.
*NEXT[n]    position to the next of the above sections following the current position.
```

In all cases, [n] is a user specified line number. Positioning at the beginning of the Data Division is indicated by the *DATA [n] command. At this point, BCOB is in edit mode.

The following interactive sequences indicate the use of BCOB positioning commands.

```
>>NEW
NEW PROGRAM NAME?  >TEST3
PROGRAM-ID?  >TEST3
AUTHOR?  >*ENV
SOURCE-COMPUTER? >
OBJECT-COMPUTER?  >
SPECIAL-NAMES? >*NEXT
SELECT? >*NEXT
I-O CONTROL? >*NEXT
000190 DATA DIVISION.
NEXT LINE NUMBER IS 000200.
>>*LIST
100 000100 IDENTIFICATION DIVISION.
110 000110 PROGRAM-ID.  TEST3.
120 000120
130 000130 ENVIRONMENT DIVISION.
140 000140 CONFIGURATION SECTION.
150 000150 SOURCE-COMPUTER.  UNIVAC-1108.
160 000160 OBJECT-COMPUTER.  UNIVAC-1108.
170 000170
180 000180
190 000190 DATA DIVISION.
END OF FILE
>>
```

```
>>NEW
NEW PROGRAM NAME?  >TEST
PROGRAM-ID?  >*ENV
SOURCE-COMPUTER? >
OBJECT COMPUTER? >
SPECIAL NAMES? >*DATA
000160 DATA DIVISION
NEXT LINE NUMBER IS 000170
>>*LIST
100 000100
110 000110  ENVIRONMENT DIVISION.
120 000120  CONFIGURATION SECTION.
130 000130  SOURCE-COMPUTER.  UNIVAC-1108.
140 000140  OBJECT-COMPUTER.  UNIVAC-1108.
150 000150
160 000160  DATA DIVISION.
END OF FILE
>>
```

In addition to the *NEW command and the positioning commands, BCOB will enter conversational mode for development of SELECT statements, and also for FD, CD, SD, RD, and SA entries.

```
>>NEW
NEW PROGRAM NAME? >TEST
PROGRAM-ID? >TEST
AUTHOR? >*NEXT
SOURCE-COMPUTER? >
OBJECT-COMPUTER? >
SPECIAL-NAMES? >*NEXT
SELECT? >*NEXT
I-O CONTROL? >*NEXT
000190 DATA DIVISION.
NEXT LINE NUMBER IS 000200.
>>250  FD  FILE-A
DEVICE? >TAPE
BLOCK CONTAINS? >5
RECORD CONTAINS? >20
LABELS? >STANDARD
VALUE OF? >
DATA RECORDS? >RECORD1
NEXT LINE NUMBER IS 000300.
>>LIST
100 000100 IDENTIFICATION DIVISION.
110 000110 PROGRAM-ID.  TEST.
120 000120
130 000130 ENVIRONMENT DIVISION.
140 000140 CONFIGURATION SECTION.
150 000150 SOURCE-COMPUTER.  UNIVAC-1108.
160 000160 OBJECT-COMPUTER.  UNIVAC-1108.
```

```
170 000170
180 000180
190 000190 DATA DIVISION.
250 000250 FD FILE-A
260 000260          BLOCK CONTAINS 5 CHARACTERS;
270 000270          RECORD CONTAINS 20;
280 000280          LABEL RECORDS ARE STANDARD;
290 000290          DATA RECORD IS RECORD1.
END OF FILE
>>
```

The following lines can then be added:

```
>>420 REPORT SECTION.
>>430 RD  REPORT1
CODE? >A-CODE
CONTROLS? >FINAL, YEAR, MONTH
MINOR: (WITH CODE) NOT FOLLOWED BY MNEMONIC NAME
PAGE? >120
HEADING? >15
FIRST DETAIL >25
LAST DETAIL? >
FOOTING? >100
NEXT LINE NUMBER IS 000500
>>LIST
100 000100 IDENTIFICATION DIVISION.
110 000110 PROGRAM-ID.  TEST.
120 000120
130 000130 ENVIRONMENT DIVISION.
140 000140 CONFIGURATION SECTION.
150 000150 SOURCE-COMPUTER.  UNIVAC-1108.
160 000160 OBJECT-COMPUTER.  UNIVAC-1108.
170 000170
180 000180
190 000190 DATA DIVISION.
250 000250 FD FILE-A
260 000260          BLOCK CONTAINS 5 CHARACTERS;
270 000270          RECORD CONTAINS 20;
280 000280          LABEL RECORDS ARE STANDARD;
290 000290          DATA RECORD IS RECORD1.
420 000420 REPORT SECTION.
430 000430 RD REPORT1
440 000440     CODE A-CODE;
450 000450     CONTROLS ARE FINAL, YEAR, MONTH;
460 000460     PAGE LIMIT IS 120 LINES,
470 000470          HEADING 15,
480 000480          FIRST DETAIL 25,
490 000490          FOOTING 100.
END OF FILE
>>
```

2.4.3.2.3. Program File Mode

When BCOB receives a CTS OLD or MERGE command, it enters program file mode and performs a phase 1 line-by-line scan of the specified element as it is found in the CTS working area.

Whenever the working area contains a complete COBOL program, a BCOB phase 2 scan can be initialized with a CTS SAVE, RUN, or COMPILE command or with a BCOB CHECK command. The BCOB *CHECK command initiates a full syntax scan but does not request a compilation or have any effect on the status of the program element. CHECK shows errors without requiring an implied full compilation.

2.4.3.3. Summary

The experienced CTS user with COBOL background should have no problem adopting the use of the BCOB command set. Even for the inexperienced, BCOB can very rapidly become a convenient tool for developing and debugging COBOL programs. It eliminates completely the need to move from code sheet to card deck to permanent file storage to subsequent edit and update, and provides the demand user efficient input and immediate diagnostic results.

2.4.4. APL 1100/CTS

APL 1100 can maintain source code by using CTS as a co-routine. APL 1100 does not run under CTS. APL 1100 runs with CTS. The distinction is that there is not the additional overhead of running under different levels of the operating system. The following subsections describe APL 1100 level 7R1. If a later version is available, refer to current APL 1100 user documentation.

The statements and their output are presented in examples as they would be printed out under the APL 1100 system. The input expression is indented eight spaces, and its result is printed at the left margin. Explanation of the result or the statement is made at the right.

For example:

```
        >2 + 6      This is input.
8                   This is output.
```

Functions which have two arguments are called dyadic functions. The addition function is dyadic; one argument (value) is placed on each side of the function name.

```
        >2+ !□
□:    3
8
```

This expression is evaluated in four steps: the quad function obtains a value, factorial uses that value to compute a result, the addition function adds two to that result, and the final value (8) is returned.

Functions provided by APL 1100 include a wide range of processes from simple addition to matrix division. In addition, user-defined functions may be formed to evaluate processes not included in the standard set. User-defined functions are named at definition time, and they follow the same syntax rules as the standard functions.

2.4.4.1. Access to APL 1100

The SPERRY UNIVAC APL 1100 processor runs with CTS under control of the SPERRY UNIVAC Series 1100 Operating System. Once access to CTS has been gained, the user need only enter:

*-> *APL[,O] [/W] .*

where *O* is a list of processor options at log-on time, and */W* is the alphanumeric key, which the user supplied to APL 1100 at sign off, locking his library file from undesired access by other users.

Examples:

*-> *APL*

*-> *APL /LIBKEY*

2.4.4.2. Processor Options

Options specified at log-on time can be used to assist the user in program checkout, provide additional security or information, or direct the execution of APL 1100 programs. The valid options are given in Table 2-1.

Table 2-1. APL 1100 Opticns

Option Letter	Purpose
B	User's terminal type is a bit-paired device.
T	User's terminal type is a typewriter-paired device.
N	Inhibits printing of images obtained via an @ADD statement.
V	Causes printing of each image read by APL 1100.
C	Assume)CSITE at log-on time (onsite print file)
S	Assume)SITE at log-on time (onsite printer).

2.4.4.3. Statements

APL 1100 statements describe the processes required to evaluate an algorithm. Since many processes are parallel rather than serial in nature, APL 1100 extends its functions to process arrays or vectors in the same manner as scalar data. There is a set of over sixty functions available to the APL 1100 user. Evaluation of each statement proceeds from right to left, with all operators having equal precedence. Thus the result of:

```
2 + 3 x 4
```

is the numeric value 14. Statements may be of any length, up to the width of an input line. The order of evaluation may be modified by inserting parentheses to indicate groupings, e.g.,

```
(2 + 3) x 4
```

yields a result of 20. Several statements may be entered on one line if they are separated by semicolons:

```
2 + 3 , 4 x 6 ; 11
```

Since each statement above produces a value, the result would be a set (vector) of three values: 5, 24, and 11.

Statements may be entered whenever input is solicited. APL 1100 input is requested by spacing to the right and then pausing for the input image. Several types of input may be requested, depending on the type of results desired. Calculator input, the type shown above, is the usual mode for APL 1100. Other input modes are called "quad," "quote-quad," "prompt," or function definition. Each mode is easily recognized by the information printed in spacing to the right prior to input acceptance.

Calculator mode indents eight spaces when requesting input. The indentation process leaves blank spaces to the left of the input image. In this mode, the statement is evaluated immediately, and the results can be printed below the input line starting in column 1:

```
        >2 + 3 x 4.0279365
14.0838095
```

If the result is a character vector or array, it is printed with no spaces between components in each row. Other arrays or vectors are printed with at least one space between components.

Quad input requests information to be evaluated, and may be recognized by the quad character in column 1 of the solicitation:

```
⎕:          2 x 3

⎕:
```

Quote-quad input obtains the actual incoming character sequence, and may be recognized by the overstruck quad and quote characters in column 1:

```
⍞           This is input.
```

Prompt input can be used to ask explicit questions. It may be recognized easily, since the input is solicited immediately to the right of the question:

```
PLEASE TYPE IN YOUR NAME >MAXINE
```

Function definition mode will request images by line number. A typical solicitation character would contain the number enclosed in brackets:

```
[13]        2 + 3 x 4
```

Once the type of input desired is ascertained, enter the statement or expression appropriately.

APL expressions follow a very simple syntax, and can be used to indicate very powerful actions. Expressions consist of values and functions. Spacing is not required between the primitive function operator and its operand, but may be inserted if desired. The values may be constants, as shown previously, or named variables. Functions may require zero, one, or two arguments (values).

A function which requires no arguments is called a niladic function, and may be used in place of a value as an argument to other functions. The quad function is niladic, so the following statement is valid:

```
!⎕
```

This statement will request input, then apply the evaluated result to the factorial function as a right argument.

Functions which have a single argument are called monadic functions. The argument is always a value located to the right of the function name. The following statement evaluates the factorial of the factorial value of 3 and returns a final result of 720:

```
!+ !3
```

The function, !3 is evaluated as 3x2x1, and returns a result of 6, which is the argument of the leftmost function. Thus, the final result 720, comes from evaluating 6x5x4x3x2x1.

2.4.5. Other Processors

The advantages of CTS extend to any processor in the operating system. CTS is designed to make it easy to create data sets of all kinds. The data set, which conforms to the system and semantics associated with a processor, is a program written in the language of that processor. To CTS, however, it is a data set. The full power of CTS is available to create, edit, test, save, retrieve, and use programs.

CTS is as useful to the experienced programmer, who can write a very tight program in assembly language, as to the occasional programmer using BASIC. Counterparts for most of the operating system control statements are available directly in CTS. Most of the remaining ones may be created through the CSF statements. Through the use of the PXQT command (see 6.5) many system processors can be accessed. For those which cannot, CTS can be left temporarily with the XCTS statement and a statement or two can be submitted directly to the Executive. Then CTS can be reentered via the @CTS statement.

CTS may be used to create partial or complete run streams, which may be added or started from within CTS via the ADD statement and the CSF statement. The COMPILE, RUN, MAP, and XQT statements in their full format handle quite complex programs entirely within CTS, using the economy of expression which CTS affords. For example, several elements from possibly different files can be compiled with different compilers, mapped with chosen libraries (with MAP directives if desired), and the resulting program executed—all with a single RUN command. This is a very substantial economy of expression, compared with either batch or ordinary demand mode operation.

Both the novice and expert benefit from the use of the editing commands. Those editing commands which locate strings of text with desired properties (see 5.1) can be used to scan the output from a processor execution as well as for finding errors directly in the source code of a program. Those editing commands which modify f (see 5.2) can be used to make corrections to the source code of the program.

Many of the features provided by a prescan module can be implemented directly with CTS statements. The TAB command (see 2.3.4) permits special formats. The CHANGE command makes it possible to expand abbreviations easily. The assumed compiler may be changed with the ASSUME COMPILER command (see 6.2.1).

A knowledge of the language of the processor to be used is necessary. As the operations performed increase in complexity, more knowledge is needed, not only about the language, but also about the operation of the processor and the operating system.

2.4.6. Controlling Prescan Local Syntax Checking – SYNTAX

Syntax: SYNTAX [k]

Abbreviation:
SYNTAX – SYN
ON – None
OFF – OF
TYPE – T

Function: To inhibit or reestablish local (line-by-line) syntax checking in a prescan module, or to display the current state of syntax checking in a prescan module.

The SYNTAX command is effective only when a prescan module is in control. Entering a prescan module always turns on local syntax checking. Consequently, any attempt to use the SYNTAX command to change the state of local syntax checking, when not under control of a prescan module, would be ineffectual.

When local syntax checking is turned on, the prescan module performs a line-by-line syntax check for errors which do not depend on relationships between statements. When the prescan module detects an error, it rejects the line and displays a diagnostic message, requesting a correction of the difficulty at once.

In some cases this syntax checking may not be wanted. Perhaps an entire paper tape is to be read in or an ADD command is to enter lines from an element or file. In either case, there is no chance to correct lines on a one-by-one basis. The SYNTAX command can turn off the local syntax checking to accommodate such situations.

BFOR, the Fieldata FORTRAN prescan module (see 2.4.2), has a pair of private commands which do the same thing. The NODIAG and DIAG commands perform the same function, but are implemented within the module. Either the NODIAG or SYNTAX OFF commands will inhibit local syntax checking in BFOR. Both the DIAG and SYNTAX ON states must be established simultaneously to permit local syntax checking in this module.

The parameter k may be coded OFF, ON, or TYPE (or their abbreviations). If k is omitted, TYPE is assumed. When coded OFF, local syntax checking is inhibited. When coded ON, the inhibition is removed. When coded TYPE, the prescan module and the state of syntax checking is displayed. The following example illustrates the above points:

```
->SYNTAX OFF
->SYN T
PRESCAN MODE=NONE
->FOR
ASCII FORTRAN PRESCAN 2R1A
>>SYN T
PRESCAN MODE=TN:ON
->SYN OFF
>>SYN
PRESCAN MODE=TN:OFF
>>CLEAR
->FOR
ASCII FORTRAN PRESCAN 2R1A
>>SYN
PRESCAN MODE=TN:ON
>>
```

2.4.7. Terminating Prescan Control – CLEAR

Syntax: CLEAR

Abbreviation: CLE

Function: To terminate control of a prescan module.

The CLEAR command terminates the control of the currently active prescan module (if any) without affecting the contents of the working area f or the assumed compiler. The CLEAR command also terminates DATA mode (see 3.7) and returns control to ELEMENT mode. It also does not affect the operation of a SCAN command which is in effect. To terminate the SCAN mode, an EDIT command must be used.

3. Saving and Retrieving Programs

3.1. Specifying a Program Element or Data File

Data images entered into CTS go into the temporary working area file, f. Because f is not preserved when the run is terminated, information in f is lost between sessions. To save partially completed work or completed programs, the contents of f can be moved into F, a cataloged file. In this way the information is preserved between sessions.

For the commands in this section which reference F, any program file can be referenced if the substitute file is explicitly named in the d field of the command. Such a substitution persists only for the one command, in contrast to the substitutions made by the ASSUME PROGRAM or ASSUME FILE commands which are valid until another ASSUME command changes them.

Normally the d field in these commands refers to an element in F or its substitute. If only a file name rather than an element name is specified, CTS will seek a data file, not a program file. In this case, the entire contents of the data file are used in the operation of the command. Usually, CTS operates in ELEMENT mode. This means that, unless specifically designated as a file (by containing an asterisk "*" or being terminated by a period), an identifier is assumed to be an element name. The DATA command (see 3.7) reverses this assumption. It establishes CTS in the DATA mode, in which elements are not permitted at all, and unspecified identifiers are assumed to be file names of data files. OLD, MERGE, SAVE, and REPLACE are the only commands affected by DATA mode.

3.2. Making a Permanent Copy – SAVE

Syntax: SAVE [d [L]]

Abbreviation: SAV

Function: To store all or part of the contents of f into a mass storage file.

The following example:

```
130 >*SAVE
->
```

shows how SAVE is usually used. The SAVE command copies all of f into F as an element with the name which has been specified for the working area. The contents of the file may be used in another terminal session. Since f is not saved by the system after logging off and F is saved, SAVE keeps the contents of f from being lost. To continue working in another session simply retrieve the

information from F back into f (see 3.5).

The COMPILER, FILE, OBJECT, PROGRAM, and SAVELENGTH options of the ASSUME command affect the operation of the SAVE command. (See 6.2 and 3.2.3.)

SAVE is one of the commands which CTS interprets differently depending on whether CTS is operating in ELEMENT or DATA mode (see 3.7). ELEMENT mode is more commonly used and is the default mode. DATA mode is entered with the DATA command. ELEMENT mode deals with elements in a program file, and DATA mode deals with an System Data Format (SDF) file (see Section 7). Consequently, when in the ELEMENT mode, SAVE causes CTS to create a symbolic element, usually in F, the contents of which are a copy of all or a part of f. In DATA mode, SAVE causes CTS to create a data file and write into it a copy of all or a part of f. The CTS responses to various situations involving SAVE differ, depending on which mode is in effect.

3.2.1. Saving f as a Program

The first parameter, d, may specify either a file name or an element name. If it specifies an element name, a file name may also be included. In this case, the file name specifies the program file into which the element is to be inserted. If the parameter specifies an element, CTS creates such an element, unless it already exists (see the following examples).

If d specifies a file name (and not an element name), CTS creates a file with the specified name and writes the contents of f (or the specified portion) into it in SDF (see Section 7). CTS recognizes a name as a file name (and not an element name) by the presence and location of an asterisk (*) or a period. Some examples of d parameter specifications and their interpretation follow:

d	Interpretation
ABC	Element ABC in F (ELEMENT mode) or file ABC (DATA mode).
FA.ABC	Element ABC in program file FA
ZZZ*FA.ABC	Element ABC in program file FA which has a qualifier of ZZZ.
D.	Data file D.
*D	Data file D.
ZZZ*D	Data file D which has a qualifier ZZZ.
.ABC	Element ABC in F (ELEMENT mode) or not an acceptable format (DATA mode).

If the parameter d is omitted, the name of f is substituted for it, unless f is unnamed, in which case CTS solicits the parameter. The name of f will have been specified by the NEW (see 5.3.1), OLD (see 3.5), or RENAME (see 5.3.5) commands. It may have any of the forms illustrated. It is possible, for example, for f to have ZA. as its name, and a SAVE command with no parameters would then create the data file ZA and write the contents of f into it.

The second parameter, L, specifies which part of f is to be saved by specifying the range of line numbers to be included. Any of the line number specification formats given in 2.2.4.2 may be used. If a range is specified, the endpoints serve only to define the range and need not be existing line numbers. If L is omitted, A is the default, resulting in all of f being saved. L may be specified only if d is present. If L is specified without d, CTS will try to interpret the L specification as d. Depending on the nature of the specification, various situations arise, most of which result in a diagnostic message.

When CTS stores the symbolic element it associates the current assumed compiler with the saved element. If no assumed compiler has been specified, CTS uses ELT as the type of the symbolic element.

The assumed compilers do not always produce unique symbolic element types. This is discussed further under the OLD command (see 3.5) which converts the symbolic element type into an assumed compiler when retrieving a symbolic element back into f.

The following listing gives the symbolic element types which CTS uses for various assumed compilers:

Assumed Compiler	Symbolic Element Type Produced
ACOB	COB
ALGOL	ALG
APL	APL
ASM	ASM
BASIC	BAS
COB	COB
DOC	DOC
ELT	ELT
FOR	FOR
FTN	FOR
MDC	MAC
MAP	MAP
MASM	MSM
NUALGOL	ALG
PLUS	PLS
PL1	PL1
RFOR	FOR
SECURE	SEC
SSG	SSG

The SAVE command changes neither the contents of f nor its name.

The response of CTS to the SAVE command depends on what conditions it finds. Examples of responses to SAVE commands follow.

- The normal response, when all parameters are correctly given or implied, is simply the solicitation character.

  For example:

  ```
  ->SAVE ABC
  ->
  ```

  The absence of any diagnostic message means that a symbolic element named ABC has been created in F, the contents of which is a copy of f.

  For a more elaborate example:

  ```
  ->100    LINE 1
  ->200    LINE 2
  ->300    LINE 3
  ->SAV       ALT.ABC      150,250
  ->
  ```

  In this case, the program file, ALT, already exists. An element named ABC is created in ALT. The range specified by L includes only the second line (line 200), so this line is the entire contents of the new element.

- If d is not given, then the name of f is used as the name of the element to be created. If, in addition, f is not named, CTS solicits a name:

  ```
  ->SAVE
  NEW PROGRAM NAME?  >
  ```

  Enter the name:

  ```
  ->SAVE
  NEW PROGRAM NAME?  >DEF
  ->
  ```

  A symbolic element DEF containing a copy of f has been created in F.

  It is also permissible to specify the L parameter, in addition to the, program name in response to the solicitation:

  ```
  ->SAV
  NEW PROGRAM NAME?  >FA.Z      110, 140
  ->
  ```

  Element Z in program file FA has been created containing those lines of f, the line numbers of which are between 110 and 140 inclusive.

- If d specifies a file rather than an element, the command creates a data file by that name and saves f in it as described in the next section.

- If a SAVE command is attempted when f is empty, no element is created and a diagnostic is given.

  ```
  ->SAVE GHI
  THE WORK AREA IS EMPTY
  ->
  ```

  Now another command or line of data may be entered.

- If the name of a symbolic element which is already in F is specified, the SAVE operation does not take place, the old element is not deleted or changed, and CTS responds with a diagnostic:

```
->SAVE ABC
<5> DUPLICATE NAME ABC - PROGRAM NOT SAVED
->
```

CTS is now ready for a new command.

This restriction applies even if the symbolic element to be created is of a different compiler type than the existing one of the same name. Only one symbolic element of a given name is permitted in a program file, so the current one would have to be discarded to permit inclusion of the new one. Since CTS is not certain this is desirable, it aborts the operation rather than destroy a potentially useful element. To destroy the contents of an element or data file and replace it with the contents of f, use the REPLACE command (see 3.3).

- If a string of characters which is not a legal file or element name is specified for d, any number of diagnostic messages could occur, depending on the nature of the string. The offending characters are usually repeated in the message to help locate what is wrong. For example, if while saving an element called FILE, the string "FI;E" was entered, the following sequence would occur:

```
->SAV FI;E
<23> ILLEGAL FILE OR PROGRAM NAME SYNTAX I;E
->
```

The offending character is the semicolon (;) and attention is directed to it by the indicated string, "I ; E" at the end of the message.

- If f is not empty, but a specification of L is given such that none of the actual line numbers of the data in f are included, no element is created and one of three diagnostic messages will be given, depending on whether the line number range specified by L is before the first line number of f, after the last line number, or between two line numbers.

For example:

```
->100 LINE 1
->150 LINE 2
->SAV A 10,50
TOP OF FILE
->SAV B 200,250
END OF FILE
->SAV C 110,120
<110> SPECIFIED LINES DO NOT EXIST
->
```

No elements have been created. CTS is ready for another command or line of data. For the three cases illustrated the line pointer, p, (see 2.2.3) is 0, 0, and unchanged, respectively.

- If CTS is under control of the prescan module of a compiler, as when the FOR, COBOL, or FTN command is used (see 2.4), data lines are scanned a line at a time for syntax errors as they are submitted. No attempt is made to detect global errors (errors which depend on the relationship between more than one statement) until the program is either compiled by the COMPILE (see 6.4.1) or RUN (see 6.2) commands, or is referenced by certain commands which treat the entire program as a unit. SAVE is such a command. Before performing a global scan the prescan module asks whether it is wanted. A *Y* response causes a global scan and any errors are noted.

  An *N* response stops the scan. The detection of errors by the global scan does not terminate the SAVE.

  The following example illustrates this:

```
->FOR F
FD FORTRAN 5R1
>>NUMBER
100 >X-10
<513> REJECTED:  STATEMENT UNKNOWN OR MISSPELLED.
>>C /-/=/
100       X=10
110 >Y=25
120 >Z=X+Y
130 >GO TO 1
140 >*LIS
100       X=10
110       Y=25
120       Z=X+Y
130       GO TO 1
END OF FILE
>>SAV A
DO YOU WANT A GLOBAL SCAN?  > Y
<615> ALLOWED (130):  MISSING END CARD.
<608> REFUSED (130):  LABEL '1' IS NOT DEFINED.
>>
```

  See 2.3.2 and 2.4.2 for an explanation of the FOR and NUMBER statements used here. Notice that the error on typing line number 100 was detected when it was submitted because it was inherent in the statement itself. Two additional errors, however, were not detected until the SAVE command prompted the prescan FORTRAN syntax checker to check for global errors. The SAVE was performed.

- If d specifies only a file, rather than an element, and the file is a program file rather than a data file, CTS responds with a message:

```
->SAV DA.
<18> DA IS NOT A DATA FILE
->
```

  The SAVE has not been performed.

- If d contains both a file specification and an element specification, and the file is a data file, rather than a program file, CTS responds with the following message:

```
->SAV FILB.A
<19> FILB IS NOT A PROGRAM FILE
->
```

The SAVE has not been performed, and CTS is ready for the next command or line of data.

- If d contains both a file specification and an element specification, and the file does not exist, CTS prints the following error:

```
->SAV  FILA.A
<68> FILA IS NOT CATALOGUED
->
```

Now another CTS command or line of data can be entered.

3.2.2. Saving f as a Data File

If d specifies only a file name, rather than an element or a file and an element, the SAVE command will save f (or the specified portion) as the contents of a System Data Format (SDF) file (see 7.1.1). CTS recognizes d as being a file name rather than an element name by the presence of an asterisk or a trailing period. CTS will also treat it as a data file if in DATA mode as described in the last section.

The second parameter, L, specifies which part of f is to be saved by specifying the range of line numbers to be included. Any of the line number specification formats given in 2.2.4.2 may be used. If L is omitted, A is the default, and all of f will be saved. L may be specified only if d is present. The line numbers which are the endpoints of the range specified by L do not necessarily have to exist in f, but the range should include at least one actual line.

The SAVE command changes neither the contents of f nor its name, but does change the line pointer, p, to zero.

The response of CTS depends on what it finds as it checks for various conditions. The following examples show responses to various SAVE commands which specify a data file name.

- The normal response, when all parameters are correctly given or implied, is a solicitation of information to permit creating the file, followed by a message indicating that the file is being created, and followed in turn by the solicitation character. The file information is solicited as if a CREATE command (see 7.5.1) had been submitted with the name of the file as the only parameter.

  For example:

```
->SAVE  ABC.
IS THIS FILE TEMP, PUBLIC, OR PRIVATE? >T
DEVICE CHARACTERISTICS: >F2
MAXIMUM SIZE?  >                    (no answer, CR)
*CRE,T  ABC., F2
->
```

  The responses to the information solicited on the second, third, and fourth lines, and the message on the fifth line are described in 7.5.1. The fifth line displayed by CTS indicates that a file has been created with the characteristics specified. The solicitation character means that all parameters were interpreted, and that the contents of f were written to the created file.

- If d is not specified, the name of f will be used. If, in addition, f is not named, the name will be solicited. Then the normal sequence will occur.

For example:

```
->SAV
NEW PROGRAM NAME?  >DEF.
IS THIS FILE TEMP, PUBLIC, OR PRIVATE?  >T
DEVICE CHARACTERISTICS:  >                    (no answer, CR)
*CRE,T  DEF.
->
```

The file has been created, the save performed, and another command or data line may be entered.

- If the parameter d contains the name of an existing empty file, the file creation sequence is skipped, as in the following:

```
->SAV  ABC.
->
```

The contents of f were written to file ABC in SDF (see 7.1.1).

- If a file with the name specified already exists, and is not empty, CTS aborts the operation with an explanatory message.

For example:

```
->SAV  ABC.
<96> DUPLICATE DATA FILE NAME - NOT SAVED
->
```

- It is possible, during the sequence of solicitation messages for creating the file, to cause an error which will abort the creation of the file. This causes the creation sequence to start over, as illustrated by the following example:

```
->SAVE  B.
IS THIS FILE TEMP, PUBLIC, OR PRIVATE?  >PUBLIC
READ AND WRITE KEYS:  >R/W
DEVICE CHARACTERISTICS:  >DEVICE
MAXIMUM SIZE?  >
*CRE,PU  B/R/W,DEVICE
<81> FORMAT OR OPTION ERROR IN CONTROL STATEMENT
->SAVE B.
IS THIS FILE TEMP, PUBLIC, OR PRIVATE? >PUB
READ AND WRITE KEYS:  >R/W
DEVICE CHARACTERISTICS: >                     (no answer, CR)
*CRE,PU  B/R/W.
->
```

The identifier, DEVICE, is not a legal device type. The attempt to use it as one created an error condition and led to the diagnostic message. The file was not created the first time so the creation sequence started over. This time the file was created and the SAVE was done.

3.2.3. Setting the Maximum Length of a Saved Line – ASSUME SAVELENGTH

Syntax: ASSUME SAVELENGTH [i]

Abbreviation: A SAV

Function: To establish the maximum length of a saved line.

The ASSUME SAVELENGTH command specifies the maximum length, i, of an image when it is saved via the SAVE (see 3.2) or REPLACE (see 3.3) commands. This length, normally 132 characters, should be shortened to 80 for compiled programs and files intended to be directly read by the Operating System. If an image exceeds this length, it is reduced by truncation. For efficiency, CTS saves images at even word intervals. If i is not an increment of six (for Fieldata characters) or four (for ASCII characters), it is rounded up to an increment of six or four respectively by CTS. If a line which exceeds the SAVELENGTH is encountered during a SAVE or REPLACE operation, the line is truncated and the operation continues after printing the following message:

```
<53> LINE (line number) IS TOO LONG
```

3.3. Updating a Copy – REPLACE

Syntax: REPLACE [d [L]]

Abbreviation: REP

Function: To discard the contents of a data file or a symbolic element of a program file, and to substitute in its place all or a designated part of f.

The REPLACE command is usually used to replace an element of F with an updated version. Typically, at the start of a session an element will be moved from F to f with the OLD command (see 3.5) and modified. Then the old version is replaced with the new, modified program.

The same end could be accomplished by doing an UNSAVE (see 3.4) followed by a SAVE (see 3.2). However, the REPLACE is easier and safer. CTS checks all parameters and conditions before performing the REPLACE, and performs no part of it unless everything is in order. Using the UNSAVE followed by a SAVE could delete the element while nothing was in f. This would result in the loss of the element. If the REPLACE is used, the old element is not deleted, and a warning is printed.

Another difference is that an UNSAVE deletes all elements (symbolic, relocatable, and absolute) with the given name, while a REPLACE deletes only a symbolic element.

The first parameter, d, may specify either an element or a file. If, as is usually the case, d specifies an element, the element is to be replaced by all or the specified part of f. The element specification may include a file name as well as the element name (see 3.2.1), in which case the file name specifies the program file in which the element resides. In the absence of a file name, F is taken as the file.

When a file name is specified, the file must exist. If the file is cataloged, but is not assigned to the run, CTS will assign it before proceeding with the REPLACE command.

The parameter L denotes the range of line numbers in f which are to be included in the new element. Any of the formats specified in 2.2.4.2 may be used. The endpoints of this range need not be existing line numbers of f, but the range must include at least one existing line. If the L parameter is missing, A is assumed, and the entire contents of f will be included in the new element.

REPLACE never alters the contents of f or its name, but will usually set the line pointer, p, to zero.

Most situations which arise are treated exactly as they are in the SAVE or UNSAVE commands, giving rise to exactly the same messages and solicitations for information.

Situations for which responses are the same as they are in the SAVE command (see 3.2) are:

- The normal case when all parameters are correctly specified or implied.
- The parameter d is not given and f is named.
- The parameter d is not given and f is not named.
- The file f is empty.
- The syntax of d is bad.
- The range of L is such that no actual lines of f are included.
- A prescan module is in control and a global error exists.
- The parameter d contains a file specification and the file does not exist.
- The parameter d contains both a file and an element name and the file is a data file.
- The parameter d contains an explicit file name and no element name.

In all of these cases, when no SAVE is performed, the original element is retained.

The following situation is treated exactly as in the UNSAVE command (see 3.4) with the exception that REPLACE considers only symbolic elements while UNSAVE considers elements of any type.

- The element described by d does not exist.

The following is an example of the use of REPLACE:

```
->FOR F
FD FORTRAN 5R1
>>NEW A
>>N
100 >2 FORMAT ()
110 >READ (5,1) A,B
120 > C = SQRT(A**2 + B**2)
130 >WRITE(6,1) A,B,C
140 >END
150 >*SAV
DO YOU WANT A GLOBAL SCAN? >Y
<606> ALLOWED (110):   '1' IS REFERENCED BUT NEVER ASSIGNED A VALUE
>>CHANGE /2/1/100
100 1      FORMAT ()
110 >*REP
DO YOU WANT A GLOBAL SCAN?  >Y
>>
```

In this example, a FORTRAN program is created which has a global error (i.e., an error which depends on the relationship between more than one statement) in the FORMAT statement. When the SAVE command was issued, the FORTRAN prescan module discovered the discrepancy and produced a diagnostic. The SAVE was performed and element A in F was created containing a copy of the contents of f. The CHANGE command corrected the error. The REPLACE command then updated element A in F with the new contents of f. If there had still been a global error, the FORTRAN prescan module would have produced a diagnostic following the REPLACE.

- If the file specified by d does not exist, CTS responds as follows:

```
->REP A.
<68> A IS NOT CATALOGUED
<73> PROGRAM NOT REPLACED
->
```

  The REPLACE operation was aborted. CTS is now ready to accept another command or line of data.

- If the file specified by d is a program file, CTS responds as follows:

```
->REP B.
<18> B IS NOT A DATA FILE
->
```

  The file B has not been changed. CTS is ready for the next command or line of data.

3.4. Discarding a Copy – UNSAVE

Syntax: UNSAVE [d]

Abbreviation: U

Function: To delete an element from a program file or to delete a data file.

The UNSAVE command is usually used to delete a symbolic element from F which has been previously saved with the SAVE command.

The UNSAVE command never affects the contents of f.

The UNSAVE command usually directs CTS to look for an element of a program file (see 7.1.1), F, unless specified otherwise, and delete the element. While UNSAVE is normally used to delete a symbolic element previously saved in F with the SAVE command (see 3.2), it will cause an element of any type to be deleted if it has the name specified by d. If the file has more than one element with the same name (possible only if they are different types) all of them will be deleted. It also deletes from the assume object file all elements with the same name.

The parameter, d, may be any legal element or file name.

NOTE:

In the absence of a file specification with an element name, both F and the assumed object file will be searched, and any element with the indicated name in either of the files will be deleted.

If d is an explicit file name, this command will check to see that the file is not a program file and delete it. If the file is cataloged but not assigned, CTS will assign it before the check.

The response of CTS to the UNSAVE command depends on the situation at the time it is used. The examples which follow illustrate responses to various situations.

- The normal response is the solicitation character, which indicates that the element has been located and deleted from the file:

```
->UNSAVE  A
->
```

  All elements with the name A in the file F and the assumed object file have been deleted.

- If the d parameter is omitted, CTS solicits the name:

```
->UNSAVE
PROGRAM NAME?  >
```

  Now enter the name of the element to be deleted:

```
->UNSAVE
PROGRAM NAME?  >B
->
```

  All elements with the name B in F and the assumed object have been deleted.

- If an element which is not in the file is specified, CTS explains this with a message and aborts the UNSAVE operation:

```
->U  B
<4> ELEMENT  .B  CANNOT BE FOUND
->
```

- If d has a syntax error, CTS will print an appropriate message indicating the offending string:

```
->U  #A
<23> ILLEGAL FILE OR PROGRAM NAME SYNTAX #A
```

  The offending string is usually printed to help locate the source of the difficulty.

- If a data file name is specified and it cannot be found, CTS responds as follows:

```
->U  A.
<68> A IS NOT CATALOGUED
->
```

  The UNSAVE operation has been aborted and CTS is ready for the next command or line of data.

- If the file specified is a program file, it will not be deleted. A diagnostic message will result as in the following illustration:

```
->U  PA.
<18> PA IS NOT A DATA FILE
->
```

3.5. Retrieving a Copy – OLD

Syntax: OLD [i] [,j] d [L]

Abbreviation: None

Function: To discard the contents of f and replace them with all or part of a symbolic element of a program file or with all or part of a data file, changing the name of f, the assumed compiler and the character set to correspond to the new contents.

The OLD command is usually used to retrieve from F an element previously saved with the SAVE command (see 3.2). Some refinements, however, are useful for special tasks.

The OLD command makes five kinds of changes to f. First, it changes the contents. Second, it changes the name. Third, if the element or data file is the opposite character set mode (ASCII or Fieldata) of the working area, the mode of the working area is changed (unless a prescanner is active). (See 2.2.4.3). Fourth, when an element is specified it changes the assumed compiler (see 6.2.1) if it is different than the current one. It may happen that aspects of f will be the same after the OLD command as before, but the OLD command has in this case discarded the old value and established an identical new one. Before making these changes the syntax of the OLD command is completely checked. If an error is found, no changes are made and a diagnostic is given.

CTS interprets the OLD command as a request to discard the contents and name of f, the character set, and the assumed compiler currently in effect if it is different than the current one. These are replaced with values connected with the designated element or file. The contents of f are replaced by all or the indicated part of the new element or file, the name of f by the name of the new element or file (by d), the assumed character set, and, if necessary, the assumed compiler by the compiler consistent with the symbolic type of the new element. An OLD of a data file does not change the assumed compiler.

If i is not specified, the lines from d retain their original line numbers. The i parameter specifies that the lines of d are to be resequenced, starting with line number i with increments of j. The j parameter may be used without i. In this case j is added to each original line number from d as it is inserted in the working area. If i is used and j is omitted, an increment of 10 is assumed.

Table 3–1 gives the common assumed compiler and options which the OLD command will put into effect for various element types.

In discussing the SAVE command (see 3.2) it was noted that the type of the assumed compiler at the time an element was saved determined the symbolic type of the element. However, several assumed compilers created symbolic elements of the same type. It is, therefore, not always possible for the OLD command to regenerate the same assumed compiler that was in effect at the time an element was saved. ELT, ALG, COB, and FOR elements could each have had more than one type of assumed compiler. Before compiling such an element, a change to the assumed compiler (see 6.2.1) may be needed to avoid problems.

Table 3–1. Assumed Compiler and Options for OLD Command

Type of Symbolic Element	Resulting Assumed Compiler	ASCII or Fieldata Element
ALG	NUALG,S	either
APL	APL	either
ASM	ASM,S	either
BAS	BASIC,R	Fieldata
COB	ACOB,ES	ASCII
COB	COB,SBE	Fieldata
DOC	DOC,LDR	either
ELT	ELT,L	either
FOR	FTN,C	ASCII
FOR	RFOR,RS	Fieldata
MAC	MDC,S	either
MAP	MAP,XS	either
MSM	MASM,S	either
PL1	PL1,S	either
SSG	SSG	either
SYM	ELT,L	either

The parameter d may specify any legal element name (see 3.2). It may, therefore, include a file name as part of this specification. If the file name is missing, F is assumed. If the file is cataloged but not assigned, CTS assigns it to the run.

The parameter d may also explicitly specify a file name with no accompanying element name. CTS recognizes d as being a a file name rather than an element name by the presence of an asterisk or a trailing period. When in DATA mode it only allows a file name.

The parameter L may be used to specify the range of line numbers of the data file or element being retrieved which is to be included in the final contents of f. The only forms of L allowed are:

n1	OLD this line only.
n1,n2	Start at line n1 and OLD through and including line n2. Line n2 cannot be less than n1.
n1+i	Start at line n1 and OLD the next i lines.
n1+	Start at line n1 and OLD the rest of the data file or element.

For example, an L of:

```
110,150
```

is allowed, while an L of:

```
150,110
```

is not.

The range specified by L should include at least one statement. If L is not specified, A is used.

The responses of CTS to various situations involving the OLD command are illustrated by the following examples:

- The normal response, when all parameters are correctly specified or implied, takes two forms. If the assumed compiler in effect is the same as the one generated from the symbolic type of the element, the following sequence occurs:

```
->OLD A
->
```

  Element A from F has replaced the contents of f so the name of f is now A, and the assumed compiler and the options are the same. If the new assumed compiler is different, a sequence like the following may occur:

```
->OLD  FILA.B  120,220
COMPILER TYPE RFOR,RS
->
```

  Lines with numbers from 120 to 220 inclusive of the element B from file FILA have replaced the contents of f. The name of f is now FILA.B. The assumed compiler is RFOR,RS.

- If d is not specified, CTS solicits it:

```
->OLD
OLD PROGRAM NAME? >
```

  Now enter the element name. The L specification may also be appended:

```
->OLD
OLD PROGRAM NAME? >B
->
```

  Element B from F has replaced the contents of f, and B is the new name of f. The assumed compiler has not changed.

- If d is specified but the element named does not exist, CTS responds with:

```
->OLD FILA.B
<4> ELEMENT FILA.B CANNOT BE FOUND
->
```

  No changes to f have been made.

- If the d specified has syntax errors, the sequence produced by CTS is:

```
->OLD PA+B
<23> ILLEGAL FILE OR PROGRAM NAME SYNTAX A+B
->
```

  The offending string is indicated to help locate the trouble. No changes to f have been made.

- If the file portion of the element specified is a data file rather than a program file (see 7.1.1), CTS responds with a diagnostic message:

```
->OLD DB.A
<19> DB IS NOT A PROGRAM FILE
->
```

  The contents of f are unchanged.

- If the file portion of the element specified does not correspond to an existing file, CTS responds with:

```
->OLD DD.A
<68> DD IS NOT CATALOGUED
->
```

  The contents of f are unchanged.

- If the mode of the working area is different from the mode of the element or data file, the mode of the working area will be changed (unless a prescanner is active). For example:

```
->OLD A
ASSUME ASCII ON
->
```

  The mode of f was Fieldata (ASCII OFF) before the OLD of A was done. Since A is an ASCII element, the OLD caused the mode to be changed to ASCII.

- If an element or data file contains both ASCII and Fieldata images, the mode of the working area will be ASCII.

- If a prescan module is in control when the OLD command is given, there is a modification of the rule for changing the assumed compiler. A prescan module can only operate on the contents of f if it contains a program written in a specific language. Therefore, it maintains an assumed compiler compatible with that language. If the OLD command attempts to assign a different assumed compiler, it may be that the symbolic element requested by the OLD compiler contains a program written in another language. CTS warns of this situation with the message:

```
->FOR F
FD FORTRAN 5R1
>>OLD B
COMPILER TYPE:  ELT,L PROCEED?  >
```

  Although the symbolic element type is ELT, the element may contain a FORTRAN program which is to be loaded into f. If the solicitation message is answered with a blank (or empty) response or any response the first letter of which is Y, CTS will load the contents of the element into f and change its name to the name of the element (to d actually), but it will not change the assumed compiler.

Any other response will result in the contents of f being unchanged. In either case the solicitation character is then given and CTS is ready for the next instruction or line of data.

- If a d is specified which is the name of a program file, CTS responds with:

```
->OLD PA.
<18> PA IS NOT A DATA FILE
->
```

Once again, f is unchanged.

- When a prescanner is active, the mode (ASCII or Fieldata) of the working area cannot be changed. If an element or data file of the opposite mode is specified, it will be changed to the mode of the working area. For example:

```
->ASSUME ASCII ON
->FOR
ASCII FORTRAN PRESCAN 2R1A
->OLD DF.
<133> FIELDATA IMAGES WERE TRANSLATED
->
```

3.6. Combining a Copy with f – MERGE

Syntax: MERGE [i] [,j] d [L]

Abbreviation: MER

Function: To merge with f all or part of a symbolic element from a program file or all or part of a data file d.

The MERGE command may be used to append a file or element to the end of f, insert a file or element between two lines of f, or interleave the lines of f with the lines of the file or element. The RESEQUENCE command (see 5.3.2) is useful in conjunction with MERGE. After using the MERGE, the line numbers of f will often lack the uniformity that makes editing easier. The RESEQUENCE command restores a uniform line numbering to f.

Where these lines will be placed when merged also depends on the ASSUME RESEQUENCE ON/OFF mode (see 5.3.6). ASSUME RESEQUENCE ON merges them as a sequential block of lines. If a line number conflict occurs, all following lines are moved down. ASSUME RESEQUENCE OFF inserts the lines without disturbing the other working area lines unless a duplicate line number occurs. If this happens, the merged line replaces the working area line. These rules apply whether or not the MERGE generates new line numbers.

The MERGE may be thought of as acting in three steps: selecting an element of a program file or a data file, extracting from the contents the lines desired, and inserting these lines into f, while resolving line number conflicts which may occur.

The selection and extraction steps work exactly the same for the MERGE command as they do for the OLD command (see 3.5). Only the d and L parameters are used in these steps, so ignoring the parameters i and j for the moment, the responses as seen at the terminal for various situations are for the most part identical. Consequently, refer to 3.5 for a description of these responses, noting first the minor differences pointed out in the following.

The MERGE command never causes CTS to consider the symbolic type of an element. Consequently, the messages connected with the assumed compiler type are never displayed. These are messages such as:

COMPILER TYPE ELT,L PROCEED? >

The restrictions on parameters d and L given in 3.5 also apply for the MERGE command. If a prescan module is in control, the lines inserted may cause a diagnostic while the MERGE is being performed.

The similarity between the MERGE and OLD commands extends only to the selection and extraction steps, and not the insertion. The effect of executing a MERGE command is quite different from that of executing an OLD command. The MERGE never changes the name of f, the assumed compiler, or the mode of the working area. The lines being merged are always translated to the mode of the working area. A warning message is given if any lines are translated. When an error is detected, the MERGE operation is aborted and the contents of f are unchanged.

The program d is merged with or appended to the current working area. The lines from d are assigned line numbers starting with i and incremented by j as they are added to f. If i and j are not specified, the lines from d are edited into the working area with their original line numbers.

The i specification can have the following forms:

i = null	Use the line numbers from d.
i = n	Insert the lines starting with a line number of n.
i = +0	Insert the lines starting with the current line number.
i = *	Insert the lines starting with the current line number plus j.
i = !	Insert the lines starting with the number of the last working area line plus j.

The j specification may be used with each of the above forms of i. If j is omitted an increment of 10 is assumed. It may be included even if i is omitted. In this case j is added to each original program line number as it is inserted into the working area.

3.6.1. Resolution of Line Number Conflicts

The MERGE command can attempt to insert into f lines with line numbers which conflict with lines already in f. The DITTO (see 5.3.3), GENERATE (see 8.3.7.4), MOVE (see 5.3.4), and RESEQUENCE (see 5.3.2) commands may also create the same condition. A line number conflict occurs when one of these commands is editing lines into a specified position in the working area (i.e., between two working area lines) and the edited line number is greater than or equal to the next working area line number. CTS handles this situation in one of two ways, selectable by the ASSUME RESEQUENCE command (see 5.3.6). The two methods are the RESEQUENCE ON method, which is the standard default case, and the RESEQUENCE OFF method.

ASSUME RES ON causes the merged lines to remain as a sequential block of lines by moving down all lines following the point of insertion if a line number conflict occurs. Lines are resequenced until there is no longer a line number conflict. CTS produces a warning stating how far the resequencing was done. ASSUME RES OFF causes the merged lines to be inserted without disturbing the other working area lines unless a duplicate line number occurs. In this case the merged line replaces the working area line. ASSUME RESEQUENCE ON/OFF rules are followed for a MERGE which retains the original program line numbers or for a MERGE which generates new line numbers for the inserted lines.

The following examples illustrate the resolution of line number conflicts. For all of the examples of this paragraph, assume three saved elements A, B, and C, in F with contents as follows:

A contains:

```
10 LINE 1
20 LINE 2
30 LINE 3
```

B contains:

```
10 LINE A
20 LINE B
30 LINE C
```

C contains:

```
1 LINE D
2 LINE E
3 LINE F
```

For a simple example:

```
->OLD A
->MER 20,5 B
->P A
10 LINE 1
20 LINE A
25 LINE B
30 LINE C
35 LINE 2
40 LINE 3
END OF FILE
->
```

Note that lines 20 and 30 of A were resequenced to resolve the line number conflict.

In the RESEQUENCE OFF method, the line number conflict is resolved by deleting the existing line and replacing it with the new line. The following examples show how line number conflicts are resolved in the DELETE method. Assuming A, B, and C as before:

```
->ASSUME RES OFF
->OLD A
->MER B 10,25
->P A
10 LINE A
20 LINE B
30 LINE 3
END OF FILE
->
```

This time the original lines with line numbers 10 and 20 were replaced by lines from B. No additional conflicts occurred. Lines from B retain the line numbers they had when they were saved because no starting line specification (i) was given on the MERGE.

3.6.2. MERGE Examples

For all of the following illustrations, assume the following three elements in F: A, B, and C which have contents as in the preceding paragraph.

There are a number of situations with the MERGE command which have no counterpart in the OLD command. Primarily, they are associated with the parameter i and with the insertion step in the implementation of the MERGE command. These are illustrated by examples in the remainder of this paragraph.

- To append the contents of element B to the contents of A:

```
->OLD A
->MERGE ! B
->LIST
10 LINE 1
20 LINE 2
30 LINE 3
1 LINE D
2 LINE E
3 LINE F
END OF FILE
->
```

- To interleave the lines of B instead of appending them, set *i* to 15. In addition, specify L if all of B is not wanted:

```
->ASSUME RES OFF
->OLD A
->MER 15 B 10,20
->P A
10 LINE 1
15 LINE A
20 LINE 2
25 LINE B
30 LINE 3
END OF FILE
->
```

  This interleaving can only be done if the ASSUME RESEQUENCE mode is OFF. If the ASSUME mode is ON the lines being inserted are maintained as sequential lines and the other lines of f are resequenced if necessary.

- If the i, j parameters are missing, the lines are inserted with the line numbers they had when they were saved by a SAVE or REPLACE:

```
->OLD B
->MERGE C
->P A
1 LINE D
2 LINE E
3 LINE F
10 LINE A
20 LINE B
30 LINE C
```

```
END OF FILE
->
```

- If the i parameter is missing and the j is specified, the lines are inserted using their old line numbers plus the increment j:

```
->ASSUME RES ON
->OLD A
->MER ,10 C
->P A
10 LINE 1
11 LINE D
12 LINE E
13 LINE F
20 LINE 2
30 LINE 3
END OF FILE
->
```

- If a negative value is used for i, the MERGE command is rejected:

```
->OLD A
->MERGE ,-5 B
<77> ILLEGAL LINE LIMIT SYNTAX, -5
->P A
10 LINE 1
20 LINE 2
30 LINE 3
END OF FILE
->
```

This error occurred because -5 is not valid syntax for j.

3.7. Selecting Data Mode - DATA

Syntax: DATA

Abbreviation: None

Function: To place CTS in the DATA mode, whereby certain CTS commands are restricted to operating on data files.

Many of the CTS commands operate on either a data file or an element of a program file. The difference between these two formats is discussed in 7.1. The normal mode for CTS is ELEMENT mode. In ELEMENT mode, those CTS commands sensitive to ELEMENT and DATA modes can be coded to operate on either type of file. In DATA mode they will operate only on data files. If, while in DATA mode, an operation on an element is requested, or if a program file is specified, a diagnostic is given and the command is rejected.

The format of the d-field specifying the element or file name can be so coded as to expressly define an element, expressly define a file, or define a name which could be either a file or element. The general format of the d-field specification for any command is "FILE.ELEMENT". The period following or preceding a name expressly defines that name as a file or element name respectively. The file name part of the specification may contain any valid operating system file name including qualifier and read/write keys. When a qualifier is specified, the syntax distinguishes that name from an

element name without specifying the period. See 3.1.1 for a description of the d parameter of the SAVE command. When this parameter is coded to be deliberately ambiguous, CTS will interpret it as the name of a data file when in DATA mode, and as an element of F when in ELEMENT mode.

The CLEAR command, or the initiation of a prescan module, terminates the DATA mode and reestablishes ELEMENT mode.

Commands sensitive to DATA and ELEMENT modes are:

SAVE	(3.2)
REPLACE	(3.3)
OLD	(3.5)
MERGE	(3.6)

4. Displaying and Printing Programs

4.1. Printing and Listing at Terminals

This section describes how to display programs or parts of programs on terminals.

4.1.1. Displaying of f – PRINT

Syntax: PRINT [L] [(c1,c2)] [k]

Abbreviation: P

Function: To display on the terminal all or part of the contents of f.

The PRINT command is used most frequently to look at a few lines of f.

The PRINT command offers a great deal of flexibility which is very useful in special situations. The first parameter, L, defines the line number range in f which the PRINT command is to use. Any of the specifications given in 2.2.8.2 may be used. If this parameter is missing, only the current line is selected. The "current" line is that line which has the line number equal to the value of the line pointer, p (see 2.2.7).

The second parameter, always enclosed in parentheses, is the range of column numbers to be displayed. Only columns c1 through c2 inclusive will be displayed. If the entire parameter is missing, the default value is governed by the ASSUME PCOLUMN command. Similarly, if either c1 or c2 is omitted, its value is taken from the ASSUME PCOLUMN command. The value of this default parameter before the first ASSUME PCOLUMN command is (1,132).

The third parameter, k, controls the display of line numbers. If it is N, the line numbers are not displayed. If it is P, the line numbers are displayed. If it is S a scale is displayed above the lines displayed. (See SPERRY UNIVAC Series 1100 Conversational Time Sharing (CTS) System, UP-7940 (current version).) If omitted, the ASSUME TYPE command is used. The LIST, QUICK, SITE, CARDS, and PUNCH commands use parameter k in the same way. The ASSUME TYPE command (see 4.3.7) is used as the default. The initial default prints the line numbers.

This same parameter controls the display of line numbers by the INSERT command (see 5.2.4).

Any of the three parameters may be omitted in any combination.

The PRINT command never changes the contents of f or its name. It is never concerned with any file other than f. The PRINT command uses the line pointer p and changes its value. If the line number specification is not given in a PRINT command, only the line specified by the current value of p is displayed. The PRINT command usually leaves p set to the line number of the last line displayed.

There are three special cases, all of which CTS notes with a message:

- If the last image displayed is:

  ```
  TOP OF FILE
  ```

  p is reset to 0.

- If the last image displayed is:

  ```
  END OF FILE
  ```

  p is reset to 0.

- If the error displayed is:

  ```
  <21> LINE n DOES NOT EXIST
  ```

  p is left unchanged.

The TYPE (see 4.3.4), COLUMN (see 2.2.1), PRINTWIDTH (see 4.3.1), PCOLUMN (see 2.2.4), and QUICK (see 4.3.2) subcommands of the ASSUME command affect the operation of the PRINT command.

The following examples illustrate the use of the PRINT command and the response of CTS to various situations involving its use.

In each of these examples, assume that the contents of f consist of the following lines:

```
10 LINE 1
20 LINE 2
30 LINE 3
40 LINE 4
50 LINE 5
```

- To display the entire contents of f, use the line specification A.

  ```
  ->PRINT A
  10 LINE 1
  20 LINE 2
  30 LINE 3
  40 LINE 4
  50 LINE 5
  END OF FILE
  ->
  ```

  The final message indicates that the line pointer p, is set to 0.

- To display selected lines of f, specify the limits of the line numbers. These limits need not correspond to an existing line number, but the range must include at least one actual line.

```
->PRINT 20,40
20 LINE 2
30 LINE 3
40 LINE 4
->
```

This time p is left set to 40.

- To display a single line, L is specified as its line number.

```
->P 30
30 LINE 3
->
```

The pointer, p, is left set to 30. A PRINT command with a blank L specification will display this line:

```
->P
30 LINE 3
->
```

- If the range of line numbers specified by L is in reverse order, they will be displayed in reverse order.

```
->P 30,5
30 LINE 3
20 LINE 2
10 LINE 1
TOP OF FILE
->
```

The final message indicates that p has been set to 0.

- The second parameter specifies the display of only certain columns. The selected columns are printed left justified.

```
->P 20,40 (4,6)
20 E 2
30 E 3
40 E 4
->
```

This feature can be useful when creating a program for a compiler which ignores character positions beyond 72. It may happen in the creating and editing of such a program, that some lines extend beyond this limit.

- If the first part of the second parameter (c1) is missing, it is usually taken to be 1, and the columns from the beginning of the line through the second parameter are displayed. The comma must be present. See 2.2.4, ASSUME PCOLUMN.

```
->P 10 (,3)
10 LIN
->
```

- If the second part of the second parameter (c2) is missing, it is usually taken to be 132 or the end of the line. The columns from the first parameter through the end of the line are displayed. The comma is optional. Again, the ASSUME PCOLUMN command applies here.

```
->P 20 (3)
20 NE 2
```

- To omit the display of line numbers, code the k parameter with N.

```
->P A N
LINE 1
LINE 2
LINE 3
LINE 4
LINE 5
END OF FILE
->
```

Specifying k does not change the ASSUME TYPE:

```
->P 20,30
20 LINE 2
30 LINE 3
->
```

- If the range of line numbers specified by L includes no actual lines of f, a message is printed depending on whether the range is before the smallest line number, between two line numbers, or after the largest line number of f.

```
->P 5,7
TOP OF FILE
->P 60,70
END OF FILE
->P 15,17
<110> SPECIFIED LINES DO NOT EXIST
->
```

p is set to 0,0, and unchanged, respectively.

- If f is empty, CTS gives the message:

```
->P A
THE WORK AREA IS EMPTY
->
```

The value of p is unchanged.

- If the parameter L specified has incorrect syntax, the response depends on the nature of the error, but CTS usually interprets the invalid character as the end of the L specification and looks for a valid k parameter. This results in the error:

```
->P Z,20
<20> ILLEGAL COMMAND SYNTAX Z,20
->
```

The value of p is unchanged.

- An error in the syntax of the k parameter results in the same message:

```
->P 10 (3,4) NQ
<20> ILLEGAL COMMAND SYNTAX NQ
->
```

  Again, p is unchanged.

- An error in the syntax of the column specification usually results in the following response:

```
->P (4,Z)
<126> ILLEGAL COLUMN LIMIT SYNTAX,Z)
->
```

  Again, p is not changed.

4.1.2. Compact Display of f – QUICK

Syntax: QUICK [L] [(c1,c2)] [k]

Abbreviation: Q

Function: To display lines of f, shortening the output by compressing strings of multiple spaces into a single space.

Except for the compression of spaces, the QUICK command works exactly like the PRINT command. See 4.1.1 for details of the operation of the command, interpretation of parameters, etc.

The following example illustrates the difference between QUICK and PRINT:

```
->PRINT A
10 LINE          1
20 LINE          2
30 LINE          3
END OF FILE
->Q  A
10 LINE 1
20 LINE 2
30 LINE 3
END OF FILE
->
```

4.1.3. Spacing Images in CTS Output Listing – SKIP

Syntax: SKIP [*n*]

Abbreviation: SKI

Function: To place blank lines into output listing.

The SKIP command places blank lines in the CTS output listing. The argument *n* specifies the number of blank lines to produce. If *n* is equal to 0 or greater than 255, then a page eject is performed as a result of the command. If *n* is omitted, then 1 is assumed for *n*.

NOTE:

The SKIP command is output device dependent (i.e., certain devices do not perform full page ejects, but instead skip three lines) so individual device response should be tested prior to using the SKIP command.

4.1.4. LIST

The first parameter of the LIST command is interpreted by CTS as a subcommand indicator, each valid parameter leading to a substantially different result. The LIST command is, therefore, in reality a set of commands. Three of these frequently used in program preparation are discussed here. Those LIST commands dealing with system interrogation are discussed in Section 9.

4.1.4.1. Displaying f – LIST L

Syntax: LIST [L] [(c1,c2)] [k]

Abbreviation: LIS

Function: To list on the terminal all or a part of f.

This command is used to list the entire contents of f.

With a few exceptions, the LIST L command behaves exactly like the PRINT command (see 4.1.1). The parameters have the same significance, and the results are identical. However, the following difference exists:

- If the L parameter is missing, it is interpreted as meaning that all of f is to be displayed rather than the single line defined by the line number currently stored in the line number pointer, p. In other words, the command:

  ```
  ->LIST
  ```

  gives the same result as:

  ```
  ->PRINT A
  ```

Keeping in mind this difference, refer to 4.1.1, the PRINT command, for an explanation of how the LIST L command works, and an illustration of the responses and diagnostic messages.

4.1.4.2. Displaying Names of Saved Elements – LIST SAVED

Syntax: LIST SAVED [, [P] [,i]] [d1 [,d2...]]

Abbreviation: LIS S

Function: To display the names and types of selected elements in one or more program files.

The LIST SAVED command is normally used to determine the names of the elements in the save file, F.

The following values of the option parameter, P, determine the type and amount of information listed for each d in the list:

A List only the absolute elements from each d.

D List deleted (unsaved or replaced) elements as well as nondeleted elements in d. A deleted element will have an asterisk (*) preceding its name.

E List all elements with name d regardless of their version name.

L List date and time that the element was created and its size, type, and name.

O List only the omnibus elements from each d.

R List only the relocatable binary elements from each d.

S List only the symbolic elements from each d.

V List only those elements in d which have the same version name as specified in each of the d parameters. An element name must be specified even though it will be ignored.

null No options defaults to the A, O, R, and S options.

The count parameter, i, limits the number of elements from d that will be listed. If i=5, for example, only the last five elements from each d of the type specified by P are listed.

If a d parameter is missing, it will be taken to mean F, which is the most common use of this command. If a series of file names, separated by commas is given, CTS displays the contents of each of the files. A blank parameter in any position except the last one is interpreted to mean F.

For each element, CTS displays the type and name. The type is ABS for absolute elements, OMN for omnibus elements, REL for relocatable elements, and assumed (compiler) for symbolic elements.

The responses of CTS to some LIST SAVED commands follow.

- The normal response is to display the elements.

```
->LIST SAVED
RUNA.
TYPE     NAME
RFOR   MAT
ELT    A
->
```

The parameter d is missing, so the F file is assumed. The run-id of this run is RUNA, so the standard file used for F has this name (see 7.1.2). The save file has two symbolic elements, A and MAT.

- Since d may specify an element name, the explicit file name syntax discussed in 3.2.1 must be used when referencing a file.

```
->LIST SAVED     FA, MAT
RUNA.
TYPE     NAME
<4> ELEMENT .FA CANNOT BE FOUND
```

```
RUNA.
TYPE     NAME
RFOR   MAT
```

The error was caused by not including a period after the file name FA, this FA was interpreted as an element name. If an element named FA existed in F (RUNA in this example) it would have been listed and no error would have been printed.

- If the elements of more than one file are to be displayed, the file names are separated by commas. If F is to be displayed, it must either be named explicitly or, if it is the first parameter, left blank.

```
->LIS  S          ,FA,FB
RUNA.
TYPE     NAME
RFOR   MAT
ELT    A

FA.
TYPE     NAME
ABS    A
NUALG NAME
<105> FILE FB IS EMPTY
->
```

The name of each file precedes the list of elements from that file and a blank line follows the list for each file.

If a data file is specified, the error message:

```
<19> file name IS NOT A PROGRAM FILE
```

is displayed. If one of the files specified does not exist, the error message:

```
<68> FC IS NOT CATALOGUED.
```

is displayed.

- The date and time that elements in a file were created, as well as their size and type, are displayed by using the L option on the LIST SAVED command.

```
->LIST SAVED, L
HKH.
DATE              TIME      SIZE   TYPE       NAME
15 FEB 77       09:30:11    12   RFOR      FFF
14 FEB 77       14:28:38    10   BASIC     DDD
14 FEB 77       09:27:27    1    BASIC     ABC
->
```

4.1.4.3. Displaying the Names of Assigned Files – LIST INUSE

Syntax: LIST INUSE

Abbreviation: LIS I

Function: To list the names of all mass storage files assigned to the run, and pertinent information about each.

Generally, it is not necessary to know which files are assigned to the run because CTS automatically assigns files when they are referenced. The LIST INUSE command causes CTS to list the full name of each file assigned to the run, including its qualifier and absolute cycle. It also gives the type of mass storage equipment requested for the file, the options used when assigning it to the run, and any additional names d uses to simplify referencing it.

Each file is represented by one line in the following format:

q*FN(c),e,op $n_1, n_2, \ldots, n_i$

For a complete understanding of these fields, a substantial knowledge of the assignment and use of mass storage files under the operating system is required. See the SPERRY UNIVAC Series 1100 Executive System, Volume 2 EXEC, Programmer Reference, UP-4144.2 (current version), as well as Section 7 of this manual. Briefly, the fields and their significance are:

q – qualifier. Every file has an additional name, called a qualifier. The qualifier may be explicitly specified when the file is created. If not, the Executive uses the project field from the @RUN control statement which started the run. The qualifier and file name together uniquely identify a file. No two cataloged files may have identical file names and qualifiers, although either file names or qualifiers of different files may be identical. A qualifier may be given when referencing a file. If not, the project field is again used as a default. In any case, both the qualifier and file name must match exactly those which the Executive has registered for the file.

FN – file name. This is the basic name of the file. It is the name explicitly given to a file when it is created and which is used to reference it via the Executive. (CTS provides the names of some of these files, as mentioned in 7.1.2.)

c – cycle. The Executive permits several versions of a cataloged mass storage file to exist simultaneously in the system, distinguishing between them by a number called a cycle number. When referencing such a file, unless a particular cycle is specified, the most recent cycle is used (i.e., the highest cycle number).

e – equipment type. When assigning a file for the first time, it is possible to specify the type of equipment to be used for the file – high speed drums, disks, FASTRAND drum, etc. The type of equipment is specified by a code, and it is this code that constitutes the e–field.

op – options. The options on the CREATE command (see 7.5.1), or @ASG control statement by which this file was assigned to the run. T means the file is temporary, A means it was cataloged before the assignment, and C means it was not cataloged before this run, but will be when the run terminates normally.

$n_1, n_2, \ldots, n_i$ – attached names.

The Executive has a feature which permits the definition of a simple alias for the complete name of a file (see the USE command, 7.5.6). Such an alias is called an attached name. Attached names are useful in simplifying references to a file the full specification of which (if qualifier and cycle are specified) may otherwise be unwieldy. They are also useful in cases where, in the course of a run different files are to be used for the same function. In such cases, the attached name is simply redefined to be an alias for the new file. Actual references to the file can always use the attached name and, therefore, always refer to the file currently used for the function.

An example of a LIST INUSE command is:

```
->LIST INUSE
FURPUR 27R2     02/15/77   14:56:45
MKTG*TPF$(0),F4,T
MKTG*CTS$RUNA(1),F4,AD       CTS$FILE
MKTG*RUNA(1),F2,A
->
```

This command was executed immediately after initializing CTS during a run with a run-id of RUNA and a project of MKTG. The three files are, in order:

1. The object file.
2. The working area file, f.
3. The save file, F.

The first line is produced by the FURPUR processor, part of the operating system. CTS uses this processor to retrieve the information about the files.

4.2. Sending the Output to Another Device

There are times when the volume of output to be printed is too large to display at the terminal. At other times a hardcopy record of a program may be needed but the terminal has no hardcopy device, as with many CRT terminals, for example. A program may also be needed on punched cards. To take care of these situations, two commands (SITE and CARDS) direct output to the onsite equipment – equipment at the same site as the central computer. Each of these commands specifies what is to be written or punched on what device, and informs the central site operator what to do with the output.

4.2.1. Sending Output to an Onsite Device – SITE

Syntax: SITE [L] [,(c1,c2)] [k]

Abbreviation: SIT

Function: To direct the output of all or the specified part of f to an onsite device.

The SITE command is used exactly like the PRINT command (see 4.1.1) except that the resulting output goes to an onsite device rather than to the terminal. The parameters have the same significance as they do for the PRINT command, except that the ASSUME OCOLUMN limits are used as the defaults if no column limits are specified. If L is omitted, however, it is assumed to be A.

The SITE command normally sends output to any onsite printer. The use of the ASSUME SITE command (see 4.3.3) permits selection of a specific device or group of devices to which the output of SITE commands will be limited. The output of SITE commands may be printed on a remote device, such as a UNIVAC 9200/9300, by specifying its identification in the ASSUME SITE command or in response to the solicitation.

The SITE command will solicit a string to use as a heading for the listing, a message for the operator specifying what to do with the output, the number of copies to be generated, and a site identification. Each of these parameters may be set via the ASSUME command (see 4.3.3, 4.3.4, 4.3.5, and 4.3.6). If a parameter has been set, the SITE command will not ask for it.

The following example illustrates these responses:

```
->SITE
HDG? >PROGRAM ABC
RTN? >JOHN DOE, MAIL STATION 1234.
COPY? > 2
SITE? >
->
```

The entire contents of f was sent to an onsite printer. The indicated heading is at the top of every page, and the message appears at the end on a separate page.

The ASSUME SITE (4.3.3), ASSUME HEADING (4.3.4), ASSUME RETURN (4.3.5), ASSUME COPY (4.3.6), ASSUME QUICK (4.3.2), and ASSUME TYPE (4.3.7) commands affect the operation of the SITE command.

4.2.2. Output to Punched Cards – CARDS

Syntax: CARDS [L] [(c1,c2)] [k]

Abbreviation: CAR

Function: To send all or part of f to a card punch.

The CARDS command is used exactly like the PRINT command (see 4.1.1), except that the resulting output goes to an onsite card punch, rather than the terminal, and the ASSUME OCOLUMN limits are used as the defaults if no column limits are specified. The parameters have the same significance as they do for the PRINT command. If L is omitted, it is assumed to be A.

NOTE:

Unless the k parameter is N, CTS line numbers will be punched as part of the card image.

The CARDS command solicits a string to be used as a heading card. This heading will be the first card punched and will identify the card deck.

The CARDS command solicits a string to be used as a message informing the operator what to do with the card deck produced: only if a carriage return is entered in response to the SITE? query. This message will be displayed on the operator's console at the central site and is meaningless if the card punch is located elsewhere.

The example that follows illustrates the use of the CARDS command.

```
->CARDS    N
HDG? >PROGRAM ABC
COPY? >
SITE? >
RTN? >SEND DECK TO JOHN DOE, MAIL STATION 1234.
->
```

One copy of the entire contents of f was punched without CTS line numbers. The message has been displayed on the operator's console and the cards are ready to be punched when the solicitation character appears.

```
-> CARDS
HDG? -> PROGRAM DEF
COPY?->
SITE?-> CP3A
->
```

In this example, the RTN? query does not appear. A message displayed at the operator's console may be meaningless since the output is not directed to the on site card punch.

The ASSUME SITE (4.3.3), ASSUME HEADING (4.3.5), ASSUME RETURN (4.3.4), ASSUME COPY (4.3.6), ASSUME QUICK (4.3.2), and ASSUME TYPE (4.3.7) commands affect the operation of the CARDS command.

4.2.3. Output to Paper Tape – PUNCH

Syntax: PUNCH [L] [(c1,c2)] [k]

Abbreviation: PUN

Function: To send all or part of f to the paper tape punch of the terminal.

The PUNCH command is used exactly like the PRINT command (see 4.1.1), except that the output is punched as well as displayed, and the ASSUME PCOLUMN limits are used as the defaults if no column limits are specified. It is intended for terminals with type II paper tape equipment. The parameters have the same significance as for the PRINT command, except that the default for a missing L parameter is A, rather than the current line. When the command is executed, a request to turn on the punch is printed and a pause allows time for this action.

CTS will cause a leader and trailer which consist of rubout characters to be punched. Since some devices will not punch the rubouts that are transmitted by CTS, try punching a small sample tape and if the leader and trailer are not punched out, manually create the leader by switching the paper tape punch on and holding the repeat key (REPT) and the rubout key down simultaneously. This may be done safely even while executing a run at the terminal, since rubouts are ignored.

When the command is executed, a request to turn on the punch is printed as follows:

```
->PUNCH
*DEPRESS PUNCH ON*
```

A pause then occurs to allow the PUNCH ON switch on the paper tape punch hardware to be pushed. After the leader, the text will be punched followed by an @EOF, a CONTRL–S, an @@END, and the trailer. When the punching is finished, a pause again occurs to allow the punch to be turned off. If a trailer was not punched, repeat the process used to create the leader. A tape so produced can be used as normal type II input.

4.2.3.1. Paper Tape Input – PTI

Syntax: PTI [i] [,j]

Abbreviation: None

Function: To start input from paper tape on devices with paper tape capability.

If the paper tape has line numbers punched on it, the i and j fields should be left blank. If there are no line numbers, CTS will use i and j to generate line numbers in a manner similar to the NUMBER command.

The generated line numbers will start with i and have increments of j. If i is not specified, 100 is assumed. The default for j is 10. An "*" or "!" may be used for i (see 2.2.4.2).

NOTE:

If line numbers are generated by CTS, the first character of a line must not be the command control character (usually an asterisk), because this will terminate the numbering.

The system responds to this command with the message START PAPER TAPE INPUT. When the input is completed the message END PAPER TAPE INPUT is printed.

If the END PAPER TAPE INPUT message does not appear after the input is finished, it means that there are no paper tape CONTRL–S, @EOF, and @@END control images on the paper tape. They must then be typed in to exit from this mode. The CTS PUNCH command automatically supplies these control images.

Input lines may be up to 132 characters in length. When the paper tape input is completed the 132 character limit remains in effect. To return to the standard 80 character line use the ASSUME INPUTWIDTH command or any command which causes an exit from CTS (see 4.2.3.2).

4.2.3.2. Setting the Line Length – ASSUME INPUTWIDTH

Syntax: ASSUME INPUTWIDTH [80/132]

Abbreviation: A INP

Function: To control the number of characters per line which can be entered from a terminal via paper tape.

The parameter should be either 80 or 132. If it is less than or equal to 80, or not specified, the input width is set to 80. If it is greater than 80, the input width is set to 132. The input width is set back to 80 when exiting from CTS on an XCTS command to protect other processors which are not prepared to read more than 80 characters of input.

4.3. Setting Defaults for Printing

4.3.1. Defining Terminal Line Length – ASSUME PRINTWIDTH

Syntax: ASSUME PRINTWIDTH [i]

Abbreviation: A PRI

Function: To limit the number of characters displayed on one line at the terminal.

Depending on the device, the number of characters CTS will display on a line may need to be increased or decreased. Use of the ASSUME PRINTWIDTH command will not cause truncation of a message longer than the line length. CTS will display the entire message on as many lines as it needs.

The parameter i is an integer determining the maximum number of character positions. For efficiency, it is rounded down, if necessary, to the largest multiple of six (for Fieldata) or four (for ASCII). Omitting i will reinstate the system standard, 132.

The effect of this command is local to CTS.

4.3.2. Compressing Output – ASSUME QUICK

Syntax: ASSUME QUICK [ON/OFF]

Abbreviation: A Q

Function: To establish or discontinue the compression of output to the terminal for certain commands by substituting a single space for any string of consecutive spaces.

This command establishes (ASSUME QUICK ON) or disables (ASSUME QUICK or ASSUME QUICK OFF) the compression of output to the terminal for the CHANGE (see 5.2.2), LOCATE (see 5.1.1), and FIND (see 5.1.2) commands, and the PRINT (see 4.1) family of commands.

4.3.3. Defining an Onsite Device – ASSUME SITE

Syntax: ASSUME SITE [X]

Abbreviation: A SIT

Function: To define the device or device group for the SITE command.

The SITE command (see 4.2.1) directs an output listing to any valid device that is configured into the EXEC. CTS normally directs such output to any onsite printer. The X parameter specifies the name of a particular device or the name of a device group as being the target for output of subsequent SITE commands.

->ASSUME SITE RMSU01

will cause subsequent output to go to the device whose site-id is RMSU01. The specific site-ids must be obtained from the site coordinator.

4.3.4. Specifying a Default Heading – ASSUME HEADING

Syntax: ASSUME HEADING [s]

Abbreviation: A HEA

Function: To set a default heading for the SITE command

The ASSUME HEADING command sets default values for the heading to be used by the SITE and CARDS commands.

4.3.5. Specifying a Default Return–to Message – ASSUME RETURN

Syntax: ASSUME RETURN [s]

Abbreviation: A RET

Function: To set a default return–to message for the SITE and CARDS commands.

The ASSUME RETURN command sets default values for the return–to message to be used by the SITE command.

4.3.6. Setting the Number of Copies – ASSUME COPY

Syntax: ASSUME COPY [i]

Abbreviation: A COP

Function: To set a default value for the number of copies

The ASSUME COPY command sets default value i for the number of copies to be generated by the SITE and CARDS commands.

4.3.7. Controlling the Line Number Display – ASSUME TYPE

Syntax: ASSUME TYPE [ON/OFF]

Abbreviation: A T

Function: To control whether line numbers are displayed with the PRINT, PUNCH, LIST, QUICK, SITE, and CARDS commands.

The PRINT, PUNCH, LIST, QUICK, SITE, and CARDS commands use the same parameter formats and the interpretation of the parameters is essentially identical. The PRINT command (see 4.1.1) describes the parameters and their interpretation. The parameter in these commands is used to explicitly control the display of line numbers for that command only.

The ASSUME TYPE command changes this option. ASSUME TYPE OFF conditions CTS to not display the line numbers. If it is omitted or coded ON, the option is set to display them.

4.3.8. Sending Print to an Alternate File – ASSUME BREAKPOINT

Syntax: ASSUME BREAKPOINT [ON/OFF]

Abbreviation: A BRE

Function: To direct output from the COMPILE, MAP, and to either the terminal or to a print file named SQUELCH$, which is automatically provided by CTS.

The ASSUME BREAKPOINT OFF command directs the output to the terminal and suppresses the DIAGNOSTIC SCAN? query. Since there is no scan file (SQUELCH$), the SCAN command will generate an error or will read the old scan file, if a RUN, COMPILE, or MAP command was done before the ASSUME BREAKPOINT OFF. The ASSUME BREAKPOINT ON or ASSUME BREAKPOINT reinstates the default condition in which output goes to the scan file.

5. Editing and Modifying Programs

5.1. Locating Information in f to be Modified

Format errors are often easily detected by displaying the entire program. A refinement is to list only a part of every line—beyond column 72 or 80, for example. This can be useful with some compilers. CTS provides three commands for this purpose which are almost identical. They are the PRINT, QUICK, and LIST L commands, which are described in detail in 4.1.1, 4.1.2, and 4.1.4.1, respectively. Two commands are available which will locate and display only those lines which contain a specified string. They are the LOCATE and FIND commands. These are described in detail here, along with some related commands which modify their operation.

5.1.1. Finding a String – LOCATE

Syntax: LOCATE s [L] [(c1,c2)] [FILLER=b] [SPACER=a] [R [=i]]

Abbreviation:
LOCATE L
FILLER F
SPACER S

Function: To search all or a specified part of the working area, f, for the occurrence of a specified string.

The LOCATE and FIND (see 5.1.2) commands are similar in function. The LOCATE searches for the given string in any column position: the FIND, only in the one specified position. For this reason it is more efficient to use FIND if the column position is known. The FIND command offers the option of applying the relations >, <, >< (not equal), =, >=, or <=. The LOCATE command does not. The FIND command has no counterpart to the FILLER and SPACER parameters of the LOCATE command.

The LOCATE command may be used in a number of ways. The various parameters permit much flexibility. Most frequently, this command will be used to find either the next occurrence or all occurrences of the exact literal string submitted. The operation of LOCATE can be explained by describing each parameter.

- s – String Parameter.

  This parameter specifies the string to be matched. If the string contains a space, it must be enclosed in quotes. If it contains a quote character, two adjacent quotes must be entered for each desired quote. For example, the string DON"T would match the string DON'T. To LOCATE

a string that contains the variable delimiter character (the percent sign,%), it is necessary to code two adjacent variable delimiter characters to avoid variable substitution or the start of a comment. For example, the string '% I' would be interpreted as the start of a comment, but the string '%% I' would be a valid string which could be used to find a comment. The FILLER and SPACER parameters permit variable spacing or character positions which are not to be used in the comparison. If there is no FILLER or SPACER, only an exact match of the given string will result in a find. A default string character can be set via the ASSUME STRING command (see 5.1.8.).

- L – Line Number Range Parameter.

  Specifies the range of line numbers of f to be searched. Any of the forms given in 2 2.8.2 may be used. If omitted, all lines following the current line are searched (equivalent to specifying *+).

- (c1,c2) – Column Limits Parameter.

  When this parameter is present, only columns c1 through c2 (inclusive) will be used during the search. No string lying partly outside these limits will result in a find If c1, c2 or the entire parameter is omitted, the corresponding value will be taken from the default established by the last execution of an ASSUME SCOLUMN command (see 2.2.5). On initiation of CTS, this default is set to (1,132). The value of c1 must not be greater than the value of c2. The parameters c1 or c2 may be strings (see 2.2.1).

- FILLER – Filler Parameter.

  There may be character positions in the given string which are not to be used in the matching process. The FILLER parameter allows these character positions to be specified in the string submitted by the s parameter. The positions so specified are then not used in determining whether a match has occurred. In other words, they will be considered to be equal to any character.

  Columns of the given string containing the character specified by the b in the FILLER parameter are the positions identified to be excluded from the match.

  For example:

  *->L A***T FILLER=**

  will result in a match when encountering ALOFT, ALL T, OR AB?*T, but not when encountering ABCT, ****T, or AT.

  If this parameter is omitted, the last execution of the ASSUME FILLER command (see 5.1.6) defines the filler character. If this command has not been executed, there is no filler character.

  Coding this parameter FILLER= (with no character specified) will cause the blank to become the filler character. Coding it FILLER (without the = as well) defines the filler character to be nonexistent for the current command.

- SPACER – Spacer Parameter.

  This parameter makes it possible to direct that strings of the spacer character (usually the blank) of different sizes are equivalent for purposes of matching. The specification "a" is the character for which a minimum number of contiguous character positions of this same character in the string being matched against it which will test equal in the matching process.

For example:

```
->L  AB#C      S=#
```

will result in a match when the text contains AB#####C, AB##C, and AB###C, but not when it contains AB#C or ABC.

The blank is the most commonly used spacer character. If the spacer parameter is omitted, the value established by the last execution of the ASSUME SPACER command (see 5.1.6) is used. Without this command, the spacer character will be nonexistent.

If this parameter is coded SPACER= (with no a specified), the spacer character is a blank. If it is coded SPACER (no = or character), the spacer character does not exist for this execution.

- R – Repeat Parameter.

  If this parameter is coded R, all matching lines in the range defined by L will be displayed. If this parameter is omitted, the first matching line is displayed, and the line pointer, p (see 2.2.3), is set to the line number of this matching line. If the L parameter is specified as A, R is always implied and need not be specified.

  The i parameter is an optional limit value for the R parameter, specifying the maximum number of matches to be made within the specified line limits.

The order specified for the first three parameters, if they exist, is mandatory. The last three parameters may be permuted among themselves, but must follow all of the first three which exist. A violation of this rule results in a diagnostic message, as in the following sequence:

```
->L  ABC  R  A
<17> KEYWORD A
->
```

If the string is omitted a diagnostic will be printed:

```
->L
<12> UNBALANCED DELIMITER.
->
```

However, if any parameters are present, the first parameter will be taken to be S. Other errors, as in the syntax of L or the column parameter, give similar diagnostics.

The normal response, when a match is made, is to display the line number and contents of the matching line. If R is in effect, matching then continues with the following lines until the end of the line number range specified by L is reached.

For example:

```
->L  ABC  100,150
135 LEARN YOUR ABC'S.
->
->L  ABC  100,150 R
135 LEARN YOUR ABC'S.
147  THE ABC OF IT.
->
```

In the first example p was left set at 135; in the second, at 147. As with other commands, if L is such that the message:

TOP OF FILE

or

END OF FILE

or

<21> LINE n DOES NOT EXIST

occurs, then p will be 0, 0, and unchanged, respectively. Other errors will usually leave p unchanged as well. An example of the R = i option is:

```
->L ABC 100,150 R = 1
135 LEARN YOUR ABC'S.
->
```

If the search was unsuccessful, the following message appears:

```
->L  XYZ  100,150  R
*NOT FOUND
->
```

The line pointer is left set to the highest numbered line between 100 and 150 inclusive.

If the ASSUME BRIEF ON command is in effect, no lines or line numbers are displayed.

For example:

```
->ASSUME BRIEF ON
->L  ABC  100,150
->
```

Assuming the same contents of f as for the previous example with the same LOCATE command, the only difference is the display of the matched line. The absence of the * NOT FOUND message indicates that a line was found.

If the ASSUME OCCURRENCES ON command is in effect, the number of lines the string occurred in is printed.

For example:

```
->ASSUME OCC ON
->LOC ABC ALL
100 ABC
110 ABC DEF GHI
150 LEARN YOUR ABC'S
170 THE ABC OF IT
NUMBER OF OCCURENCES:   4
```

5.1.2. Finding a String – FIND

Syntax: FIND s [L] [(c)] [R [=i]] [k]

Abbreviation: F

Function: To search all or a specified part of the working area, f, for the occurrence of a string in the fixed column positions specified which bears the specified relationship to the string given in the command.

The FIND command is similar in function to the LOCATE command (see 5.1.1). The FIND command searches for a string in the fixed column position only; the LOCATE, in any column position. The FIND command also permits specifying the relations >, <, >< (not equal), =, >=, and <= in the sense of the collating sequence. The LOCATE command does not. Both commands normally search for strings which, in this sense, are equal. The LOCATE command permits adjusting the matching properties of the specified string with the FILLER and SPACER parameters. The FIND command has no such capability.

The FIND command will be used most frequently to find the next occurrence or all occurrences of a specific string in a specific column position. In a COBOL program, for example, the line numbers of all file descriptions may be wanted. Searching for the string "FD" beginning in column 8 will accomplish this. Similarly, searching for FD ΔΔOLD–MASTER beginning in column 8 will find the line where the specific file description occurs.

- s – String Parameter.

  This parameter defines the string to be used in the search. If the string contains spaces, it must be enclosed in quotes. If it contains a quote character, two adjacent (single) quotes must be entered for each desired quote. For example the string "s would match the string 's. To FIND a string that contains the variable delimiter character (the percent sign,%), it is necessary to code two adjacent variable delimiter characters to avoid variable substitution or the start of a comment. For example, the string '% I' would be interpreted as the start of a comment, but the string '%% I' would be a valid string which could be used to locate a comment. A default string character can be set via the ASSUME string command (see 5.1.8.). Tab characters will be interpreted, and the intervening positions will be filled with blanks (see 2.3.4). The tab positions refer to the column numbers relative to the line, not necessarily relative to the string.

  In other words, if a tab is set at column 15 and the tab character is ";", then the command:

  –> FIND A ; B 100,150 (10)

  will result in the string s being A ΔΔΔΔB where Δ stands for a blank. The column parameter (which follows) means that the first character of s goes into column 10. This is taken into account when interpreting the tab character.

- L – Line Number Range Parameter.

  Specifies the range of line numbers of f to be searched. Any of the forms given in 2.2.8.2 may be used. If omitted, all the lines following the current line will be searched (equivalent to using *+).

- (c) – Column Parameter.

  Specifies the column in the searched lines where the string being compared is to begin. The position in the searched line is fixed. This is similar to a column limits parameter, but since the position of the desired string is fixed, only one value is needed, rather than the usual two. If omitted, the value of c1 from the most recently ASSUME SCOLUMN command (see 2.2.5) is used. If no such command has been executed, 1 is used. Unlike other commands that have column limits, c may not be a string.

- R – Repeat Parameter.

  If this parameter is coded, all lines meeting the specified criterion will be displayed. If it is omitted, only the first such line is displayed, and the line pointer, p, is set to the line number of that line, making it the current line for the next command. It is not necessary to specify this parameter if the L parameter is specified with A. In that case, R is automatically assumed and all lines meeting the criterion will be displayed.

  The i parameter is an optional limit value for the R parameter, specifying the maximum number of matches to be found within the specified line limits.

- k – Relation Parameter.

  Acceptable values for this parameter and the corresponding relations are:

= or null	equal to s
>	greater than s
<	less than s
><	not equal to s
<>	not equal to s
>= or =>	greater than or equal to s
<= or =<	less than or equal to s

  where s is the string specified in the string parameter. These relationships refer to the collating sequence of the characters. If the string s is the single character D, for example, and k were coded as >, a line with E as the matching string would satisfy the relationship, and the line would be displayed.

The normal response, when all parameters are correctly specified, is to display the line number and contents of the line found. If R is in effect, the comparison process continues with each line until the end of the line number range specified by L is reached. Each successful find results in the display of the line. If the ASSUME OCCURRENCES ON command is in effect, the number of lines the string occurred in is also printed.

For example:

```
->F  ABC  100,200  (6)
157 YOUR ABC'S.
->


->ASSUME OCC ON
->F  ABC  100,200  (6) R
157 YOUR ABC'S.
183 67890ABCDEFG
NUMBER OF OCCURRENCES: 2
->
```

In the first example only the first find was displayed. In the second, all finds within the line number range were displayed.

If the repeat option (R = i) is typed in, at most, i lines (which contain a left-justified, fixed-column substring that matches s) will be found.

If there was no successful comparison, a message is displayed:

```
->F  ABC  100,200
*NOT FOUND
->
```

Any combination of parameters except s may be omitted, but the parameters not omitted must be in the order indicated. If the order is violated or a parameter is badly specified, a message like the following will occur:

```
->F  ABC  110,120  R (11)
<17> KEYWORD (11)
->
```

The FIND command will leave the line pointer, p, set to the line number of the displayed line when R is not specified. When R is specified or if no find is made, p is left set to the line number of the last line in the range specified by L. If the message:

```
TOP OF FILE
```

or

```
END OF FILE
```

or

```
<21> LINE n DOES NOT EXIST
```

appears, p is 0, 0, or unchanged, respectively. When other error messages are displayed, p is normally left unchanged.

5.1.3. Controlling the Display of Matched Lines – ASSUME BRIEF

Syntax: ASSUME BRIEF [ON/OFF]

Abbreviation: A BRI

Function: To control whether those lines which are successfully matched during the search performed by LOCATE, FIND, INLINE, INSERT, or CHANGE commands are displayed.

CTS normally displays both line numbers and line contents of matched lines for successful searches. The ASSUME BRIEF ON command conditions CTS to inhibit this display. The ASSUME BRIEF OFF command reinstates the original condition of displaying the matched and verification lines.

The ASSUME LINES command (see 5.1.4) independently controls the display of line numbers for successful searches. However, the line numbers are displayed only when in ASSUME BRIEF OFF mode. Thus both line numbers and line contents, the contents only, or neither may be displayed. These commands do not provide for listing the line numbers only.

5.1.4. Controlling the Display of Line Numbers of Matched Lines – ASSUME LINES

Syntax: ASSUME LINES k

Abbreviation: A LIN

Function: To condition CTS to display or not to display (depending on k) the line numbers of lines successfully matched during the execution of a LOCATE or FIND command and of verification lines printed by the CHANGE, INLINE, or INSERT command.

CTS normally displays both line number and line contents of matched lines for successful searches. An ASSUME LINES command with the parameter k coded OFF, causes CTS to inhibit display of the line numbers on all subsequent successful searches until it encounters an ASSUME LINES command with the k parameter coded ON. CTS then resumes the display of line numbers for successful searches. The display of both line numbers and line contents is inhibited by the ASSUME BRIEF ON command (see 5.1.3). The ASSUME BRIEF OFF command (see 5.1.3) reverses the effect of an ASSUME BRIEF ON command. Therefore, line numbers are displayed only when in ASSUME BRIEF OFF mode and only when the last ASSUME LINES command, if any, had its k parameter set to ON.

The ON may not be abbreviated and OFF can be abbreviated OF.

5.1.5. Reprinting of Lines Keyed into CTS – ASSUME ECHO

Syntax: ASSUME ECHO [ON/OFF]

Abbreviation: A ECH

Function: To control whether CTS types out each line keyed in.

The ASSUME ECHO command either turns on or off a mode in which each line keyed into CTS is typed back out. This mode is automatically established if CTS is called as a batch processor, so that the input lines are displayed with the output lines in the print listing. ASSUME ECHO ON establishes the echo print and ASSUME ECHO OFF or just ASSUME ECHO reinstates the normal condition where the lines are not echoed.

For example:

```
->ASSUME ECHO ON
->BAS
BAS
BBASIC 9R1
>>OLD ECHODEMO
OLD ECHODEMO
>>LIST
LIST
100 PRINT ' ENTER VALUES FOR A, B, AND C '
110 INPUT A, B, C
120 LET D = A*B/C
130 PRINT ' THE VALUE OF D = ' D
140 END
END OF FILE
>>RUN
RUN
```

```
ENTER VALUES FOR A, B, AND C
? > 4, 5, 10
THE VALUE OF D = 2

TIME :   .054
>>
```

5.1.6. Setting the FILLER Default for LOCATE – ASSUME FILLER

Syntax: ASSUME FILLER [=b]

Abbreviation: A FILL

Function: To set the default FILLER character for subsequent LOCATE commands.

The form of the parameter in the ASSUME FILLER command parallels its form in the LOCATE command The LOCATE command is the only command to which the default applies. Specifically, it does not apply to the CHANGE command (see 5.2.2) even though it has a similar parameter.

When the parameter and accompanying equals sign are omitted, there is no FILLER default. This is the standard state for this option.

See the LOCATE command (5.1.1) for details of how the FILLER parameter is used.

5.1.7. Setting the SPACER Default for LOCATE – ASSUME SPACER

Syntax: ASSUME SPACER [=a]

Abbreviation: A SP

Function: To set the default SPACER character for subsequent LOCATE commands.

The form of the parameter parallels its form in the LOCATE command. The LOCATE command is the only command to which the default applies. Specifically, it does not apply to the CHANGE command (see 5.2.2) even though it has a similar parameter.

When the parameter and accompanying equals sign are omitted, there is no SPACER default. This is the standard CTS state for this option.

See the LOCATE command (5.1.1) for details of how the SPACER parameter is used.

5.1.8. Setting the STRING Default – ASSUME STRING

Syntax: ASSUME STRING X = [s]

Abbreviation: A STR

Function: To set the default STRING character.

The ASSUME STRING command sets the string character, X, equivalent to the string, s, which can be used by the LOCATE, FIND, and CHANGE commands. The string character, X, is set by the user and is one character in length. Quotes are not needed as delimiters unless there are embedded blanks in the string and two quotes must be entered for each desired quote. Tab characters are not evaluated by the ASSUME STRING command.

For example:

```
->NEW ASTR
->1 ABCDEF
->2 123456ABC
->3 DEFDEF
->SAVE
->ASSUME STRING *=ABC
->LOCATE * R
1 ABCDEF
2 123456ABC
END OF FILE
->ASSUME STRING # = 123
->FIND #
2 123456ABC
```

5.1.9. Controlling the Printing of the NUMBER OF OCCURRENCES message – ASSUME OCCURRENCES

Syntax: ASSUME OCCURRENCES [ON/OFF]

Abbreviation: A OCC

Function: To control whether CTS prints the NUMBER OF OCCURRENCES message after a LOCATE, FIND or CHANGE command has been executed.

Frequently it is helpful to know the number of occurrences of a string after a LOCATE, FIND or CHANGE command. The ASSUME OCCURRENCES on command prints a message with this information.

The message "NUMBER OF OCCURRENCES: *m*" informs the user of the number of occurrences of the searched-for string and is printed only by the LOCATE, FIND, and CHANGE commands. Its meaning is slightly different for the CHANGE command than it is for the LOCATE and FIND commands.

The LOCATE and FIND commands print the number of lines the string occurred in. This is because these commands search through the line until the first occurrence of the string is encountered. There may be many occurrences of the string but searching is stopped when the first one is encountered. When the first string is encountered, the LOCATE and FIND commands are satisfied.

The CHANGE command prints the number of times the string actually occurred. This can be done with the CHANGE command since it may have to check for mutiple occurrences of the string in the same line.

If the ASSUME OCCURRENCES OFF command is in effect, the number of occurrences can still be referenced by the OCC ( ) command (see Table 12-1).

To locate the actual number of occurrences of a string, the CHANGE command can be utilized. If, on a CHANGE command, string S1 is the same as string S2 the work area will not be affected, but since the CHANGE command was used, the actual number of occurrences will be printed rather than the number of lines (which would be printed if a LOCATE or FIND command had been executed).

For example:

```
->A OCC ON
```

```
->NEW OCCS
1 ABCDEF
2 ABC123
3 123DEFDEFDEF


->LOCATE ABC A
1 ABCDEF
2 ABC123
END OF FILE
NUMBER OF OCCURRENCES:  2


->FIND 123 A (4)
2 ABC 123
END OF FILE
NUMBER OF OCCURRENCES:  1


->CHANGE /123/456/ A
2 ABC456
3 456DEFDEFDEF
END OF FILE
NUMBER OF OCCURRENCES:  2


->CHANGE /DEF/DEF/ A
1 ABCDEF
3 456DEFDEFDEF
END OF FILE
NUMBER OF OCCURRENCES:  4
```

5.2. Modifying Lines of f

The most elementary, and often easiest, way to edit a line is simply to replace it by entering a line with the same line number as the line to be changed. CTS has four commands (DELETE, CHANGE, INLINE, and INSERT) specifically designed to help make modifications to f. Frequently, more than one of them will be suitable to make a particular modification. At other times one will be clearly advantageous.

5.2.1. Discarding Part of f – DELETE

Syntax: DELETE [L] [(c1,c2)]

Abbreviation: D

Function: To delete from f selected lines or selected columns of selected lines.

The most common use of the DELETE command is to delete a set of lines from the working area, f. This is done by omitting the column specification. In this case, the lines of f with line numbers in the range specified by L are completely deleted from f. The line numbers no longer exist. This is not the same as deleting the contents of a line, which can be done, for example, with a DELETE command in which an explicit column specification is given. In this case, the line number is not deleted, and f still contains a blank line associated with this number.

- L – Line Number Range.

  This parameter defines a range of line numbers of f to be operated on. Any of the formats in 2.2.8.2 is valid for L. If L is omitted, only the current line is selected.

- (c1,c2) – Column Specification.

  This parameter defines a range of column numbers, the contents of which are to be deleted for each line selected (by the rule provided by the L parameter). If this parameter is omitted, CTS automatically supplies it. The normal default is (1,132). This default may be changed with the ASSUME ECOLUMN command (see 2.2.2).

  The DELETE command uses this parameter as the basis of deciding whether to remove lines from f entirely, or to operate only on their contents. If the resulting limits are (1,132) then CTS removes lines from f completely. The numbers of these lines no longer exist in f. If the resulting column limits are not (1,132) or either c1 or c2 is a string, then the line is never deleted, even if the contents disappear entirely. It remains in f as a blank line.

The DELETE command is usually used to remove a line from f. Frequently, as in the following example, the line pointer, p, is already set to this line.

```
->PRINT 110
110 THIS IS LINE 2.
->DELETE
->
```

The PRINT command set p to 110. The DELETE command removed this line from f. (This assumes that no ASSUME ECOLUMN command other than (1,132) is in effect.)

If a column parameter is explicitly specified and is not (1,132), the specified columns of each line are removed, and the trailing part of the line is shifted left to fill the vacancy.

```
->PRINT A
100 THIS IS LINE 1.
110 THIS IS LINE 2.
120 THIS IS LINE 3.
END OF FILE
->D A (9,13)
100 THIS IS 1.
110 THIS IS 2.
120 THIS IS 3.
END OF FILE
->
```

The word LINE has been deleted from each line. Continuing with the contents of f as left in the above example, the next example shows what happens when the entire contents of a line are deleted, both with the implied column specification and with an explicit one. No ASSUME ECOLUMN command is in effect.

```
->D 110
->D 120  (1,50)
->PRINT A
100 THIS IS 1.
120
END OF FILE
->
```

Line 110 has been completely deleted (no longer exists) and line 120, while empty, still exists.

If c1 is a string, and c2 is not specified, all columns beginning with the column that matches the string c1 are removed, for each line.

```
->PRINT A
100 THIS IS LINE 1.
110 THIS IS LINE 2.
120 THIS IS LINE 3.
END OF FILE
->D  A  ('L',)
100 THIS IS
110 THIS IS
120 THIS IS
END OF FILE
->
```

The normal set of diagnostic messages will result from improperly specified parameters. The normal response is simply the solicitation character. This indicates that the range specified in L contained some lines. An end-of-file message where L specifies a range with increasing line numbers, does not tell whether any lines were encountered or not. The diagnostics are normally self-explanatory.

For example:

```
->D  A
THE WORK AREA IS EMPTY
->OLD ELTA
->D  A  10,15
<17> KEYWORD 10,15
->D  A  (10,15
<11> UNBALANCED PARENTHESIS
->
```

The DELETE command leaves the line pointer, p, set to the line number of the last line operated on when the lines are not being completely removed from f. When the lines are being removed, p is left set to the smallest line number deleted. The current line, in this case, does not exist in f.

The DELETE command is the only command which can selectively discard lines of f.

5.2.2. Replacing Strings – CHANGE

Syntax: CHANGE 's1's2' [L] [(c1,c2)] [FILLER=b] [SPACER=a] [R [=i]] [O [=i]]

Abbreviation:
CHANGE C
FILLER F
SPACER S

Function: To locate occurrences of a given string and replace them with another given string.

The CHANGE command is probably the most useful single editing command. It is used to correct errors in a line of f. A great deal of flexibility is available with judicious use of its parameters. It can be made to scan a number of lines, searching within specified column limits for a match to the string s1. Each time it finds a match, it replaces string s1 in the line with string s2. The search can be specified for all occurrences of s1 on a line or for only the first.

The CHANGE command is one of the three CTS commands that search the contents of f for a string which meets a test of comparison against a given string. The other two are the LOCATE (see 5.1.1) and FIND (see 5.1.2) commands. The ASSUME BRIEF (see 5.1.3) and ASSUME LINES (see 5.1.4) commands affect the operation of the CHANGE command. CTS normally displays line numbers and new contents for all lines changed during execution of a CHANGE command. The ASSUME BRIEF and ASSUME LINES commands modify, eliminate, or reinstate this display of changed lines.

The searching phase of the CHANGE command is almost identical to the LOCATE command. The string s1 is used for the search. FILLER and SPACER characters have the same meaning in s1 as they do in the string of the LOCATE command. The ASSUME SPACER and ASSUME FILLER commands, however, affect only LOCATE, not CHANGE. The repeat parameter, R, is interpreted differently in the two commands.

- 's1's2' – String Parameter.

  This parameter specifies two strings, s1 and s2. String s1 is used for the search, and s2 is used to replace the matched string in the line. If s2 is not the same length as s1, the part of the line following s1 will be shifted to the right to make room for s2, or to the left to pack the line. Either s1 or s2 may be null (for example, ''AB'). The first nonblank character encountered is taken to be the string delimiter. The quote, while often used, has no unique significance in this regard. Obviously, the character used as the string delimiter must not be in either string. The slash (/) is often used.

  The default string character, set by the ASSUME STRING command (see 5.1.8.) may be used in place of either string S1 or S2 (but not both).

For example:

```
-> OLD ASTR
-> PRINT ALL
1  ABCDEF
2  123456ABC
3  DEF
-> ASSUME STRING # = 123
-> CHANGE /#/!!!/ ALL
2 !!!456ABC
END OF FILE
-> LIST
```

```
1 ABCDEF
2 !!!456ABC
3 DEFDEF
END OF FILE
->ASSUME STRING *=+++
->CHANGE /DEF/*/ ALL
1 ABC+++
3 ++++++
END OF FILE
->LIST
1 ABC+++
2 !!!456ABC
3 ++++++
END OF FILE
```

To change a string that contains the variable delimiter character (the percent sign,%), it is necessary to code two adjacent variable delimiter characters to avoid variable substitution or the start of a comment. For example, the command:

```
CHANGE /%'/I/
```

would result in an error since the first % would be interpreted as the start of variable substitution. However, the command:

```
CHANGE /%%I/I/
```

would cause the % to be removed from the current line.

The FILLER and SPACER characters (b and a) have special functions which are described in the following paragraphs.

- L – Line Number Range Parameter.

  This parameter defines a range of line numbers. Any line in f, the line number of which is in this range, is included in the search in the order (ascending or descending) indicated by the parameter. Any of the forms of 2.2.8.2 is acceptable. If L is omitted, only the current line is used. If L is specified as A, all lines of f are included and the R parameter is turned on whether or not it is coded explicitly.

- (c1,c2) – Column Specification Parameter.

  This parameter explicitly limits the search on each line to the column limits specified. The entire string must lie within these columns to be eligible for a match. If this parameter is omitted, the default is (1,132) unless it has been changed by an ASSUME SCOLUMN command (see 2.2.5). Omitting either c1 or c2 will cause substitution of the corresponding part of the current default column parameter. Parameters c1 or c2 may be strings (see 2.2.1).

- FILLER=b – Filler Parameter.

  This parameter has no default. If it is not explicitly coded in the CHANGE command, no filler character exists. Note the difference from the LOCATE command (see 5.1.1) where a default can be set up by an ASSUME FILLER command.

  The appearance of the filler character, b in s1 is taken to mean that this column position is not to be used for matching. Saying it another way, a b in s1 successfully matches any character at all. Thus, if s1 were:

  ```
  AB##E
  ```

and # is the filler character, then ABCDE, ABQ;E, and AB**E in a line of f would all match successfully, while A#CDE in a line of f would not.

When the filler character, b, also appears in s2, the character that matched the first filler of s1 is used in place of the first filler of s2, the character that matched the second filler of s1 is used in place of the second filler of s2, etc. If the number of filler characters in s2 is greater than the number of filler characters in s1, a diagnostic is given.

- SPACER=a – Spacer Parameter.

Like the filler parameter, this parameter has no default. If it is not explicitly specified, it does not exist. Again this is in contrast to the LOCATE command (see 5.1.1) where an ASSUME SPACER command establishes a default.

The spacer character, a, has a special significance only in s1. In s1 it is used to establish a minimum string of spacer characters which will match successfully with any string of similar characters in a line of f at least as long. If the spacer character is a blank, and s1 is:

GO ΔTO

then if a line in f has GO ΔΔTO, GO ΔTO, or GO ΔΔΔΔTO, a successful match is obtained. A line containing GOTO would not be matched. See 5.1.1 for another example of how SPACER works.

- R – Repeat Parameter.

If this parameter is present, all occurrences of string s1 on each line are replaced with s2. If it is not present, only the first occurrence of s1 in each line is replaced. For example, all lines of L are subject to at least one change whether R is coded or not. When the line number range parameter, L, is coded with A, the R parameter is assumed automatically, whether or not it is actually present. In this case, all occurrences of string s1 anywhere in f are replaced by s2.

The i option on either R or O specifies the maximum number of changed lines. For example:

```
->PRINT A
100 EERDVARK
110 MENDELA
120 ESTRODOME
END OF FILE
->GO 100
->C /E/A/ 100,120 R=2
100 AARDVARK
110 MANDALA
->LIST
100 AARDVARK
110 MANDALA
120 ESTRODOME
->
```

Each occurrence of s1 ("E") was replaced by s2 ("A") for a maximum of two lines in the range of lines specified (100,120).

- O – One Parameter

  The O parameter turns off the repeat option (i.e., at most only one change will be made per line). If both R and O are included, the last one specified will be used.

The CHANGE command is used most frequently to correct errors in a single line, as in the following example:

```
->PRINT  150
150 HME IS THD HUNTTER
->C /HM/HOM/
150 HOME IS THD HUNTTER
->C /D/E/
150 HOME IS THE HUNTTER
->C /TT/T/
150 HOME IS THE HUNTER
->
```

A misspelled word which occurs frequently in a program may also be corrected. Perhaps the identifier MXLIM in a FORTRAN program is misspelled as MAXLIM in a few places, and may occur elsewhere as well. Use the LOCATE command (see 5.1.1) to make certain they are in error, and the CHANGE command to correct them:

```
->LOCATE MAXLIM A
120         IF (K-MAXLIM)  25,,
135         I=MAXLIM
215         MAXLIM=MAXLIM+DELTA
->C /MAXLIM/MXLIM/ 120,215  R
120         IF (K-MXLIM) 25,,
135         I=MXLIM
215         MXLIM=MXLIM+DELTA
->
```

The LOCATE command assures that each occurrence of MAXLIM needs the change. The repeat parameter need not be specified in this LOCATE command, because it is automatically assumed when A is used for the line number parameter. The CHANGE command uses explicit line number limits, since the LOCATE showed the range needed. In this case, R must be explicitly specified. If not, the second occurrence of MAXLIM on line 215 would not have been changed.

It is possible to simplify the creation of a program by simplifying the spelling of commonly used words of the programming language during the input phase, and expanding them with the CHANGE command after all input has been created. For example, using ALGOL, the following substitutions might be made:

Symbol	Meaning
#B	BEGIN
#A	ARRAY
#P	PROCEDURE
#C	COMPLEX
#I	INTEGER

This would make keying in the program easier. After completing the initial keying in, use a series of CHANGE commands such as:

```
->C  /#B/BEGIN/ A
```

to expand the definitions to what the compiler expects. Then use:

```
->LOCATE  #  A
```

to find if any of the abbreviations have been missed. To find out if any of the substitutions expanded lines beyond column 72, use:

```
->FIND '                    ' A (73) R  ><
```

This will display any line with contents not equal to spaces in columns 73–87, since there are 15 spaces in the string. Refer to the FIND command (see 5.1.2) for details of this command.

The CHANGE command can give the normal set of diagnostics arising from improper specifications or error conditions. The next example shows a few of these:

```
->C  /ABC/123/ A
THE WORK AREA IS EMPTY
->OLD A
->C  /ABC/123/ A
*NOT FOUND
->C  /ABC/123 A
<12> UNBALANCED DELIMITER
->C  /A/1/  Z
<17> KEYWORD Z
->
```

As with other editing commands, the CHANGE command leaves the line pointer set to the line number of the last line scanned. If the message TOP OF FILE or END OF FILE occurs, p is set to 0. If an error message or the message <21> LINE n DOES NOT EXIST occurs, p is not changed.

5.2.3. Editing a Line – INLINE

Syntax: INLINE [L] [t]

Abbreviation: INL

Function: To facilitate insertion, deletion, and replacement of characters in a line, permitting the specification of the change by matching columns of the displayed line.

The INLINE command displays the line specified by L and solicits the editing string on the next line of the terminal. The editing function is indicated by a character (I for insert, R for replace, and D for delete) in the character position before the one where the editing is to become effective. This character is followed by a string terminated by the termination character (!) unless the parameter t is coded.

The parameter L determines which line is to be edited. If L is missing, the current line is used. Any of the forms of L given in 2.2.8.2 may be used, but if L specifies a range, only the first line in the range is used. The rest are ignored.

The character which terminates an editing string is !. If this character is to be part of the editing string, the termination character for this execution may be changed only by coding the t parameter. The first character of the string coded for this parameter will be taken as the termination character. If L is omitted the character A may not be used for it. If it were so coded, it would be taken as the L parameter rather than the t parameter.

The following examples show the responses obtained when the INLINE command is used:

```
->PRINT 102
102 ABCDEFGHIJ
->INL
+++ABCDEFGHIJ
+>   R345!
102 AB345FGHIJ
->INL   K
+++AB345FGHIJ
+>   ICDEK
102 ABCDE345FGHIJ
+>INL
+++ABCDE345FGHIJ
+>      D   !
102 ABCDEFGHIJ
->
```

The above example shows all three types of editing possible; replace, insert, and delete. The PRINT command set the line pointer to 102 and none of the other commands changed it, so no coding of L was necessary. The second INLINE command also shows the use of the optional terminator parameter, valid only for that execution. CTS inserts a +++ before the line image. This allows editing the first character of the line by placing an editing character under the last +.

Besides the usual diagnostic messages, the INLINE command can cause several unique ones:

```
->INL 102
+++ABCDEFGHIJ
+>   T  !
<84> BAD EDIT CHARACTER
->INL
+++ABCDEFGHIJ
+>
<86> NO EDIT CHARACTER
->INL   D
+++ABCDEFGHIJ
+> R123 !
<82> MISSING TERMINATOR CHARACTER
->
```

The line pointer, p, is always displayed and left set to the line number of the line selected for editing. Even if the editing itself causes an error, p is still left set to this line number. If the line number specification is such that no line is included, then p is 0 when the message TOP OF FILE or END OF FILE, occurs, and unchanged when the message <21> LINE n DOES NOT EXIST occurs.

5.2.4. Inserting Strings – INSERT

Syntax: INSERT S [L] [(c1,c2)] [k]

Abbreviation: I

Function: To insert a string into a specified field in specified lines of f.

The INSERT command performs an insertion on each line of f indicated by the line number range parameter, L. For each line, the INSERT command:

- Clears columns c1 through c2 of the line, either by discarding the contents or by shifting these and succeeding columns to the right.

- Inserts the given string, positioning it according to the specification in the k parameter.

- Either substitutes fill characters into those columns c1 through c2 which do not contain the string, or packs the line by left–shifting the portion beyond column c2 to fill these unused columns.

- s – String Parameter.

  This parameter provides the string to be inserted in each line (or to replace it). If the string includes a space, it must be enclosed by quotes. If it contains a quote character, two adjacent quotes must be entered for each desired quote. To INSERT a string that contains the variable delimiter character (the percent sign, %), it is necessary to code two adjacent variable delimiter characters to avoid variable-substitution or the start of a comment. The default string character can be set by the ASSUME STRING command (see 5.1.8.).

- L – Line Number Range Parameter.

  This parameter defines a line number range. Any line in f within this range is included in the INSERT operation. Any of the standard formats given in 2.2.8.2 is permissible. If omitted, the current line is the only line used.

- (c1,c2) – Column Specification.

  This parameter is used a number of ways depending on the value of the k parameter. If k is not omitted entirely, then this parameter must provide enough room for the specified string. If k is omitted, the value of c2 is not important, except that it must be at least as large as c1.

  If this parameter is omitted, the default value, normally (1,132) unless an ASSUME ECOLUMN command (see 2.2.5) is in effect, is used. Similarily, if either c1 or c2 is omitted, the corresponding portion of the default value is used.

  If this parameter is (1,132), and k is specified, the INSERT command deletes the contents of the line before performing the insertion. Both conditions must pertain. After deleting the contents of the line, the insertion is made according to the k specification, using (1,132) for the column parameter.

- k – Positioning Parameter.

  The interpretation of this parameter gives flexibility to the INSERT command. Six insertions are available. In three of these, a fill character is also defined by the k parameter. The following table shows the behavior of the INSERT command for each of the options:

Value in k	Behavior
null	1. Create space for the string by right–shifting the characters in and beyond column c1. The number of positions shifted is determined by the length of the string. The value of c2 is not used. 2. Insert the string in the columns vacated (starting at column c1). Note that no characters of the original string are deleted.
Ca	1. Delete columns c1 through c2 of the line.

2. Position the string in the center of the deleted area.

3. Substitute the character a (which may be a blank) in those columns of the line from c1 to c2 which do not contain the string.

RJa — As with Ca, except in step 2, position the string at the right edge of the deleted area.

LJa — As with Ca, except in step 2, position the string at the left edge of the deleted area.

PACK (or P)

1. Delete columns c1 through c2 of the line.

2. Position the string at the left edge of the deleted area.

3. Pack the line by left-shifting the columns beyond column c2 until the character in column (c2+1) is adjacent to the rightmost character of the inserted string.

W

1. Erase the whole line.

2. Insert the string starting at column c1. The value of c2 is not used.

The fill character, a, is the character immediately following the positioning code. It may be any character, including a blank.

The following example illustrates the difference between the various k options:

```
->PRINT A
100 ABCDEFGHIJKLMNOPQRSTUVWXYZ
110 ABCDEFGHIJKLMNOPQRSTUVWXYZ
120 ABCDEFGHIJKLMNOPQRSTUVWXYZ
130 ABCDEFGHIJKLMNOPQRSTUVWXYZ
140 ABCDEFGHIJKLMNOPQRSTUVWXYZ
150 ABCDEFGHIJKLMNOPQRSTUVWXYZ
END OF FILE
->I ***** 100 (5,15)
100 ABCD*****EFGHIJKLMNOPQRSTUVWXYZ
->I ***** 110 (5,15) C+
110 ABCD+++*****+++PQRSTUVWXYZ
->I ***** 120 (5,15) RJ+
120 ABCD++++++*****PQRSTUVWXYZ
->I ***** 130 (5,15) LJ+
130 ABCD*****++++++PQRSTUVWXYZ
->I ***** 140 (5,15) P
140 ABCD*****PQRSTUVWXYZ
->I */*** 150  (5,15)W
150     ******
->
```

Six INSERT commands are executed, each with a string of five asterisks, each with the column parameter of (5,15) which is an 11-character field, and each with a different type of k parameter. Where applicable, the plus sign (+) is used as the fill character. The last example demonstrates that the column limits are not used in determining which portion of the line to erase.

The next example illustrates what happens when a line is replaced:

```
->ASSUME PRINTWIDTH 36
->PRINT 100
100 ABCDEFGHIJKLMNOPQRSTUVWXYZ
->I  123456  100  CT
100 TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT12345
6TTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTTT
TTTTTTTTTTTTTTTTTTTTTTTTTTTT
->
```

The ASSUME PRINTWIDTH command (see 4.3.1) limits the number of characters printed per line to 36 (ASCII OFF mode). The INSERT command has deleted the contents of the present line and then performed the insert. It did this because no column parameter was coded and the default was (1,132). The INSERT created a 132 character string which took four lines to display, since only 36 characters per line were permitted. This example shows what the result would have been if it had been coded RJT or LJT. For either an empty k parameter, a W, or a PACK, the resulting line would have been:

```
100 123456
```

because when the insertion was made, the line was blank.

An attempt to specify a string too long for the column parameters (except for null k or W parameters), will display a diagnostic:

```
->I ******  100  (5,8)  C
<24> STRING EXCEEDS COLUMN LIMITS
->
```

The line is not changed.

5.3. Manipulation in f

The previous paragraph described commands which generally manipulate characters within working area lines. This section covers a number of commands which initialize f or move information from one area of f to another.

5.3.1. Erasing and Naming f – NEW

Syntax: NEW [d]

Abbreviation: None

Function: Discards the contents of the working area, f, and substitutes the parameter d as the name of the new contents (not yet entered).

The NEW command sets f to begin development of a new program or data set. It discards the contents of f, deletes the name, and substitutes the name given in parameter d. If the parameter, d, is omitted, CTS will solicit it. The parameter may be an explicit file name, an explicit element name, or a name which could be either. Acceptable forms include file or element, a file name, or an element name which includes a file name. If the name is not acceptable, a diagnostic message is displayed. If d is accepted, it becomes the name of f and the contents of f are discarded.

The following example illustrates some of these points:

```
->NEW
NEW PROGRAM NAME?  >###
<23> ILLEGAL FILE OR PROGRAM NAME SYNTAX ###
->NEW A
->
```

The working area, f, is now empty, and the name of f is now A. This could be either a file name or an element name, and CTS will interpret it either way depending on whether it is in DATA mode or ELEMENT mode at the time the name is used.

5.3.2. Reorganizing Line Numbers – RESEQUENCE

Syntax: RESEQUENCE [i [,j] [L]]

Abbreviation: RES

Function: To systematically renumber a contiguous set of lines in f.

The RESEQUENCE command takes the portion of f the line numbers of which fall into the range defined by L, and creates a new line number for each line in turn, according to the line number sequence parameter i, j. The first line in the range is given the line number i; the second, i+j; the third i+2j; and so on. Note that if L is one of the specifications which implies a backwards sequence, the lines will be in reverse order at the conclusion of the command. Both parts of the line number sequence parameter must be nonnegative integers. Hence, the sequence will always be ascending. L may be any of the forms of 2.2.8.2. If L is omitted, A is assumed for this parameter. If j is omitted, an increment of 10 is used. If i is omitted (allowed only if the entire parameter field is blank) 100 is taken as a default.

When resequencing part of f, it is possible to generate line numbers which are already represented in the nonresequenced part. When this happens, CTS resolves the line number conflict in one of two ways – the resequence method or the delete method. The choice is controlled by an ASSUME RESEQUENCE command. The default, if no such command has been executed, is the resequence method. See 5.3.6 for details of the two methods of resolving line number conflicts.

The following example illustrates some of the points discussed:

```
->PRINT A
100 LINE 1
110 LINE 2
120 LINE 3
END OF FILE
->RES  200,50   100,120
->PRINT A
200 LINE 1
250 LINE 2
300 LINE 3
END OF FILE
->RES  100  300-
->PRINT A
100 LINE 3
110 LINE 2
120 LINE 1
```

```
END OF FILE
->RES  200  200,100
->PRINT A
200 LINE 1
210 LINE 2
220 LINE 3
->RES
->PRINT A
100 LINE 1
110 LINE 2
120 LINE 3
END OF FILE
->RES  200,5  110+
->PRINT A
100 LINE 1
200 LINE 2
205 LINE 2
END OF FILE
->
```

Note especially the behavior for descending L specifications, the resequencing of part of f (the last RES command), and the RES with all parameters missing. This latter was equivalent to:

```
->RES  100,10  A
```

5.3.3. Nondestructive Line Copy – DITTO

Syntax: DITTO L i[,j]

Abbreviation: DIT

Function: To reproduce in f with different line numbers, a contiguous group of lines which already exist in f, leaving the old lines undisturbed.

The DITTO command duplicates a set of lines, giving them new line numbers. In effect, the lines are copied elsewhere in f, but the original lines are left undisturbed. The parameter L defines a range of line numbers. Any lines of f with line numbers within this range will be operated upon by the command. L also determines whether the lines within this range are to be referenced in order of ascending or descending original line numbers. Any of the forms of L given in 2.2.8.2 may be used.

The second parameter defines the sequence of line numbers to be assigned to the new lines. The first line will be assigned line number i; the second, i+j; the third, i+2j; and so on. If j is omitted, 10 is the default.

It is possible, by this process to create lines whose line numbers conflict with lines already represented in f. CTS handles these conflicts in one of two ways, the resequence method and the delete method. These are discussed in 5.3.6. The choice between the two methods is controlled by the ASSUME RESEQUENCE command. The resequence method is used before the first such command is executed.

The DITTO command is especially useful in cases where successive sets of data are to be created which differ from each other only slightly. The DITTO command creates a new set which is then edited with commands such as INSERT (see 5.2.4), INLINE (see 5.2.3), or CHANGE (see 5.2.2).

For example:

```
->N
100 >SET 1
110 >17.5  27.3  250
120 >10    17    23
130 >*DITTO  100,120  130,10
->I  2  130  (5,15)  LJ
130 SET 2
->I  15  150  (1,2)  C
150  15  17  23
->PRINT A
100 SET 1
110 17.5  27.3  250
120  10    17    23
130  SET 2
140  17.5  27.3  250
150  15    17    23
END OF FILE
->
```

If L or i is omitted, a diagnostic is displayed and the command is not executed. If j is omitted, a value of 10 is supplied as the default.

To illustrate:

```
->P  A
100 LINE 1
110 LINE 2
120 LINE 3
END OF FILE
->DIT  100,120
<22> REQUIRED SYNTAX IS MISSING
->DIT  100,120  200
->p  A
100 LINE 1
110 LINE 2
120 LINE 3
200 LINE 1
210 LINE 2
220 LINE 3
END OF FILE
->
```

5.3.4. Destructive Line Copy – MOVE

Syntax: MOVE L i[,j]

Abbreviation: M

Function: To move lines in f by assigning them new line numbers and discarding the old ones.

The MOVE command is identical to the DITTO command (see 5.3.3) except that it deletes the old lines from f. The function is similar to the RESEQUENCE command (see 5.3.2), although the order of the parameters is reversed and the defaults are different. See the DITTO command for details of operation

and parameter interpretation.

5.3.5. Changing the Name of f – RENAME

Syntax: RENAME [d]

Abbreviation: REN

Function: To change the name of f, without changing the contents.

The name d must be a legal name. CTS checks the syntax of this parameter, and if it finds a mistake, aborts the operation with a suitable diagnostic message.

For example:

```
->RENAME  &/*@
<23> ILLEGAL FILE OR PROGRAM NAME SYNTAX &/*@
->
```

No check is made to see if a file exists – only the syntax is checked.

A RENAME command with no d parameter specified clears the name of f. As when CTS is first initiated, f is not named after such a command:

```
->REN
->T DKN( )

->
```

The working area, f, is now unnamed, as indicated by the blank line printed as the value of the DKN( ) function (see 9.1.3).

RENAME never affects the contents of f, only the name.

5.3.6. Resolving Line Number Conflicts – ASSUME RESEQUENCE

Syntax: ASSUME RESEQUENCE [k]

Abbreviation: A RES

Function: To specify whether working area lines are to be resequenced when a line number conflict occurs.

The MERGE, GENERATE, MOVE, DITTO, and RESEQUENCE commands either add new lines of data to the contents of f or modify the line numbers of a part of f. Depending on the values of parameters, it is possible for these instructions to create a line which has a line number identical or greater than the one which already exists in f. Since it is not possible to have two lines in f with the same line number and not always desirable to break up the sequence of lines being edited, the conflict must be resolved. A line number conflict occurs whenever one of these commands is editing lines into a specified position in f (i.e., between two existing lines) and the new line number is greater than or equal to the next working area line number. Two methods of resolution are provided by CTS, the RESEQUENCE method and the DELETE method. The ASSUME RESEQUENCE command conditions CTS to use the method indicated by the parameter k. This method will be used until another ASSUME RESEQUENCE command changes it.

Select the RESEQUENCE method by using the parameter k as ON. If the parameter k is omitted, or contains a string other than OFF, CTS assumes the RESEQUENCE method as a default. The default mode for CTS is RESEQUENCE ON.

In the ASSUME RESEQUENCE ON mode the lines being edited into the working area are handled as a continuous block of lines. If this block of lines does not fit into the specified position in the working area due to a line number conflict, the working area lines greater than or equal to the conflicting lines are pushed down. They are pushed down by resequencing their line numbers beginning with the highest line number in the inserted lines plus one.

Each successive line in the working area is resequenced using an increment of one until the new line number of the last resequenced line is less than the line number of the next working area line. A warning message is printed to indicate that an auto resequence has occurred and to indicate which lines were affected. The message is:

<140> WARNING - AUTO RESEQUENCE THROUGH THE LAST LINE

Select the DELETE method by using the parameter k as OFF. In the DELETE method the old line is discarded and replaced with the new one.

In ASSUME RESEQUENCE OFF mode the lines being edited into the working area do not necessarily remain as a continuous block of lines. They may become interleaved with the working area lines. If an edited line has the same line number as a working area line, the working area line is replaced by the edited line.

See 3.6.1 and 3.6.2 for some examples of the two methods of line number conflict resolution. The response to a correctly specified command is the solicitation sequence:

```
->ASSUME RESEQUENCE OFF
->
```

6. Execution and Creation of Object Programs

6.1. General

When the lines of code which constitute a program have been created in f (expressed symbolically in a programming language), the program may be executed. Section 2 discusses the execution of programs as part of a sequence of steps for creating a working, error-free program, and shows that CTS has many features which make this creation process — including the execution phases — even more convenient.

CTS can be used for creating and executing complex programs by users with a wide range of experience. The CTS commands used to execute programs are very flexible, allowing the omission of parameters to provide flexibility. CTS automatically provides defaults for the missing parameters, making it easier for the novice to use the system.

6.1.1. Methods Used

There are three methods of executing a program expressed in symbolic code. An interpreter does not create a machine language program counterpart of the source code, but performs the execution by operating directly on the source code (or a reformatted representation of it). The interpreter, therefore, treats each source language statement as a call to a subroutine, with parts of the statement treated as parameters. APL is implemented as an interpreter.

In another method, sometimes called "compile-and-go," a compiler analyzes the source code, producing machine language instructions to perform the operations described by the source code. The resulting program is immediately executed. BASIC is implemented in this way. ASCII FORTRAN and COBOL can also be executed in this fashion via the ASSUME CHECKOUT ON command (see 11.2.4.2).

In the third method, a compiler produces machine language instructions, but in an intermediate form not directly executable. The intermediate form is called relocatable. The program produced may not be complete. It may be only a part of a large, complex program – a subroutine, for example. Executing such a program requires an additional step – bringing together all of the relocatable parts to produce an executable program. This process is called collection or mapping. This method normally gives a greater flexibility in the creation of the program. It may be subdivided into smaller parts and each part may even be coded in a different source language. If the program is to operate in well defined phases, segments may overlay each other in storage, to avoid excessive use of storage for parts of the program which are no longer needed or which will not be needed until a later part of the execution. This third method, involving the use of relocatable elements, is the most common method used. FORTRAN, COBOL, ALGOL, and MASM all produce relocatable elements.

6.1.2. Operating System Aspects of Compilation, Collection, and Execution

The compilation, collection, and execution processes make extensive use of program files (see 7.1.1). A program file may contain three primary types of elements: symbolic, relocatable, and absolute. It also contains a table of contents with the name and other essential information about each element in the file. The compilation process in the Series 1100 Operating System involves taking symbolic code, either from the run stream or from a symbolic element of a program file, possibly applying corrections to it, and producing a relocatable element and, optionally, an updated symbolic element. All three elements involved can be in the same file or different files and can have the same or different names. Except for special cases, only one relocatable element is produced per compilation. If a program consists of more than one part, a compilation for each part is required. The parts do not necessarily have to be in the same language. A single program may contain parts written in ALGOL, FORTRAN, COBOL, and assembly language, although so extreme a case is not common. Programs with two different languages are often encountered, however.

When all parts of the program are in relocatable form, an executable form of the entire program can be created by the process of collection. The Collector (MAP processor) performs this function. Directives to the MAP processor should come from a symbolic element of a program file, but may follow in the run stream. Normally the directives indicate what relocatable element in what file is the main program and what program files, if any, besides the standard system file are to be used as libraries (where needed relocatable elements may be found). The MAP processor selects additional relocatable elements (i.e., collects them) by matching undefined labels in the elements already collected to entry points with the same name in relocatable elements in the program files used as libraries. This process continues until no unmatched undefined labels remain. The MAP processor then fits the elements together, modifying addresses as needed, to produce a single absolute element in a program file. This element may be loaded and executed without further changes. Although it is an absolute element, the addresses are relative to the beginning of the element. By using relocation registers, the operating system loads the element into, and executes it from, any part of storage available. Execution is implemented with a single Executive control statement, which specifies the absolute element to be executed.

To illustrate this process, assume program file PA contains symbolic elements SUB1, SUB2, SUB3, and PROGA, along with elements from other programs. PROGA is a main program written in FORTRAN; SUB1 and SUB2 are subroutines written in FORTRAN; and SUB3 is a subroutine written in assembly language. The following portion of a run stream would compile the elements and execute the program:

```
@FOR,S     PA.PROGA, PA.A
@FOR,S     PA.SUB1, PA.SUB1
@FOR,S     PA.SUB2, PA.SUB2
@MASM,L     PA.SUB3, PA.SUB3
@PREP      PA.
@MAP,IS    ,PA.A
LIB        PA.
IN         PA.A
@XQT       PA.A
```

The first three control statements compile the three symbolic FORTRAN elements. The first compiles the element PROGA and produces in file PA the relocatable element A. The other two produce in PA relocatable elements of the same name as the symbolics from which they were compiled. The fourth control statement calls the assembler which creates the relocatable element SUB3 in PA from the symbolic element SUB3. The fifth control statement prepares the file PA so it may be used as a library. Essentially, it creates a table of entry points and places this table in the file. This must be done before using the file as a library if any pertinent relocatable elements have been added, deleted, or changed since the last PREP.

The next statement calls the MAP processor to collect the program and produce an absolute element in PA with the name A. The directive which follows tells the processor that file PA is to be used as a library. Relocatable elements with entry points matching undefined labels are taken from this file in preference to the system library. The second directive specifies that relocatable element A in file PA is to be included in the collection. The MAP processor thus starts with only this element. It then introduces other relocatable elements which match its undefined labels (and any new undefined labels brought in with the new elements) until no more unmatched, undefined labels exist. The absolute element is then created from the collected elements.

The final control statement tells the operating system to execute the absolute element A in file PA.

Each of the processor call control statements has a string of characters in the subfield immediately following the name of the processor. These are options which the processor uses to govern its activity. Each character is an option. A number of options have standard meanings to most of the processors. For more information, refer to SPERRY UNIVAC Series 1100 Executive System, Volume 2 EXEC, Programmer Reference, UP-4144.2 (current version). Study both the language and the compiler operation for each compiler used, especially if the special features and options are used. The appropriate programmer reference manual for each compiler contains this information.

6.1.3. Compilation, Collection, and Execution Under CTS

CTS accepts a command which describes in a general way what is to be done. This usually leads to the setup by CTS of a number of operations for the operating system to perform. When the entire process has been set up, CTS directs the operating system to perform these operations. This saves effort and reduces the possibility of mechanical errors.

Since it is designed with the interactive user in mind, CTS avoids operations which are inefficient. For example, many of the processors (or compilers) have been written to operate efficiently under batch mode. This mode is normally used for runs submitted via the onsite card reader. Obviously, no interaction is possible in this mode. Such compilers normally produce an output listing at the end of the compilation. If such a compiler is used from a terminal, the volume and speed of this output is excessive and it either passes by so quickly that it is hard to follow or it ties up the terminal for a long time. CTS solves this problem automatically by directing such output to a special file. The SCAN command (see 11.1.1) examines any part of the output.

In addition, some compilers are not reentrant and often use large amounts of high-speed storage, a valuable system resource. Each user of a nonreentrant compiler has a private copy of the compiler. It is desirable to minimize the time the compiler is in high-speed storage. If the novice using demand mode calls such a compiler and enters the program a line at a time, a great deal of waste is incurred. The efficient way to operate — even in demand mode — is to create the entire program first, and then call the compiler to compile it all at once. CTS forces this situation. The programs are first created in f and then compiled all at once.

A prescan module is even more efficient, since it finds most errors before the program is actually compiled. Syntax checking is only a relatively small part of the compilation process, and prescan modules are reentrant, so it is more efficient to find an error in this way than by compilation.

Though CTS is particularly useful for the novice, avoiding pitfalls leading to inefficiencies and minimizing the knowledge needed to run a program, CTS is also useful to the expert. The convenience of the CTS interface, the prescan module, and the comprehensive editing features lighten work loads considerably. The CTS commands dealing with the compiling, mapping, and execution of programs are sufficiently comprehensive to permit the construction and execution of large, complex programs.

6.2. Compiling, Collecting, and Executing in One Operation – RUN

Syntax: RUN [*] [(C1 [, P [, E]])] [d1 [,d2...]] [(C2...)]

Abbreviation: R

Function: To produce and execute an absolute element from one or more symbolic elements.

The operation of the RUN command varies from the very simple to the very complex. Taking the simplest case first, the simple command:

```
->RUN
```

will cause the compilation, mapping, and execution of the symbolic program in f, using the assumed compiler (see 6.2.1). The following example illustrates this situation:

```
->OLD TRI
->LIS
100 10      FORMAT()
110 1       READ(5,10) A,B
120         IF (A .LT. 0) GO TO 2
130         C = SQRT (A**2 + B**2)
140         WRITE (6,10) A,B,C
150         GO TO 1
160 2       END
END OF FILE
->RUN
COMPILING...
>3,4
     3.0000     4.0000     5.0000
>-1,1
NORMAL EXIT EXECUTION TIME:              20 MILLISECONDS
*DIAGNOSTIC SCAN? >Y
1  @RFOR,RS TPF$.NAME$
2  RFOR   5.1   01/25-10:09-(0)
23 END RFOR
25 @PREP TPF$.
27 @MAP,S ,TPF$.NAME$
33
34 ADDRESS LIMITS:      001000 016540       7009 IBANK WORDS DECIMAL
35                       040000 047433       3868 DBANK WORDS DECIMAL
*END DIAGNOSTIC SCAN
->
```

CTS implements this simple RUN command by first creating symbolic element NAME$ in the object file (see 7.1.2) which is a copy of the program in f. CTS then creates a partial run stream and submits the partial run stream to the Executive for implementation. The steps in this partial run stream are:

1. Redirect all output to the scan file SQUELCH$ instead of going to the terminal.

2. Compile the symbolic element NAME$ in the object file using the assumed compiler, producing in the object file a relocatable element with the name specified on an ASSUME RELOCATABLE command (if any) or the name NAME$.

3. Prepare the object file to be used as a library (@PREP control statement). The object file could contain relocatable elements from previous compilations to be used as subroutines. If so, this step is necessary. If not, it is superfluous.

4. Collect the program, using the MAP processor. The relocatable element, normally NAME$, from the object file is used as the main program. The object file is used as a library. The absolute (executable) element produced in the object file has the name specified on an ASSUME XQT command (if any), or is called NAME$.

5. Redirect subsequent output to the terminal. Output sent to the scan file was, therefore, the output from the compilation, the @PREP statement, and the output from the collection (including associated control statements).

6. Execute the element, NAME$ from the object file. Note that output is sent to the terminal. Also, since this control statement is the last image in the add file, any input to the executing program is solicited from the terminal.

From the time the partial run stream was submitted to the Executive to the termination of the executing program, CTS was not in control. CTS gained control when the program terminated and displayed the message:

```
*DIAGNOSTIC SCAN?>
```

Answering this message with *Y* caused a display of the essential elements in the scan file, all pertinent control statements, and selected output lines from the compilation and collection.

In this simple case, the object file is left with three new elements, all called NAME$. One is symbolic, one relocatable, and one absolute. Elements previously in the file are still there unless they had the same name and type as one of the new elements, in which case the new elements replaced them.

The more complicated RUN commands may be explained as variations on the basic sequence given above. Consider a more complicated case where the above program requires two subroutines and a function. The main program is TRI1 and the names of the elements in the save file are SUB1, SUB2, and FUNC1. They are created in the following example:

```
->FOR FIELDATA
FD FORTRAN 5R1
>>NEW TRI1
>>N
100 >1 CALL IN(A,B)
110 >IF (A .LT. 0) GO TO 2
120 > C = SQRT (SUMSQ(A,B))
130 > CALL OUT (A,B,C)
140 >GO TO 1
150 >2 END
160 >*LIS
100 1      CALL IN(A,B)
110        IF (A .LT. 0) GO TO 2
120        C = SQRT(SUMSQ(A,B))
130        CALL OUT(A,B,C)
140        GO TO 1
150 2      END
END OF FILE
>>SAV
DO YOU WANT A GLOBAL SCAN? >Y
```

```
>>NEW  SUB1
>>N
100 >S:E IN (X1, X2)
110 >1 FORMAT( )
120 >READ (5, 1) X1, X2
130 >RETURN
140 >END
150 >*LIS
100         SUBROUTINE IN (X1, X2)
110 1       FORMAT ( )
120         READ (5,1) X1, X2
120         RETURN
140         END
END OF FILE
>>SAV
DO YOU WANT A GLOBAL SCAN? >Y
>>NEW SUB2
>>N
100 >S:E OUT(X1, X2, X3)
110 >1 FORMAT( )
120 >WRITE (6,1) X1, X2, X3
130 >RETURN
140 >END
150 >*P A
100         SUBROUTINE OUT(X1,X2,X3)
110   1     FORMAT( )
120         WRITE(6,1) X1,X2,X3
130         RETURN
140         END
END OF FILE
>>SAV
DO YOU WANT A GLOBAL SCAN? >Y
>>NEW FUNC1
>>N
100 >F:N SUMSQ(X1,X2)
110 >SUMSQ = X1**2 + X2**2
120 >RETURN
130 >END
140 >*LIS
100         FUNCTION SUMSQ(X1,X2)
110         SUMSQ = X1**2 + X2**2
120         RETURN
130         END
END OF FILE
>>SAV
DO YOU WANT A GLOBAL SCAN? >Y
>>LIS S
RUNTST
TYPE     NAME
FOR    FUNC1
FOR    SUB2
FOR    SUB1
FOR    TRI1
FOR    TRI
>>
```

There are now five elements in the save file, RUNTST. The original, completely self-contained program, TRI (from the previous example), and the new TRI1, accomplish the same thing, but TRI1 calls on two subroutines and a function to do it. These are elements SUB1, SUB2, and FUNC1. They have entry points, IN, OUT, and SUMSQ, respectively. These entry points are undefined labels in the main program, TRI1. For example, the statement in TRI1:

```
CALL IN(A,B)
```

creates the undefined label, IN. This is matched during collection with the entry point IN, which is in element SUB1. This should help to clarify the relationship between an element name, an undefined label, and an entry point.

It is not essential to follow the logic of these program elements. What is important is that TRI1 needs SUB1, SUB2, and FUNC1 to be a complete program, and collection is effected by first selecting TRI1 and matching its undefined labels with entry points into other relocatable elements (which are SUB1, SUB2, and FUNC1).

Continuing the example, the new program is now compiled, collected and executed:

```
>>OLD TRI1
>>RUN SUB1, SUB2, FUNC1
DO YOU WANT A GLOBAL SCAN? >Y
COMPILING...
>1,1
     1.0000     1.0000     1.4142
>6,8
     6.0000     8.0000     10.0000
>-1,1
NORMAL EXIT EXECUTION TIME:              23 MILLISECONDS
DIAGNOSTIC SCAN?  >N
>>
```

The first command brought TRI1 into f. This is important, because when f is not empty, the RUN command causes CTS to assume that this is the main program unless an ASSUME MAIN has been done. As such, it is used to start the process of matching undefined labels during collection and contains the starting address for the program. If the collector does not start with the main program, the generated absolute element normally will be in error. Thus, for more complex RUN commands it is important not only to compile each part with the appropriate compiler, but also to arrange for CTS to use the correct element as the main program. There are three conditions to consider:

1. If f is not empty, its contents form a relocatable element in the object file with the ASSUME RELOCATABLE name or the default name, NAME$. This relocatable element is selected as the main program during collection.

2. If f is empty, the elements explicitly stated on the RUN command are compiled one at a time, starting with the leftmost and taking each in turn. Each compilation produces a relocatable element in the object file with the same element name as the symbolic element from which it is created. The last relocatable element created (that is, the relocatable element created from the rightmost symbolic element on the RUN command) is used as the main program in this case.

3. The ASSUME MAIN command establishes the name of the relocatable element which is assumed to be the main program. This overrides selection based on the criteria mentioned above. More information on using ASSUME MAIN is given in 6.4.2.1.

The parameters of the RUN command have the following significance:

- The asterisk immediately following the word RUN directs CTS to use the parameters from the most recent RUN command as if they had been coded on the present command. No other parameters should be coded on a RUN command if the asterisk is used. Be sure that conditions are compatible. A common error is to use f to fix an error and forget to restore it to a state consistent with the RUN* command.

- The other parameters consist of a compiler parameter and a list of symbolic elements. The compiler parameter has the form:

  C,P [, E]

  where C is the name of the compiler (as registered in the operating system) and P is a string of letters (options) which the compiler uses to modify its behavior. The E parameter represents a string of extra options to be used by the ASCII COBOL compiler only. If there are no options, the comma is omitted.

- The compiler parameter may be omitted, in which case the assumed compiler is used. If there is no assumed compiler, CTS solicits one.

- The list of symbolic elements are the elements which the specified or assumed compiler is to compile. As part of its designation an element may specify the name of the file containing it. If the file name is missing, it is assumed to be F, the save file.

An example of a RUN command with three clusters is:

```
RUN   (FOR)   A,B,C   (MASM,S)   X1.D   (FOR)   E
```

If f is empty, this results in compilation by the FORTRAN compiler (batch version) of symbolic elements A, B, and C which are located in F. Then the Assembler would assemble element D in file X1. Finally, the FORTRAN compiler would compile element E in F. In each case, the relocatable elements would go into the object file. Unless an ASSUME MAIN has been done, element E will be used as the main program.

If f was not empty, its contents would have been transferred to the object file as a symbolic element called NAME$. The first compilation would then have been the FORTRAN compiler compiling this element (even if the assumed compiler were different). The remaining compilations would follow as described above. The compiler of the first cluster, then, whether assumed or explicit, applies to the contents of f, if any, as well as to the list of elements in the first cluster.

After the compilations, mapping and execution are performed as described previously.

As a final example, an assembler version of the function in the previous example is created and the program is compiled and executed as before, but using this new version of the function. ASSUME MAIN is also used.

```
->NEW FUNC2
->N
100 >    AXR$ .
110 >SUMSQ* .
120 >            L          A0,*0,X11   .
130 >            FM         A0,A0       .
140 >            L          A1,*1,X11   .
150 >            FM         A1,A1       .
```

```
160 >           FA          A0,A1       .
170 >           J           3,X11       .
180 >*SAV
->NEW A
->A MAIN TRI1
->RUN (FOR,S) TRI1, SUB1, SUB2 (ASM,S) FUNC2
MAIN PROG: TRI1
COMPILING...
>3,4,
     3.0000     4.0000     5.0000
>-1,0
*DIAGNOSTIC SCAN? >N
->
```

Because the ASSUME MAIN command specified TRI1 as the main program, the normal rule for determining the main program (which would have selected FUNC2) was overridden. Also, CTS announced that this was being done by displaying a message.

It is sometimes useful to know how a RUN command changes the contents of the object file. If f is not empty, a symbolic element called NAME$ is created and placed into the object file. This element is then compiled, producing in the object file a relocatable element NAME$ or the ASSUMED RELOCATABLE name. If f is empty, these two elements are not produced. Each additional compilation produces in the object file a relocatable element which has the same name as the symbolic element from which it was created. The collection process creates an absolute element called NAME$ in the object file unless an ASSUME XQT has been done.

If the object file was not empty when the RUN command was submitted, any elements created as a result of the RUN command replace corresponding elements (same name and type) which are already there. The original elements of the object file not replaced in this way remain in the file. This can lead to errors or to unexpected results. For example, the object file can wind up with more than one relocatable element (different element names) with the same entry point. This would cause confusion during collection.

Some of the ASSUME commands affect the running of programs.

6.2.1. Setting the Assumed Compiler – ASSUME COMPILER

Syntax: ASSUME COMPILER [C [, P [, E]]]

Abbreviation: A COM

Function: To establish an explicit compiler and compile options as the assumed compiler associated with the working area, f.

The assumed compiler is the name (as it would appear on the Executive control statement) of the compiler (or other processor) which will be used to process the information in f when a COMPILE (see 6.4.1) or RUN (see 6.2) command is encountered which does not explicitly have a compiler name in the first parameter position. CTS sets the assumed compiler automatically, but occasionally it has no basis on which to make a decision and uses ELT,L. The ASSUME COMPILER command permits replacing the present value with the needed one.

In coding the parameters, C is the name of the processor (compiler) as recognized by the operating system. P is a string of options to be applied during compilation. The E option represents a string of extra options to be used by the ASCII COBOL compiler only. These options are compiler dependent, although most compilers have common definitions for some options. If P is not specified, the options used are those shown in the list of assumed compilers in 3.5.

NOTE:

Using this command requires an understanding of the processor call statement of the operating system (see the SPERRY UNIVAC Series 1100 Executive System, Volume 2 EXEC, Programmer Reference, UP-4144.2 (current version)).

Any of the prescan modules reset the assumed compiler to be compatible with the programs they are prepared to handle.

6.2.2. Changing the Save and Object File – ASSUME FILE

Syntax: ASSUME FILE [FN]

Abbreviation: A F

Function: To direct CTS to use the file with the name specified in parameter FN as both the save file, F, and the object file (see 7.1).

The ASSUME FILE command defines any existing program file (or empty file) specified by FN to be the save file (F), exactly as the ASSUME PROGRAM command (see 6.2.4) does. It also defines this same file to be the object file, just as the ASSUME OBJECT command (see 6.2.3) does. It has the same effect as if these two commands were issued separately (with the same FN, of course).

The file specified by FN must exist. If it is cataloged but not assigned to the run, CTS will assign it.

An ASSUME FILE command with no FN parameter will cause both the object and save files to be the standard CTS file used for the save file. In other words, it sets the save file, F, back to its standard file; but the object file is also set to this file, which is not its standard file.

The ASSUME FILE command never affects the contents or existence of the files concerned, but merely conditions the system to change which files are used for F and the object file. No information is transferred or destroyed.

Any CTS command which uses either F or the object file as a default is affected by the changes which this command produce.

Examples of CTS responses connected with the use of an ASSUME FILE command follow.

- The normal response, when FN is correctly specified, is the solicitation character.

```
->ASSUME FILE PA
->
```

  The file, PA will now be treated as both the save file, F, and the object file.

- If the parameter FN is not present, it will be taken as the standard CTS-created save file, the name of which is the run-id of the present run.

```
->A FILE
->
```

  If the run-id of this run is RUNA, then the file named RUNA will be used for both F and the object file.

- If the file designated by FN does not exist, a diagnostic message is displayed and the existing assumed save and object file names are not changed.

```
->A FILE PB
<68> PB IS NOT CATALOGUED
->
```

- If the FN parameter is specified with incorrect syntax, a message is displayed and the existing save and object file names are not changed.

```
->A FILE &*-
<23> ILLEGAL FILE OR PROGRAM NAME SYNTAX &*-
->
```

6.2.3. Changing the Object File – ASSUME OBJECT

Syntax: ASSUME OBJECT [FN]

Abbreviation: A O

Function: To substitute the named file for the object file currently being used by CTS.

The object file is the file in which the COMPILE (see 6.4.1), MAP (see 6.4.2), and RUN (see 6.2) commands place the elements created by compilation and collection. It is also the default file for the XQT command (see 6.3.1).

The file name coded in parameter FN becomes the new object file. It must exist. If it exists, but has not been assigned, CTS will assign it. If the parameter FN is omitted, the standard object file (TPF$) is reestablished.

6.2.4. Changing the Save File – ASSUME PROGRAM

Syntax: ASSUME PROGRAM [FN]

Abbreviation: A PRO

Function: To direct CTS to use the file with the name specified in parameter FN as the save file, F.

When CTS is initialized it checks to see if a file exists with the name of the run under which it is being initialized (see 7.1). If it exists, it is assigned to the run. If the file does not already exist, it is created, cataloged, and assigned to the run. This file is known as the save file, or F. It is a program file. It provides a standard repository for programs or data images which are to be used in another session (another run). This file is the only file maintained automatically by CTS which is cataloged and, therefore, is saved by the Executive when the run terminates normally.

A file with a name different from that of this standard F file may be used. A private file may be used for F during part of the session. A group project may use a common file to save programs. The ASSUME PROGRAM command defines any existing program file (or empty file) as the file to be used as F from this point on.

The file specified in the parameter must be an existing file. If it is cataloged but not assigned to the run, CTS will assign it.

No information is transferred and no files are released. All files remain as they were.

To return to the original CTS file for F, submit an ASSUME PROGRAM command with the FN parameter omitted.

If CTS detects an error in the file name parameter, the save file F is not changed.

Those CTS commands which assume F as a default are also affected by the ASSUME FILE command.

Some examples of CTS responses to the ASSUME PROGRAM command follow.

- The normal response is the solicitation character, indicating that the change has been successfully made.

```
->ASSUME PROGRAM PB
->
```

  PB is now the save file, F.

- Using the abbreviations correctly gives the same result.

```
->A  PRO  PA
->
```

- To reestablish the standard file for F, submit the command with a blank FN parameter.

```
->A  PRO
->
```

  CTS will now use the file it originally created for this purpose as the save file.

- If the file specified is a data file, it is accepted, but any attempt to use it will fail.

```
->A PROG DA
->SAV
<19> DA IS NOT A PROGRAM FILE
->
```

  The data file DA is now F, but it is not usable, because of its type. Another ASSUME PROGRAM command must be used before F can be used or the file DA can be erased.

- If the file specified is nonexistent, a diagnostic message is displayed and F will be unchanged.

```
->A PROG PB
<68> PB IS NOT CATALOGUED
->
```

- If the syntax of the file name is in error, CTS displays a message and F is not changed.

```
->A PRO > AB
<23> ILLEGAL FILE OR PROGRAM NAME SYNTAX > AB
->
```

6.2.5. Changing the Name of the Relocatable Element – ASSUME RELOCATABLE

Syntax: ASSUME RELOCATABLE [d]

Abbreviation: A REL

Function: To specify the name of the relocatable which is produced when the program in the working area is compiled by a COMPILE or RUN command.

The d specification can be a file name and element name. If only an element name is specified, the current assumed object file (usually TPF$) will be used. If only a file name is specified, NAME$ is used for the element name. If d is not specified, the object file and NAME$ are assumed.

NOTE:

If the assumed object file is changed after an ASSUME REL which did not specify a file name has been done, the old object file will be used.

This command allows the specification of unique names to save the relocatable elements produced by a COMPILE or RUN command. To use the program again, only a MAP is necessary. This also allows doing a series of compiles to produce a different relocatable element each time by changing the ASSUME REL name. These relocatables may then be combined to form one executable element by a MAP command. If the ASSUME REL is not changed, each COMPILE or RUN with a program in the working area will produce a relocatable by the same name which replaces the last relocatable by that name.

The ASSUME XQT (see 6.3.2.) command can be used in the same manner as the ASSUME REL command to specify a name for the executable element produced.

6.3. Executing, Naming, and Saving Absolute Elements

The discussion of the RUN command in 6.2 identified three kinds of operations:

1. Producing relocatable elements from symbolic elements – compilation.

2. Producing an absolute element from relocatable elements – collection.

3. Execution of an absolute element.

Although the RUN command performs all of these operations, each of them may be performed individually with the COMPILE (see 6.4.1), MAP (see 6.4.2), and XQT (see 6.3.1) commands, respectively. There are also a number of commands which affect one or more of these steps, whether performed individually or collectively. In particular, these are the COMPILER, FILE, LIBRARIES, MAIN, MAP, OBJECT, PROGRAM, XQT, and RELOCATABLE subcommands of the ASSUME command.

6.3.1. Executing an Absolute Element — XQT

Syntax: XQT [,P] [d]

Abbreviation: None

Function: To execute an absolute element.

The XQT command may be used to execute any absolute element. Normally, however, it is used to execute the absolute element which has just been created with a MAP (see 6.4.2) or RUN (see 6.2) command. To do this use the simplest form of the XQT command:

```
->XQT
```

Unless defaults are changed explicitly, CTS executes the absolute element NAME$ in file TPF$. If the object file has been changed with the ASSUME FILE (see 6.2.2) or ASSUME OBJECT commands (see 6.2.3), it is used instead of TPF$. If the ASSUME XQT command (see 6.3.2) has been used, the defaults of any of the parameters of the XQT command may be changed.

To execute a specific element in a specific file, simply code the XQT command as in :

```
->XQT  FA.EA
```

In this case, CTS executes absolute element EA in file FA.

Some programs are written to look for options on the @XQT operating system control statement which causes them to execute. These options are submitted as a string of letters, each letter being an option. The order of the letters is not significant. SPERRY UNIVAC Series 1100 Executive System, Volume 2 EXEC, Programmer Reference, UP-4144.2 (current version) contains more information. Any program compiled under RFOR, for example, responds to an S option by printing one extra line at the termination of the program with information about certain contingencies encountered during the execution of the program. Options may be specified by following the XQT immediately with a comma and the string of options.

For example:

```
->XQT,S   FA.EA
```

The most common diagnostic encountered using the XQT command is:

```
->XQT
PROGRAM NOT FOUND
->
```

It means that the program name, generated by default in this case, does not exist.

6.3.2. Naming the Absolute Element — ASSUME XQT

Syntax: ASSUME XQT [d] [,P]

Abbreviation: A XQT

Function: To establish a default file name and element name, d, to be used for the absolute element in the XQT, MAP, and RUN commands, and a default option string, P, to be used for the XQT and RUN commands.

Most commonly, the ASSUME XQT command is used for creating an absolute element which is to be saved, rather than used and discarded. For example, consider the following part of a CTS session which starts with f empty and all defaults standard:

```
->RUN (MASM,S) FUNC2 (FTN,S) SUB1, SUB2, TRI1
->XQT
>3,4
      3.0000     4.0000     5.0000
>-1,1
*DIAGNOSTIC SCAN? >N
->
```

A number of relocatable elements and the single absolute element NAME$ have been created. To save this absolute element, copy it to the file wanted and change its name. Had the RUN been preceded by:

```
->A XQT B.TRI
```

the absolute element created by the RUN command would have been called TRI and placed in file B. The object file, TPF$, would still have been used as before, with the single exception that it would not contain the absolute element.

The parameter d may be a file name only (e.g., B.), an element name only (e.g., TRI), or both (e.g., B.TRI). A RUN or MAP command supplies the missing parts with default values.

For example, if f is empty, all other defaults are standard, and we have the following sequence:

```
->A XQT TRI
->RUN (FOR,S) TRI
COMPILING...
```

the absolute element produced is element TRI in file TPF$. On the other hand, the sequence:

```
->A XQT B.
->RUN (FOR,S) TRI
COMPILING...
```

would have created an absolute element NAME$ in file B.

Omitting d reestablishes NAME$ as the name of the absolute element.

The parameter, P refers to the execution step of the RUN command and to the XQT command. It is a string of options for the program being executed (see 6.3.1). If P is empty, the comma associated with it can also be omitted. To cancel the effect of P from a previous ASSUME XQT command the comma is required. Since the MAP command has no execution step, the parameter P has no effect on its operation.

The RUN and MAP commands always use the defaults set up by the most recent ASSUME XQT command. The XQT command, however, only uses these defaults if all parameters of the XQT command are empty.

For example:

```
->A XQT AB.
->XQT Z
```

would result in executing element Z from file TPF$.

On the other hand,

```
->A XQT AB.
->XQT
```

would result in the execution of element NAME$ in file AB.

6.4. Creating Relocatable and Absolute Elements

Paragraph 6.3 pointed out the three steps of the RUN command: compilation, collection, and execution. It also showed how to perform the execution step separately with the XQT command. This paragraph shows how the compilation (COMPILE) and collection (MAP) may be performed independently.

6.4.1. Creating Relocatable Elements — COMPILE

Syntax: COMPILE [(C1 [, P [, E]])] [d1,d2,...,] [(C2...)]

Abbreviation: COM

Function: To compile symbolic elements and place the resulting relocatable elements produced into the object file.

The COMPILE command performs the same function as the first step of the RUN command (see 6.2). With the exception of the asterisk of the RUN command, the parameters have the same format and significance. If the working area is not empty, it creates a symbolic element called NAME$ in the object file. It then creates a partial run stream to perform the compilation. CTS then submits the add file to the operating system for implementation. Compiler output is directed to the scan file. At the termination of the last compilation, CTS regains control and displays the message:

```
*DIAGNOSTIC SCAN? >
```

The partial run stream created in the add file is identical to the run stream which would have been created by a RUN command up to the control statement which prepares the file to be a library (@PREP). In the case of the COMPILE command, this @PREP and all subsequent control statements are omitted.

Refer to 6.2 for more details on the format and significance of the parameters, or the implementation of this command, keeping in mind that for the COMPILE command the process stops with the compilations.

If a series of compilations is to produce different relocatable elements, an ASSUME RELOCATABLE command must be used before each COMPILE to generate unique names (see 6.2.5).

6.4.2. Creating an Absolute Element — MAP

Syntax: MAP

Abbreviation: None

Function: To collect relocatable elements and produce an absolute (i.e., executable) element.

The MAP command independently performs the second step of the RUN command (see 6.3). Unlike the RUN command, however, the MAP command has no parameters. In the absence of explicit controls set up by the relevant subcommands of the ASSUME command, the MAP command chooses its defaults by examining f and the object file.

To properly set up the collection process, the MAP command needs three types of information:

1. What program files (other than the system library) are to be used as libraries.

2. What element is to be used as the main element.

3. What name is to be given to the absolute element and into what file it is to be placed.

The object file is always taken as a library file. In addition, those files specified on the most recently executed ASSUME LIBRARIES command (see 6.4.2.2) are also used as libraries. All program files used as libraries (except the system library) are prepared to be libraries (with the @PREP Executive control statement).

The main program is selected in one of three ways. If an ASSUME MAIN command (see 6.4.2.1) has designated the main program, it is taken. A message indicating this fact is displayed during the execution of the MAP command. If no ASSUME MAIN command is in effect, the ASSUME RELOCATABLE name is taken, if there is one. Otherwise, the object file is searched for the presence of a relocatable element NAME$. If such an element exists, it is taken as the main program, otherwise, the relocatable element with the same name as the name of f in the object file is taken as the main program.

If the name of the absolute element or the file into which it is to be placed is specified on an ASSUME XQT command (see 6.3.2), the specified portions are taken. If the file is not specified the object file is used. If the name is not specified NAME$ is used.

As with the RUN, COMPILE, and XQT commands, CTS executes the MAP command by creating a partial run stream in the add file and then directing the operating system to use it.

The heart of the operations performed by this partial run stream is the collection itself, caused by "@MAP" (Executive control statement) followed by a series of directives for the Collector (MAP processor). These directives inform the MAP processor what libraries to use, what the main program is, etc. The ASSUME MAP command allows these directives to be specified.

A number of CTS commands affect the operation of the collection process as implemented in the RUN (see 6.2) and MAP (see 6.4.2) commands. Some of them affect the process in an obvious way, the ASSUME OBJECT command (see 6.2.3), for example. Others involve the collection process more closely. These are emphasized in the next few paragraphs.

6.4.2.1. Specifying the Main Program — ASSUME MAIN

Syntax: ASSUME MAIN [d]

Abbreviation: A MAI

Function: To specify the main program name for the RUN and MAP commands.

Both the RUN (see 6.2) and the MAP (6.4.2) commands involve the creation of an executable element by the process called collection (see 6.1.1 and 6.1.2). For collections where several relocatable elements are involved, it is important to start with the main element. This can be accomplished by thoroughly understanding the default mechanism which these commands use for selecting the main program and carefully arranging for this default to be the correct element. The default is the element produced by compiling the working area. The ASSUME MAIN command can specify the element to be used as the main program, avoiding the risk of selecting the wrong element. The parameter may refer to any available program file. The element need not exist at the time of the ASSUME MAIN command, but an error results if it does not exist when the collection for a RUN or MAP command occurs.

Omitting d restores the standard system behavior.

6.4.2.2. Specifying Additional Libraries — ASSUME LIBRARIES

Syntax: ASSUME LIBRARIES [F1 [,F2...]]

Abbreviation: A LIB

Function: To cause CTS to use the named program files, in the order given, when searching for a relocatable element during the collection process.

In the collection process an executable (absolute) element is created from a relocatable element designated as a main program by collecting from one or more libraries additional relocatable elements. The Executive, unless directed otherwise, uses the system library as the only such library. CTS always specifies to the Executive that the object file (usually TPF$) is also a library. The ASSUME LIBRARIES command permits the use of additional libraries as well. The parameters, Fn, must be existing, nonempty program files. If any are not, the entire command is rejected. Each ASSUME LIBRARIES command invalidates all previous ASSUME LIBRARIES commands.

The sequence in which the libraries will be searched is:

object file, F1, F2, ..., system library.

An ASSUME LIBRARIES command with no parameters eliminates from the process all special libraries, and only the object file and system library will be used following such a command.

The RUN (see 6.2) and MAP (6.4.2) are the only commands causing CTS to start the collection process. To use a file as a library, it must first be prepared. The @PREP Executive control statement performs this function. CTS prepares each library used (except the system library) each time a collection is performed, unless the file is read-only. Therefore, if possible, library files should be prepped read-only files.

The ASSUME LIBRARIES command is effective even when the ASSUME MAP command (see 6.4.2.3) is in effect.

6.4.2.3. Specifying MAP Directives — ASSUME MAP

Syntax: ASSUME MAP [d] [,P]

Abbreviation: A MAP

Function: To replace the standard MAP directives supplied by CTS with those in the specified element of a program file.

To replace the set of MAP directives which CTS normally creates to implement a collection, use the ASSUME MAP command. The RUN (see 6.2) and MAP (see 6.4.2) commands involve collection. They implement it by creating a partial run stream, part of which is a MAP processor call statement (@MAP) followed by suitable directives which depend on the nature of the collection. This partial run stream is then submitted to the Executive for execution.

When an ASSUME MAP command with a parameter d is submitted, a subsequent collection caused by either the RUN or MAP commands generates the usual set of control statements in the partial run stream. It generates @PREP control statements for the object file and for any additional libraries (see ASSUME LIBRARIES, see 6.4.2.2). It also generates the MAP processor call statement. The ASSUME XQT command (see 6.3.2) or, in its absence, the normal default, still governs the name and file of the resulting absolute element. However, the symbolic input field of this processor call statement now contains the element specified in the d parameter of the ASSUME MAP command. If no extra libraries are defined, no MAP directives are generated by CTS. If an ASSUME LIBRARIES command is in effect, CTS generates a single MAP directive defining the additional libraries. If specified, P is a list of options to place on the Collector call statement (@MAP).

An ASSUME MAP command with no d parameter disables the feature, and CTS again generates the normal MAP directives. If the P parameter is omitted, Collector options return to CTS-generated defaults.

Before using this feature, a working knowledge of the Collector (see the SPERRY UNIVAC Series 1100 Collector (MAP Processor), Programmer Reference, UP-8721 (current version)) and the process of collection as directed by CTS (see 6.1 and 6.2) are required.

6.5. Initiating a Processor Call — PXQT

Syntax: PXQT [,E] s

Abbreviation: PXQ

Function: To allow a processor call from within CTS.

The PXQT command permits a direct processor call from within CTS. The string s contains the processor call in Executive control statement format (minus the leading @). When the processor has finished execution, CTS is automatically reloaded.

Please note that should CTS be interrupted with an @@X CIO during the time that another processor is active, CTS will never know about it. The EXEC terminates the active processor and since the EXEC CTS reload bit is on, simply reloads CTS. Abnormal termination of a processor is also transparent to CTS.

The E option causes an @EOF to be submitted after the PXQT command. Certain processors may require an @EOF.

Example:

```
->CRE,T TAPE.,U9V,REEL1
*CRE,T TAPE,U9V,REEL1
->PXQT,E MOVE TAPE.,2
FURPUR 28R1     73R1    06/05/80 12:01:09
->PXQT,E COPY,S TPF$., TAPE.
->PXQT FILE.RUNTIME,S
      TODAY IS 6/5/80 at 12:01:43
->DATE
05 JUN 80     12:02:01
->
```

NOTE:

Certain processors at individual sites may not be executable with the PXQT command.

7. File Handling

7.1. Mass Storage Files

A large amount of information must be available to a time sharing system. Mass storage is where this information is kept.

The general file handling capabilities of CTS and the operating system are described in this section. Many hardware devices are mentioned that may not be found at all sites. An installation may also have some specific restrictions on the use of mass storage. Some installations may not allow files to be cataloged permanently, or may wish that these files be copied out to paper tape, magnetic tape, or removable disk packs. Check with the system administrator about any such options or restrictions.

7.1.1. Mass Storage Files in the Series 1100 Operating System

The SPERRY UNIVAC Series 1100 Operating System recognizes two standard formatted files and provides for the establishment of files with formats recognizable only by programs written especially to create and use them. The operating system manipulates and handles the information in the standard formatted files. Through utility routines it can create, modify, and copy the files, print information about their contents, print part of the contents, etc. However, the operating system is not concerned with the contents of files with formats peculiar to the programs using them. It provides for their creation, keeps track of their physical location, and performs the actual reading and writing at the request of programs to which they are assigned.

When a file is first created it is empty. An empty file has no format at all. The first information written into it determines its format. When a CREATE command (see 7.5.1) (or its Executive counterpart, the @ASG) creates a file for the first time, the Executive registers the name of the file and assigns space for it. Its format is not yet established.

The first type of standard format file is called the data file. Its format is called SDF (System Data Format). This type of file has been designed to handle sets of data images which are to be accessed sequentially. The operating system usually stores card images or print images in these files. The data file is meant to be referenced in a serial fashion. Consequently, SDF does not provide a direct or efficient way of getting to the n^{th} data image of a set. It is done by reading and discarding n–1 images. On the other hand, storage efficiency is provided by this format in that long strings of trailing spaces are discarded.

The second type of standard format file is the program file. Its format is called simply, program file format. This format provides for storing in the file any number of four basic types of elements with descriptive and indexing information needed to locate individual elements expeditiously. The four

types of elements are symbolic, relocatable, omnibus, and absolute. Symbolic elements are data images which may constitute source language statements for programs. Relocatable elements are compiled programs or subroutines in a form not yet suitable for execution, but ready to be combined with other relocatable elements in a process called collection or mapping (see 6.1.1). This produces programs ready for execution. The executable programs which result from the collection process are called absolute elements. Omnibus elements are in a nonstandard format and are used by processors written to handle their unique format.

In addition to its format, the Executive permits files to have other properties. For example, a file may be either temporary or cataloged. If it is a temporary file, it ceases to exist when the run to which it is assigned terminates. Its physical storage space then becomes available for other uses after run termination. A cataloged file, however, is retained indefinitely until the operating system is explicitly requested to discard it. The Executive maintains a directory of cataloged files so they may be assigned to runs which request them.

7.1.2. Use of Mass Storage Files by CTS

CTS makes use of four mass storage files. CTS automatically assigns them, but user files may be used in place of three of these. CTS also allows creating, deleting, and manipulating user files. The four CTS files and their normal names are:

File	File name
working area file and add file	CTS$run-id or CTS$identifier with USE name of CTS$FILE
save file	run-id or identifier
object file	TPF$
scan file	SQUELCH$

The run-id portion of the save and working area file name is the name of the run appearing on the @RUN control statement submitted to the operating system to start the run. This use of the word, "run" is not to be confused with the *RUN command under CTS which causes the program in the working area to be compiled, collected, and executed. The "identifier" portion of the save and working area file is the value obtained if the F option was specified at CTS initialization time.

The first three of these files are created when CTS has finished its initialization. They exist throughout the session. The scan file and add file portion of CTS$FILE are created only if needed.

The working area or add file may not be reassigned or otherwise manipulated. It is used internally by CTS and does not normally exist after the run terminates. The working area file is set up to be deleted when the run terminates normally. The working area file, called f in this manual, is the file into which data images are placed when entered. The working area file is neither an SDF file nor a program file. Its format is designed especially to facilitate the operations CTS must perform, such as rapidly locating a line, inserting a line, etc. The add file is the first part of CTS$FILE. It also is used by CTS to communicate with the operating system. This file is used only through CTS and is transparent to the user.

The save file is created by CTS as a cataloged file. This file is not destroyed when the run terminates. It is this file, therefore, where information is to be saved. If this file exists when CTS initializes itself, it does not attempt to create it, but assigns the existing file to the run. In this way, the information saved from the previous session is immediately available. CTS may be directed to use a different file for the save file (see 6.2.4 and 6.2.2, ASSUME PROGRAM and ASSUME FILE). This file is also referred to as F. It is a program file.

At the start of every run, the Executive creates a temporary file named TPF$ (Temporary Program File) and assigns it to the run. This is the file that CTS uses as the object file. CTS may be directed to use another file for the object file (see ASSUME OBJECT (6.2.3)and ASSUME FILE (6.2.2). The object file and save file may be the same file. The object file is a program file. It is used to store the relocatable, absolute, and symbolic elements created by RUN (see 6.2), COMPILE (see 6.4.1), or MAP (see 6.4.2) commands.

The scan file enables CTS to use compilers designed for batch (noninteractive) mode of operation in an interactive environment and to avoid some of the less desirable side effects which come from the batch orientation. Such compilers usually create more voluminous output listings than desired for direct output to a terminal. CTS directs this output to the scan file rather than the terminal. After the operation is completed, these listings or parts of them may be inspected with the same mechanism that is used to LIST, PRINT, or edit parts of f, the SCAN command (see 11.1.1). The scan file is a data file in SDF. It can be changed by doing a "USE SQUELCH$, fn" command where fn is the name of the file to be used as the scan file.

7.2. Permanent and Temporary Files

When a file is created (see 7.5.1), it must be specified as one of the following three types:

Type	Explanation
PRIVATE	This file is to become permanently cataloged in the Master File Directory but will be available to be assigned only by runs having the same project-id as the run which created the file.
PUBLIC	This file is to be cataloged as a PUBLIC file in the Master File Directory. Any run may access this file as long as the qualified file name, and read/write keys are properly specified.
TEMPORARY	This file is not to be saved by the system after the run session is terminated. A temporary file is allowed to have a name identical to that of an unassigned cataloged file.

7.3. Drum, Disk, and Tape Files

Sperry Univac provides many kinds of hardware peripheral devices on which mass storage files may reside. These differ in capacity, access time, transfer rate, and access techniques.

When a file is created, CTS requests the following:

```
DEVICE CHARACTERISTICS: >
```

The possible responses to this request are numerous. Created files will be stored on tape, drum, or disk.

1. Drum or Disk Files

For drum or disk files the response consists of the drum or disk type followed by the maximum size of the file desired. If the size is not given, an additional question is issued. If a disk file is specified, the pack-id is also solicited.

The drum or disk type response may be any of the following for sector-addressable files:

Response	Meaning
FAST	Sector-addressable file on fastest available drum devices
DISC	Sector-addressable file simulated on disk
F	Sector-addressable file on fastest available drum device
FCS	Unitized Channel Storage
F4	Sector-addressable file simulated on UNIVAC FH-432 Drum Unit
FH4	Sector-addressable file simulated on UNIVAC FH-432 Drum Unit
432	Sector-addressable file simulated on UNIVAC FH-432 Drum Unit
F17	Sector-addressable file simulated on UNIVAC FH-1782 Drum Unit
FH17	Sector-addressable file simulated on UNIVAC FH-1782 Drum Unit
1782	Sector-addressable file simulated on UNIVAC FH-1782 Drum Unit
F8	Sector-addressable file simulated on UNIVAC FH-880 Drum Unit
FH8	Sector-addressable file simulated on UNIVAC FH-880 Drum Unit
880	Sector-addressable file simulated on UNIVAC FH-880 Drum Unit
F2	Sector-addressable file on FASTRAND II and III Drum Unit, and UNIVAC 8460 Disk Unit
F14	Sector-addressable file simulated on UNIVAC 8414 Disk Unit
F24	Sector-addressable file simulated on UNIVAC 8424 Disk Unit
F25	Sector-addressable file simulated on SPERRY UNIVAC 8425 Disk Unit
F30	Sector-addressable file simulated on SPERRY UNIVAC 8430 Disk Unit
F33	Sector-addressable file simulated on SPERRY UNIVAC 8433 Disk Unit
F34	Sector-addressable file simulated on SPERRY UNIVAC 8434 Disk Unit
F40	Sector-addressable file simulated on SPERRY UNIVAC 8440 Disk Unit
F50	Sector-addressable file simulated on SPERRY UNIVAC 8405-00 Disk Unit
F54	Sector-addressable file simulated on SPERRY UNIVAC 8405-04 Disk Unit
F60	Sector-addressable file simulated on UNIVAC 8460 Disk Unit

Word-addressable drum may be obtained with the following responses:

Response	Meaning
D	Word-addressable file on available device
DCS	Word-addressable file on unitized channel or extended storage
D4	Word-addressable file on UNIVAC FH-432 Drum Unit
D8	Word-addressable file on UNIVAC FH-880 Drum Unit
D14	Word-addressable file, simulated on UNIVAC 8414 Disk Unit
D17	Word-addressable file on UNIVAC FH-1782 Drum Unit
D24	Word-addressable file, simulated on UNIVAC 8424 Disk Unit
D25	Word-addressable file, simulated on SPERRY UNIVAC 8425 Disk Unit
D30	Word-addressable file, simulated on SPERRY UNIVAC 8430 Disk Unit
D33	Word-addressable file, simulated on SPERRY UNIVAC 8433 Disk Unit
D40	Word-addressable file, simulated on UNIVAC 8440 Disk Unit

D50	Word–addressable file, simulated on SPERRY UNIVAC 8405–00 Disk Unit
D54	Word–addressable file, simulated on SPERRY UNIVAC 8405–04 Disk Unit

The type of device is followed by at least one space and a maximum file size as an integer. This is followed by at least one space and a word which indicates the type of units requested. These descriptors are as follows:

SECTORS	for 28 word units
TRACK	for 64 sector units
POSITION	for 64 track units
WORDS	for word units

If none of these is given, TRACK is assumed.

The following abbreviations can be used:

TR for TRACK
WOR for WORDS
POS for POSITION
SEC for SECTORS

If the file being created is to be used as a program file, then the minimum size of the file is 29 tracks, since 28 tracks are reserved for the directory. The default size for a created file is 128 tracks.

2. Tape Files

The tape type response may be any one of the following:

Response	Meaning
TAPE	UNISERVO VIIIC Magnetic Tape Unit
7TR	UNISERVO VIIIC (7–track) Magnetic Tape Unit
9TR	UNISERVO VIIIC (9–track) Magnetic Tape Unit
T	Tape, type independent
C	UNISERVO VIIIC, VIC, or IVC Magnetic Tape Units
C9	UNISERVO VIIIC or VIC (9–track) Magnetic Tape Units
CB	UNISERVO VIIIC, VIC or IVC Magnetic Tape Units
U	UNISERVO VIIIC, VIC, 12, or 16 (7–track) Magnetic Tape Units
U9	Density independent (9–track)
U9H	800 FPI density (9–track)
U9V	1600 FPI density (9–track)
12	UNISERVO 12 (7–track) Magnetic Tape Unit
12N	UNISERVO 12 (9–track) Magnetic Tape Unit
12D	UNISERVO 12 (dual density 9–track) Magnetic Tape Unit
14	UNISERVO 14 (7–track) Magnetic Tape Unit
14N	UNISERVO 14 (9–track) Magnetic Tape Unit
14D	UNISERVO 14 (dual density 9–track) Magnetic Tape Unit
16	UNISERVO 16 (7–track) Magnetic Tape Unit
16N	UNISERVO 16 (9–track) Magnetic Tape Unit
16D	UNISERVO 16 (dual density 9–track) Magnetic Tape Unit
20N	UNISERVO 20 (9–track) Magnetic Tape Unit
U30	UNISERVO 30 (7–track) Magnetic Tape Unit

U30N	UNISERVO 30 (9-track) Magnetic Tape Unit
U30D	UNISERVO 30 (dual density 9-track) Magnetic Tape Unit
U32N	UNISERVO 32 (9-track) Magnetic Tape Unit
U34N	UNISERVO 34 (9-track) Magnetic Tape Unit
U36N	UNISERVO 36 (9-track) Magnetic Tape Unit
8C	UNISERVO VIIIC Magnetic Tape Unit
6C	UNISERVO VIC Magnetic Tape Unit
4C	UNISERVO IVC Magnetic Tape Unit

The tape from the preceding list is followed by at least one space and a list of tape options, with a blank (space) as separator between options. The option list is as follows:

Tape Options	Meaning
6250	High density
1600	High density
800	High density
556	Medium density
200	Low density
HI	High density
LO	Low density
MED	Medium density
ODD	Odd parity (binary file)
EVEN	Even parity
BIN	Odd parity, no hardware translate
TRANS	Turns on hardware translate
DC	Turns on data converter, no hardware translate
OFF	Turns off data converter
REEL (reel no.)	Specifies tape reel to be mounted
NUMB (reel no.)	Specifies tape reel to be mounted
SAVE	Specifies a blank reel to be mounted
STORE	Same as SAVE
SCRATCH	Specifies a scratch (temporary) reel is to be mounted

The option TRANS signals that tape translation is requested. If necessary, CTS will query for the type of translation by asking: PROCESSOR CODE/TAPE CODE?>. Valid responses are:

Processor/Tape Translations

ASCII/BCD
ASCII/FLDATA
ASCII/XS3
EBCDIC/BCD
FLDATA/BCD
FLDATA/XS3
XS3/EBCDIC
XS3/ASCII
XS3/BCD
ASCII/EBCDIC
FLDATA/ASCII
FLDATA/EBCDIC

If the optional reel number is not specified, then the question REEL NO? is printed and a response of SAVE, STORE, SCRATCH, or the actual reel number is required. The question DO YOU WISH TO WRITE? is then printed, and a response of *YES* or *NO* is required. This response is necessary in order to inform the operator whether the tape should be mounted with a write ring for write enable.

The SCRATCH option indicates that the tape file is to be temporary and the reel will not be saved at the end of the run.

The SAVE and STORE options indicate that a new permanent tape file is to be created and the reel is to be saved for future use. In this case, the operator indicates the reel number of the newly created file.

7.4. Security

When a file is created using the Executive, a file name is specified. A qualifier is also attached, even if it is not specified. No one can access this file without knowing the file name. This affords a certain measure of security for the file. However, a much greater security method is available in the ability to specify read and write keys.

When a public file is created CTS will ask:

```
READ AND WRITE KEYS:>
```

Respond by transmitting a blank line if no read or write keys are desired.

If read and/or write keys are desired, respond with *r/w* where *r* is the read key and *w* is the write key. Either key must be from one to six characters long. Commas, slashes, and spaces may not be used. If only *w* is desired then respond with */w*.

These keys give additional protection against undesired use of the file.

These keys must be specified when the file is assigned in a subsequent terminal session. CTS will query for read/write keys when they are needed.

Take care that others do not know the file names, read, or write keys.

7.5. Manipulating File Contents

7.5.1. CREATE

Syntax: CREATE [,P] [s]

Abbreviation: CRE

Function: To request the characteristics of the desired file and to assign and catalog the file.

After learning the form of the Executive control language, it is possible to provide both string s and option letters P to supply all needed information to the CREATE command, and thereby avoid the question and answer sequence for creation of the file. The form of the information required for [,P] [s] is identical to that of the @ASG control statement described in SPERRY UNIVAC Series 1100 Executive System, Volume 2 EXEC, Programmer Reference, UP-4144.2 (current version).

However, the simplest form is to simply type the key word, CREATE. CTS will respond with a series of questions as follows:

- FILENAME?>

  The file name must be transmitted. It must be from 1 to 12 characters long and may be preceded by a qualifier if desired. The qualifier may be from 1 to 12 characters long and must be separated from the file name by an asterisk. For example, a response of Z indicates a file is to be created with file name Z, while a response of QUAL*Z indicates a file should be created with file name Z and the name is qualified by QUAL.

- IS THIS FILE TEMP, PUBLIC, OR PRIVATE?>

  Respond with:

  1. *TEMP (T)*, if the file is to be temporary, i.e., not to be saved after the run is completed.
  2. *PUBLIC (PUB)*, if others may have access to the file. This is the default, if no response was given to the question.
  3. *PRIVATE (PRI)*, if others are not to have access to the file. A private file is restricted to those having the same project-id (field 3 of the @RUN statement).

- READ AND WRITE KEYS:>

  (See 7.4 for appropriate response.)

- DEVICE CHARACTERISTICS:>

  (See 7.3 for appropriate response.)

After all of the questions have been answered, CTS will respond with the image in the full syntax form which could have been provided to short-cut the questions.

For example:

```
>>CRE DEF
IS THIS FILE TEMP, PUBLIC, OR PRIVATE? >T
DEVICE CHARACTERISTICS: >
*CRE,T DEF .
>>
```

Notice that the last line specifies the full syntax of the information that has been provided as responses to the questions. If the CRE were replaced with @ASG (in EXEC mode), this would be the exact image that CTS would transmit to the Executive for the assignment of this file.

7.5.2. PURGE

Syntax: PURGE [F1 [,F2...]]

Abbreviation: PUR

Function: To decatalog (eliminate from the Master File Directory) the files specified by F1, F2,...

Prior to decataloging a file, CTS must have all the appropriate information. This includes read/write keys, if they were specified when the file was created.

NOTE:

If the current working area images were read from a file by an OLD command, CTS still requires access to that file and it should not be purged or released or packed by the EXEC mode command @PACK.

If no file name F1 has been specified in the PURGE command, CTS will solicit it with the question:

```
FILE NAME?>
```

If read or write keys have not been specified as part of the file name at the time of cataloging or if these keys were specified at the time of cataloging and have already been specified by this run in some way, then CTS proceeds to purge the file from the Master File Directory.

If read or write keys were specified at the time of cataloging the file but have not been specified at some time during this terminal session, then CTS will respond:

```
*ENTER KEYS qualifier*file-name/>
```

Answer this by typing the read or write keys in the form *r/w*. CTS will then purge the file.

Once CTS has purged the file, it solicits the next command.

For example:

```
->PURGE
FILE NAME? >ABC
->PURGE A,B
*ENTER KEYS CTSDEMO*A/>A/A
*ENTER KEYS CTSDEMO*B/>B/B
->
```

Since read or write keys may be included as part of a file name, they may also be included as the file name of the PURGE command, as in the following example:

```
->PURGE A/A/A,B/B/B
->
```

Notice that this is identical to the last PURGE command in the previous example.

7.5.3. RELEASE

Syntax: RELEASE [F1 [,F2...]]

Abbreviation: REL

Function: To make the specified cataloged files available for exclusive use by other users.

It may appear that all that is necessary to gain access to a file is to reference it. This is not the case, however. CTS assigns each referenced file. The Executive allows files to be shared, and resolves any access conflicts. If, however, a file is assigned for exclusive use, others may not access the file until it is released. CTS does not assign any files for exclusive use (except for the PURGE command), but many other processors such as FURPUR do require exclusive use. This cannot be obtained unless the file is released by all other runs.

If the file contains information which others need, it should be released as soon as possible so that they may access it. Giving the file name is adequate even if read or write keys exist. This is because, to release a file, a user must have had access to the file and, in order to gain that access, must have specified any keys that existed.

NOTE:

If the current working area images were read from a file by an OLD command, CTS still requires access to that file and it should not be purged or released or packed by the EXEC mode command @PACK.

If a file name is not specified, CTS solicits the name by printing:

```
FILE NAME?>
```

After the RELEASE command, the file still exists and may be assigned by another user or it may be referenced subsequently in the same terminal session unless it was a temporary file. A temporary file no longer exists when released.

7.5.4. COPY

Syntax: COPY [,P] [F1.] [E1] [, [F2.] [E2]]

Abbreviation: COP

Function: To copy elements or files from one area to another.

F1 and F2 are user files, and E1 and E2 represent elements in these files. The straightforward case where all four parameters have been specified indicates copying of the element E1 from file F1 to file F2 with the element name E2. If an element named E2 did not exist in file F2 prior to the copy, then that element and element name are established. If an element named E2 did already exist, then the element is replaced. The input file and element F1 and E1 must exist and the output file F2 must exist. Otherwise a diagnostic appears indicating that the operation has not taken place.

When an element name is specified, all elements by that name (symbolic, relocatable, omnibus, and absolute) will be copied unless the A, O, R, and S options are specified. If the A, O, R, or S options are used, only the type of elements specified are copied. Any combination of these options may be used on a COPY or TRANSFER command.

P has the following option meanings:

P = I	Copy the input data file as an element into the output program file. This may not be used with the A, O, R, or S options.
P = R	Copy relocatable elements from the input program file to the output program file.
P = A	Same as P = R except for absolute elements.
P = O	Same as P = R except for omnibus elements.
P = S	Same as P = R except for symbolic elements.

If an option other than I, R, A, O, S is specified or an incorrect file type is used (i.e., A, R, S options on a data file), the following diagnostic is given:

```
<111> ILLEGAL OR CONFLICTING OPTION SYNTAX
```

If the file name is not specified, F is assumed.

For example:

```
>>COPY A1,A.A1
FURPUR 27R2        02/15/77   10:10:23
1 SYM
>>COPY A.A1,A2
FURPUR 0026-06/12-09:02
1 SYM
>>LIS SAV
KMB.
TYPE     NAME
BASIC A2
BASIC B1
BASIC A1
>>
```

The first COPY takes the program A1 from the program file and copies it into the file A. Notice the message in the second line indicating that the file utility routine processor, FURPUR (see SPERRY UNIVAC Series 1100 FURPUR, Programmer Reference, UP-8724 (current version)), has been activated to perform the COPY. That processor also displays the message 1 SYM which indicates that one symbolic element has been copied.

The second COPY shows F being used as the default file to receive the element. It is a COPY of the same program element from file A back to F, but now renamed as element A2. Notice the LIST SAVED command shows the existence of both the programs A1 and A2.

If both the input and output element names are omitted from the syntax, the entire file is copied. Notice in this case the requirement that periods must follow the file names. For example:

```
->COPY A., B.
FURPUR 27R2  02/15/77 17:10:01
4 SYM
->
```

will copy the entire contents of file A into file B.

In addition to issuing diagnostic messages, CTS may ask for read/write keys, may print CTS assignment diagnostics, or may print FURPUR diagnostics and messages.

7.5.5. USE

Syntax: USE F1, F2

Abbreviation: USE

Function: To associate an internal (logical) file name, F1, with an external (assigned) file name, F2.

The USE command allows specifying an additional name for a file, which is used during the terminal session or within programs, rather than the file name as it is known in the Master File Directory. This may be done for several reasons. It may be desirable to shorten a long or cumbersome file name that is to be referenced often. It may be necessary to provide a name which has already been specified in a program but the actual name of the file in the master directory is different. Rather than change

the program, a different name for the file is used. It may also help to resolve ambiguities in cases where the same file name has been specified for more than one file but with different qualifiers.

F1 is a file name from 1 to 12 characters in length.

F2 may be in the form of the fully expanded file specification [*q* *]*fn*[/*r*/*w*] where *q* is a qualifier of from 1 to 12 characters in length (this in most cases is the project-id), *fn* is the file name (up to 12 characters long), and *r*/*w* are read/write keys, each of which is up to six characters in length.

The following are examples of the USE command:

```
>>USE C,B
>>USE D.,C.
>>USE E,CTSDEMO*KMB
>>USE 10,A
>>USE F,CTSDEMO*G/G/G
<69> WARNING**FILE HAS NOT BEEN CREATED
>>CRE,U
FILE NAME? >G/G/G
DEVICE CHARACTERISTICS: >
*CRE G/G/G
>>LIS SAV F
F.
<105> FILE F IS EMPTY
>>
```

In this example, files A and B, as well as file CTSDEMO*KMB, already exist. The first USE command allows the subsequent use of the name C to reference file B. The second USE statement allows the name D to also reference file B. Note that 10 is used for one of the file names. This could be used in a FORTRAN program as logical unit number 10. Also notice the warning diagnostic in the case where one of the files had not been previously created. It is a warning only, and the subsequent creation allows full reference to the file if the run's project-id is CTSDEMO.

7.5.6. PACK

Syntax: PACK [,P] [F1 [, F2 ...]]

Abbreviation: PAC

Function: To eliminate deleted (unsaved or replaced) programs from the specified program files.

When programs are unsaved or replaced from a program file, they are not physically removed from the file and the space that they occupy is not reused. Thus, if programs are to be continually saved or replaced, the used space within the program file continues to grow, and could conceivably reach the maximum limit of that file. The PACK command allows space occupied by deleted programs to be reused. This is done by taking the current, usable information and overwriting the previously deleted information.

If specified, P is a list of options to apply to the packing process. If the file specification is omitted, the assumed program file F is packed.

The valid options for PACK are:

no option	Remove all deleted elements
A	Remove all elements except nondeleted absolute elements
I	Release all unused space even if initially reserved
N	Do not release any unused space
O	Remove all elements except nondeleted omnibus elements
P	Create an entry point table after packing the file
R	Remove all elements except nondeleted relocatable elements
S	Remove all elements except nondeleted source elements Following are examples of the PACK command:

```
->PACK A,B
FURPUR 27R2      02/15/77   14:25:36
END PACK. TEXT=15, TOC=1, SYM=7
->PACK
FURPUR 27R2      02/15/77   14:27:03
END PACK. TEXT=10, TOC=1, SYM=4
```

NOTE:

If the current working area images were read from a file by an OLD command, CTS still requires access to that file and it should not be purged or released or packed by the EXEC mode command @PACK.

7.5.7. ADD

Syntax: ADD [FN.] [E]

Abbreviation: ADD

Function: To cause the operating system to interpret line by line the contents of the specified file or element as though they had been typed at the terminal.

If FN is not specified, the program file F is assumed. If E is not specified, the entire file FN is interpreted line for line. This file must be a data file, not a program file.

The ADD command is the same as a CSF command with a string S='ADD...' unless in subroutine mode when the string S='ADD,R...' (see 7.6.).

7.5.8. ERASE

Syntax: ERASE F1 [, F2, ...]

Abbreviation: ERA

Function: To cause all previously saved programs or data in files F1, F2, ... to be deleted, without releasing or decataloging the files.

Files F1, F2, ... may be program files or data files. After they are erased, they are empty and may be used as either program or data files. This is the same as performing an UNSAVE on each program in a program file and then performing a PACK on the file, except the file remains a program file after the PACK, even though it is empty. If all file specifications are omitted, CTS will solicit a file name. Erasing and reusing a file is less costly than purging and recreating the file.

7.6. Submitting Operating System Control Statements – CSF

Syntax: CSF 's'

Abbreviation: None

Function: To submit to the Executive from within CTS a control statement via the CSF$ interface with the Executive.

The operating system provides an interface to programs via the command:

ER CSF$

Whereby, a program may submit to the Executive images of certain types of control statements. The Executive interprets these images as if they had been in the runstream. CTS uses this interface to submit the control statement specified on a CSF command.

The string s may be any one of the control statements listed below. It should be enclosed in quotes and follow the operating system rules of syntax. The result of this command is either the performance of the control statement or a diagnostic message. If no messages are displayed the operation was successful. Some of the messages are printed by CTS and further description of the error can be requested with the EXPLAIN command. These are preceded by a string in the form '<number>'. Others are printed by the Executive and are not recognized by EXPLAIN.

The string must be enclosed in quotes but need not begin with a master space (@).

The following operating system commands can be performed via the CSF command:

ADD	FREE	RSTRT
ASG	LOG	START
BRKPT	MODE	SYM
CAT	QUAL	USE
CKPT		

For an explanation of this capability and the control statements permissible, refer to SPERRY UNIVAC Series 1100 Executive System, Volume 2 EXEC, Programmer Reference, UP-4144.2 (current version).

7.7. Examples of File Usage

These examples show the creation of a data file D1, which has one record containing three images: 1.2, 2.3, and 3.4. A FORTRAN program is written to read this file and print the data images. Then a BASIC program is written to do the same function. Finally, an alternate program file is produced and the two programs are stored in this program file.

7.7.1. FORTRAN

Example:

```
->NEW D1.
->NUMBER
100 >1.2,2.3,3.4
110 >*SAVE
IS THIS FILE TEMP, PUBLIC, OR PRIVATE? >PRI
READ AND WRITE KEYS: >
DEVICE CHARACTERISTICS: >
*CRE,U D1.
->FOR F
FD FORTRAN 5R1
>>NEW F1
>>N
100 >10 FORMAT ( )
110 >READ (11,10) A,B,C
120 >WRITE (6,10) A,B,C
130 >END
140 >*SAVE
DO YOU WANT A GLOBAL SCAN? >YES
>>USE 11,D1
>>RUN
DO YOU WANT A GLOBAL SCAN? >NO
COMPILING...
              1.2000    2.3000    3.4000

NORMAL EXIT.  EXECUTION TIME:     10 MILLISECONDS.
*DIAGNOSTIC SCAN?  >NO
```

7.7.2. BASIC

Example:

```
>>BASIC
BBASIC 9R1
>>NEW JCG
>>N
100 >OPEN D1 FOR SYMBOLIC INPUT AS FILE 1
110 >INPUT FROM 1: A,B,C
120 >PRINT A,B,C
130 >END
140 >*SAVE
>>RUN
 1.2             2.3             3.4
 TIME:             .036
>>
```

7.7.3. Alternate Program File

Example:

```
>>CRE ALTPROGFILE
IS THIS FILE TEMP, PUBLIC, OR PRIVATE? >PRI
READ AND WRITE KEYS: >
DEVICE CHARACTERISTICS: >
*CRE,U ALTPROGFILE,F2
>>ASSUME PROG ALTPROGFILE
>>SAVE
>>LIS SAVED
ALTPROGFILE.
TYPE     NAME
BASIC JCG
>>COPY KMB.F1,ALTPROGFILE.F1
FURPUR 27R2      02/15/77  14:28:01
 1 SYM
>>LIST SAVED
ALTPROGFILE.
TYPE     NAME
FOR    F1
BASIC JCG
>>
```

8. Subroutines

8.1. General

CTS commands can be put together so that they will interact when executed. This is known as a subroutine.

CTS subroutines are composed of one or more CTS commands which might be generally or frequently used. They may be stored permanently and controlled by an individual, or they may be controlled by an installation, and thereby available to everyone.

The CTS commands already explained have a great deal of additional flexibility. Strings of characters may be freely substituted into the commands, and additional CTS commands will add even more flexibility.

These commands give CTS subroutines the general appearance of a programming language. This is not accidental. These new commands give the rudiments of a programming language, but these capabilities are at the command level rather than at a programming language level. These commands can read and write values, declare variables and assign values to them. Most importantly, decisions can be programmed.

These decisions may be based on the values of the variables, or on other indicators. Making decisions means that the next command executed is based on the decision made and is not necessarily the next line of code. In this case, the code is not the code of a specific language, but rather the command structure of CTS itself. Thus, any valid CTS command or any number of such commands can be skipped. An important attribute is that looping as well as jumping is allowed. This allows repetitive and selective execution. This is, in essence, a dynamic run stream.

Since the commands can be any CTS commands, programs can be executed selectively with an OLD/RUN combination. Editing commands can be skipped or executed as the program dictates, providing a programmable editor. The working area can be built or altered, thus dynamically building a batch run stream which might subsequently be stored and executed.

The CTS subroutines may be extremely complex, and therefore they may have bugs in them. Very thorough debugging is needed prior to declaring them production subroutines.

This section shows how to build, program, debug, and execute a CTS subroutine.

8.2. Building a Subroutine

In order to understand CTS subroutines, think about the form of CTS commands. They are merely strings of characters typed at a terminal. In fact, they look very much like anything else transmitted from a terminal (for example, a line of code to a BASIC or FORTRAN program, or even input to a program which is executing). To save these subroutines and manipulate them as one would a FORTRAN or BASIC program, the lines need to be numbered. The general form of the line of a CTS subroutine is:

CTS-line-number [subroutine-command-number] CTS-command

Whereas a command or line number must begin in column 1, the subroutine line number or CTS-command may begin in any column.

The maximum number of subroutine command numbers is 448. The subroutine command number must be between 1 and 131071.

For example:

```
100 10 OLD ABC
110 RUN
120 JUMP 10
```

In this example 100 is a CTS line number. The 10 is a subroutine command number. Notice in the third line a JUMP to statement 10. This 10 would always be the object of a JUMP statement and that would be its only purpose. Notice line 110 and line 120 do not have CTS subroutine command numbers. They are never jumped to and therefore do not need command numbers. If these three commands were taken together as a subroutine and executed the system would be in a loop executing the program ABC because there is no conditional jump out of the loop.

The requested lines of the working area f or of a program in F must be declared as a CTS subroutine before they are executed. This definition may be made explicitly by the SUBROUTINE or PROC command (see 8.2.2 and 8.2.3) or done automatically when a program is referenced by a CALL command (see 8.4.1).

The character mode (ASCII or Fieldata) of the lines in a subroutine definition will be determined by the mode of the working area if the subroutine definition was caused by a SUBROUTINE command. Otherwise, the mode will be determined by the mode of the source element specified on the PROC or CALL command. If this mode is not the same as the mode of the working area, each line of the subroutine definition will be translated to conform to the mode of the working area when the line is executed.

This translation may affect commands (JUMP, BRANCH, LOWER( ), TYPE...) which are checking for or using lowercase ASCII.

NOTE:

When in Fieldata mode, an attempt to use a CTS subroutine with a command line containing the Fieldata stop code character (077) will result in the line's rejection. This character has special Executive meaning and should be avoided.

Subroutines execute in SYNTAX OFF mode (see 2.4.7); therefore, commands unique to a prescanner are not allowed.

8.2.1. SAVE

For an explanation of the SAVE command, see 3.2.

Probably the most common way to build CTS subroutines is line-by-line in the working area, similar to the manner in which a BASIC or FORTRAN program is built. Then simply save the working area. Once it is saved there is nothing really unique about the CTS subroutine other than it contains CTS commands rather than images which are BASIC, FORTRAN, or some other programming language.

An example of subroutine building and saving is as follows:

```
->NEW SUB1
->N
100 >TYPE 'I AM SUBROUTINE ONE.'
110 >*SAVE
```

This is a trivial 1-line subroutine using the CTS TYPE command. Notice the naming of the working area, SUB1, and the subsequent SAVE.

Here is the execution of the preceding subroutine:

```
->NEW SUB2
->CALL SUB1
I AM SUBROUTINE ONE.
->
```

The first line was included to clear the working area and prove that the subroutine is actually executed from the SAVE file as opposed to being executed from the working area. Notice that the third line, I AM SUBROUTINE ONE. demonstrates the execution of the subroutine itself. Note also that a CALL on a subroutine automatically executes it.

Obviously, during the establishment of a subroutine in this manner the syntax of the subroutine should not be scanned. Notice in these examples that the hyphen (-) has been used as the solicitation character, indicating that no syntax prescanner was associated when the subroutine was written.

8.2.2. SUBROUTINE

Syntax: SUBROUTINE d [L]

Abbreviation: SUB

Function: To make all or part of f available as a CTS subroutine.

The SUBROUTINE command specifies that all or part of the working area is to be treated as a CTS subroutine. This subroutine is not saved in F and, therefore, is available only for the duration of the terminal session. The subroutine name d is, however, defined as a variable which will exist during the entire terminal session unless the DROP command is executed. Do not confuse this with the SUBROUTINE in FORTRAN. Notice that the SUBROUTINE command is given after CTS commands have been placed in the working area.

For example:

```
->N
100 >TYPE 'I AM SUBROUTINE TWO.'
110 >*SUBROUTINE SUB2
->CALL SUB2
I AM SUBROUTINE TWO.
->
```

After the working area has been named SUB2 by a NEW statement, the line of code 100 is entered into the working area and then the SUBROUTINE command is given. Notice that the asterisk directs immediate action by CTS which declares the entire working area as a subroutine by the name SUB2. The entire working area is only the one line 100. The calling of SUB2 results in the execution.

The subroutine will be called by the name, d, and it will comprise the lines specified by the line specification L. If the parameter d is omitted, the name of the working area is used. If L is omitted, all of the working area will be included in the subroutine.

Since a subroutine can be used as a variable (see CALL command, 8.4.1), the name of the subroutine is only the element name portion of the d parameter. That is, the qualifier, file name, and version name are not used in naming the subroutine. For this reason, subroutines should have unique element names.

This syntax leads to some rather interesting and complicated possibilities. Notice that more than one subroutine can be declared pertinent to the working area.

For example:

```
->N
100 >T 'LINE 100'
110 >T 'LINE 110'
120 >T 'LINE 120'
130 >T 'LINE 130'
140 >T 'LINE 140'
150 >T 'LINE 150'
160 >*MAN
->SUB SUB5A 100,110
->SUB SUB5B 110,120
->SUB SUB5C 100,120
->SUB SUB5D 120,150
->CALL SUB5A
LINE 100
LINE 110
->CALL SUB5B
LINE 110
LINE 120
->
```

In this example notice that T is the abbreviation for the command TYPE, and that four subroutines have been declared. Two of the subroutines have also been executed.

Notice in the preceding example that, in addition to more than one subroutine being declared some of the subroutines actually overlap. This is allowable and the declared subroutines do not interfere with one another.

Subroutines may call other subroutines. This is illustrated in the following example:

```
->N
100 >TYPE 'LINE 100'
110 >TYPE 'LINE 110'
120 >CALL SUB2
130 >TYPE 'LINE 130'
140 >*SUBROUTINE SUB1 110,130
->SUBROUTINE SUB2 100
->CALL SUB1
LINE 110
LINE 100
LINE 130
->
```

The subroutine is as it was declared at the time the SUBROUTINE command was executed. Thus, any subsequent changes of the working area will not be reflected in the content of the subroutine itself. If the subroutine is in error or should be altered, alter the working area. Then the subroutine must subsequently be redeclared.

For example:

```
100 >TYPE 'LINE 100'
110 >*SUBROUTINE SUB1 100
->CALL SUB1
LINE 100
->C /100/XXX/100
100 TYPE 'LINE XXX'
->CALL SUB1
LINE 100
->NEW DEF
->CALL SUB1
LINE 100
->P A
THE WORK AREA IS EMPTY
->
```

This example shows that a subroutine, once declared by the SUB command, is saved by the CTS system automatically, and called from the saved location rather than from f or F after a SUBROUTINE command.

NOTE:

The text of the subroutine must be saved if it is going to be used in future terminal sessions.

8.2.3. PROC

Syntax: PROC d

Abbreviation: PRO

Function: To make a stored program d available as a CTS subroutine without disturbing f.

The PROC command is similar to the SUBROUTINE capability, but there is a distinct difference.

The execution of this command is equivalent to performing the following:

```
OLD d
SUB
```

except that the working area is not altered.

Since a subroutine can be used as a variable (see CALL command, 8.4.1), the name of the subroutine is only the element name portion of the d parameter. That is, the qualifier, file name, and version name are not used in naming the subroutine. For this reason, subroutines should have unique element names.

A program need not be explicitly declared as a CTS subroutine with a SUBROUTINE or PROC command. This will be done automatically when a stored program is referenced by a CALL command (see 8.4).

The subroutine d is now available to be executed as a subroutine even if the original stored program d is later destroyed by an UNSAVE command, but it is available only for the duration of the terminal session. Replacing an element with one of the same element name as d will destroy this subroutine definition so that the new definition will be used on a subsequent CALL command.

This is very similar to what is done by the SUBROUTINE command (see 8.2.2). The SUBROUTINE declaration is static. That is, CTS takes what currently exists in the working area, stores it elsewhere, and declares that as a callable subroutine. PROC does the same thing, except the source is not from the working area f but rather from the element and file specified by d. If a file is not given, the SAVE file F is used.

For example:

```
->NEW SUB3
->N
100 >TYPE 'I AM SUBROUTINE THREE.'
110 >*SAVE
->NEW SUB4
->PROC SUB3
->UNSAVE SUB3
->OLD SUB3
<4> ELEMENT    .SUB3    CANNOT BE FOUND.
->CALL SUB3
I AM SUBROUTINE THREE.
->
```

Notice in this example that SUB3 has been saved and the working area destroyed. Then PROC SUB3 establishes SUB3 for the duration of the terminal session even though the next command, UNSAVE SUB3, destroys it from the SAVE file. The diagnostic after the OLD SUB3 statement proves that it was destroyed. The CALL SUB3 and the execution of the subroutine show that it is still available.

8.3. Programming a Subroutine

There are three ways to cause CTS commands to be treated as a CTS subroutine. Any CTS command may be a line of a subroutine.

Perhaps the easiest way to think of these subroutines is to think of them as programs (which happen to be written in the CTS language), which can stop suddenly, even in the middle of a program, to execute some other program. Also recognize the extent of the language commands, editing commands, file manipulation commands, etc.

8.3.1. Variables

Variables may be established and they may be given values and referenced by CTS commands. These variables are similar in nature to the variables contained in any programming language. They are local to CTS, however. The variable name consists of 1 to 12 alphanumeric characters, the first of which must be alphabetic. A variable may be assigned an integer value, a real value, or a string value.

NOTE:

Some variable names are reserved for internal use. These variables always begin with the letters SYS. Therefore, do not use a name starting with these three letters.

Variables may be given values by the SET command (see 8.3.2) or by the QUERY command (see 8.3.3). Variables may be used by the TYPE command (see 8.3.4) and they may be tested by the JUMP command (see 8.3.5). Variables may also be inserted into a CTS command, using the percent sign (%) (see 8.3.6). Variables may be dropped or "deactivated" by the DROP command (see 8.3.8). The use of these variables will become much clearer in the following paragraphs.

8.3.2. SET

Syntax: SET v=e

Abbreviation: S

Function: To evaluate the expression e and store the result into the variable v.

The SET command evaluates the arithmetic expression which may contain other variables, numeric constants, string constants, or functions. See 12.1 for a complete description of e.

A SET command may be abbreviated S in a subroutine but the command name must appear. When not in a subroutine (in the desk calculator) the command name may be dropped, leaving only v=e to define the variable.

For example:

```
->SET A=43.1
->SET B='ABCDEF'
->SET C=B A
->TYPE C
ABCDEF43.1
```

In the above example the first SET command involves a numeric variable the second, a string variable. The third SET is used to concatenate these variables. Since the only acceptable form is a string, the resultant variable, C, is a string variable. Notice that the concatenation is performed by placing a space between the listed variables, B and A.

8.3.3. QUERY

Syntax: QUERY [,O] v s

Abbreviation: QU

Function: To elicit a response from a user by typing the string s (usually a question) at the terminal and placing the response in the variable, v.

The QUERY command serves as the input command, providing the dynamic assignment of values to variables. Thus, QUERY allows the caller of a subroutine to assign a value to a variable. In fact, the subroutine may be programmed such that the variable is assigned many different values during a terminal session.

When this command is executed, the string s (usually a question) is printed at the terminal. Next, a response places a value in the variable v.

This command will usually appear in a CTS subroutine. It may be executed directly, however, though it is wasteful to ask a question just to have a response placed in a variable. The same thing could be done immediately with a SET command.

It is not necessary to place quotes around the string. This is because the system can easily distinguish that the string begins with the first nonblank character after the variable and ends with the last nonblank character of the line. Be careful, however, to remember to put the variable name in the command. If not, the first word of the question will become the name of a variable and only the succeeding words would be typed as the question.

For example:

```
->QUERY WHAT IS YOUR ANSWER?
IS YOUR ANSWER?  >THIS IS MY ANSWER.
->TYPE WHAT
THIS IS MY ANSWER.
->
```

In this example the word WHAT looks like the name of a variable and is, therefore, assigned the answer to the query. This is shown by the TYPE WHAT statement which does in fact type the result of the query.

If ",O" is specified, the string s is printed on the operator console at the computer site rather than on the terminal and the response is expected from the computer operator. Also see the OPR command (see 10.2.2).

A variable may be assigned an integer value, a real value, or a string value. This holds true not only for the SET command but also the QUERY command. Notice in the following example that this assignment is dynamic. That is, it may be a string at one point in the session and an integer or real value at some other point in the session.

For example:

```
->SET ANS=1
->QUERY ANS VALUE?
VALUE? >ABCDEF
->TYPE ANS
ABCDEF
```

```
->SET ANS=1
->TYPE ANS
1
->
```

Notice that CTS obviously has to look at the input value and decide whether it fits a legitimate numeric form. If so, it assigns that numeric value; otherwise, it takes the value as a string.

8.3.4. TYPE

Syntax: TYPE e1 [e2 e3...]

Abbreviation: T

Function: The expressions e1, e2... are evaluated and the results are printed by the terminal.

The TYPE command is analogous to an output command like PRINT in BASIC, or WRITE in FORTRAN. Since the expressions are evaluated first, complex expressions containing constants and variables may be specified in the TYPE command.

If the terms are not separated by an operator, concatenation is performed as the strings of characters are typed on the terminal. To separate expression values with a blank in the print, place a blank character string between them on the command or use the TAB function.

For example:

```
->SET A=43.1
->SET B='ABCDEF'
->SET C= B A
->TYPE C
ABCDEF43.1
->TYPE 'THE VALUES ARE ' B ' AND ' A
THE VALUES ARE ABCDEF AND 43.1
```

Several of the string functions which may be used in an expression are particularly useful to format the value for output in a subroutine (see 12.1.4). For example, the TXT function will return a specified portion of a string or line in f and TRM will remove the trailing blanks. The FMT function provides a flexible choice of printed forms for numeric results.

8.3.5. JUMP

Syntax: JUMP i [k]

Abbreviation: J

Function: Causes transfer of control within a CTS subroutine.

Perhaps the most important aspect of a programming language is the ability to make decisions and thus cause transfer of control out of the straight sequential order. This is provided in CTS subroutines by the JUMP command.

If the k parameter is omitted, the command is an unconditional jump. The parameter i may be either a signed or unsigned integer or a sign. If it is unsigned, then the integer is taken to be a subroutine command number. This can be seen in the command in line 160 of the following subroutine:

```
->P A
100 10 QUERY A WHAT NUMBER DO YOU WANT THE SQUARE ROOT OF?
110 JUMP 20 IF A=0
120 JUMP 30 IF A>0
130 TYPE 'I CANNOT TAKE THE SQUARE ROOT OF A NEGATIVE NUMBER.'
140 JUMP 10
150 30 TYPE SQR(A)
160 JUMP 10
170 20 RETURN
END OF FILE
->REP
->CALL SUB
WHAT NUMBER DO YOU WANT THE SQUARE ROOT OF? >4
2.
WHAT NUMBER DO YOU WANT THE SQUARE ROOT OF? >-7
I CANNOT TAKE THE SQUARE ROOT OF A NEGATIVE NUMBER.
WHAT NUMBER DO YOU WANT THE SQUARE ROOT OF?  >12
3.4641016151377546
WHAT NUMBER DO YOU WANT THE SQUARE ROOT OF?  >0
```

If the integer i is signed, then the transfer is either backward (if i is negative) i commands, or forward (if i is positive) i commands. It might be wise to avoid this programming practice, due to the problems it can create when maintaining CTS subroutines.

If i is a "+" sign it means the same as a RETURN. If i is a "–" sign control jumps to the first line of the subroutine.

In the preceding example, the JUMP at line 140 or line 160 would transfer control to the command numbered 10. This is the command appearing in line number 100. Do not confuse the command number with the CTS line number. Two examples of command numbers in the above example are the command number 30 in line number 150, and the command number 20 in line number 170.

The most valuable aspect of the JUMP command is the conditional transfer. This is accomplished by specifying the optional key, k. This key may be of two forms, either specific key words, or the form, "IF e r e".

Values for k are:

END	Transfer if the line pointer p is at either end of the working area, f.
NO END	Transfer if the line pointer p is at neither end of f.
ERR	Transfer if an error has occurred, or if the error indicator was set. (See ERROR, 8.3.5.1).
NO ERR	Transfer if an error has not occurred or if the error indicator is clear. (See ERROR, 8.3.5.1.)
FIND	Transfer if the last LOCATE, CHANGE, or FIND command was successful, or if the FIND indicator is set. (See FOUND, 8.3.5.2.)

NO FIND — Transfer if the last LOCATE, CHANGE, or FIND command was not successful, or if the FIND indicator is clear. (See FOUND, 8.3.5.2.)

IF e r e — Transfer if the relation is true. The character e is a string or numeric expression and r is one of the relational operators:

Operator	Meaning
=	equal
>	greater than
>= or =>	greater than or equal
<	less than
<= or =<	less than or equal
<> or ><	not equal

All relational operators are approximate for real operands with the single exception of zero. That is, the last six binary bits of an operand are used to round the number, and then these bits are set to zero. Hence, for comparison purposes, only slightly more than 16 significant decimal digits are used.

8.3.5.1. ERROR

Syntax: ERROR [k]

Abbreviation: ERR

Function: To clear or set the error indicator.

If the parameter k is given as the single character N, the error indicator is cleared. Otherwise the ERROR command causes the error indicator to be set. This makes it possible for a nested CTS subroutine to return an indicator to its caller showing that some type of error has occurred. The error indicator is also set when an error message is given (see ASSUME SBUG, 8.4.3.1.). The error indicator is cleared by a CALL command and, also, whenever tested by the JUMP command.

8.3.5.2. FOUND

Syntax: FOUND [k]

Abbreviation: FOU

Function: To clear or set the FIND indicator.

If the parameter k is given as the single character N, the FIND indicator is cleared. Otherwise the FIND indicator is set. This command makes a CTS subroutine return a FIND/NO FIND indication to its caller. The FIND indicator may be interrogated by the JUMP command, and it is altered by the FIND, LOCATE, and CHANGE commands.

Unlike the error indicator, the FIND indicator is not cleared by a CALL command, nor is it altered by a JUMP command.

For example:

```
->NEW SUB4
->N
100 >FOU
110 >JUMP 10 FIND
120 >TYPE '120'
130 >10 JUMP 20 FIND
140 >TYPE '140'
150 >20 TYPE '150'
160 >RETURN
170 >*SAVE
->CALL SUB4
150
-> 100 FOU N
->REP
->CALL SUB4
120
140
150
```

The first execution of the above subroutine shows that the JUMP command does not reset the FIND indicator. The indicator is initially set in statement 100 and the JUMP in command 110 occurs, then the subsequent JUMP in command 130 also occurs, showing that the JUMP in command 110 did not reset the FIND indicator.

When the FIND indicator is cleared initially in the subroutine, as is done in the second execution of the subroutine (note the alteration of statement 100 to clear the FIND indicator and the subsequent replacement of the subroutine), the JUMP commands do not alter the FIND indicator itself.

8.3.5.3. BRANCH

Syntax: BRANCH %V%,S_1, [S_2,...,S_N,] (i_1,i_2,...,i_n)

Abbreviation: BRA

Function: To allow conditional transfers of control within a subroutine like the JUMP command but with more than one subroutine command number i.

The BRANCH command compares the character string in the first field to the strings S_1,S_2,...,S_N searching for a match. If no match is found, the next subroutine command is performed. If a match is found, a jump to the subroutine line number corresponding to the matched key is performed.

Each of the strings S must be delimited by commas. Quote delimiters are acceptable but are only necessary if the string contains commas or leading or trailing quotes. The quote delimiters indicate that all characters between the quotes are to be evaluated rather than all characters between the commas. If a string is to contain a quote character two adjacent quotes must be entered for each desired quote, thus 'A"B' would be evaluated as A'B.

The BRANCH command format assumes that a variable reference (see 8.3.6) is always specified for the first field %V%. The variable delimeters % cause the variable value to be placed in the field prior to performing the character string comparison. As in any CTS command, a variable reference within % delimiters may be used in any field, not just the first field as shown.

For a successful match to occur, the value of V and string S_N must match character-for-character for the length of V. Note that V may be shorter than S_N and a match can still occur.

Here is an example:

```
->QUERY  N DO YOU WANT THE NEWS?
->BRANCH %N%,YES,NO,(10,20)
```

If the reply was *Y*, *YE* or *YES* the command would jump to line 10. If the reply was *NO* or *N* the command would jump to line 20. If anything else was specified (such as *YESS*), the next statement would be performed.

Variables may be entered via the SET command or the QUERY command. The BRANCH command is affected by how the variable is entered since the SET command evaluates the variable while the QUERY command does not.

For example, in the following subroutine:

```
SET V=1.2E2
BRANCH %V%,1.2E2,120.,(10,20)
```

The next line executed would be line 20 since V has been evaluated as 120. by the SET command.

```
QUERY V ENTER A NUMBER
BRANCH %V%,1.2E2,120.,(10,20)
```

In this example, if 1.2E2 is entered in response to the QUERY, line 10 is executed because the QUERY command does not evaluate the variable. 120 or 120. would have to be entered for line 20 to be executed.

The QUERY command treats blanks and all quotes as part of the string. If A'B were entered in response to a QUERY, it would match the string 'A' 'B' or the string A' 'B since string evaluation changes two single quotes into a single quote.

Notice also that the strings S are evaluated as strings and not integers or real numbers. That is, 123 will be evaluated as 123; 12.3E2 will be evaluated as 12.3E2 and not 1230.

8.3.6. Variable Substitution in CTS Commands

The value of a variable may be substituted into a command anywhere after the initial keyword simply by enclosing the variable name between percent (%) signs. The value of the variable may be a string, numerals, or anything allowed by the command, such as keywords, line limits, column limits, etc. This substitution occurs as if the value of the variable had been inserted manually in place of the %V% form.

NOTE:

The % character must be immediately adjacent to the variable name. If it is not, the % character will be interpreted as the start of a comment.

This means virtually everything in a CTS command can be a variable. This might be thought of as a macro capability. The only exceptions are the initial command word itself and the i parameter in the JUMP and BRANCH commands

The following is an example of some variable substitutions:

```
->100 ABCDEFGH
->110 DEF
->SET A=100
->SET B=110
->P %A%,%B%
100 ABCDEFGH
110 DEF
->SET A1=10
->P %A1%0
100 ABCDEFGH
```

The first two lines of this example place some data in lines 100 and 110 of the working area f. Then A is set to 100 and B is set to 110. The PRINT statement is abbreviated P and might be thought of as a P 100,110; and, in fact, that command types lines 100 and 110. Then the variable A1 is set to 10, and the next PRINT command is effectively a PRINT 100. The 0 following the %A1% has been concatenated by the system to the value of A1, forming 100.

This could become quite complex as is seen in the following example:

```
->SET C='0,1'
->P %A1% %C% %A1%
100 ABCDEFGH
110 DEF
->
```

This shows the concatenation of three variables, A1, C, and A1. This is the same A1 used in the previous example with the value 10. Thus, the PRINT command becomes PRINT 100,110. Again, it types the two lines.

The above examples may cause some confusion as to the difference between string variables and numeric variables. Since A,B and A1 were all numerical values, CTS takes the numeric value and converts it to a string prior to placing that string into the command. This can be seen in the following example:

```
->SET A0=00
->T 1%A0%
10
->T 1%A0%%A0%
100
->
```

Here A0 is set to 00 which is interpreted as a simple numeric zero. Thus, the TYPE command would display 10. The next TYPE command, as it includes two A0's, effectively becomes the command TYPE 100. Again, this is because CTS has taken the value (in this case 0) and included it in the string.

Due to the nature of this simple string substitution, there are some rather interesting applications. One is to specify a formula as a string of characters, and then cause the evaluation of that expression:

```
->SET EXP='(-B+SQR(B*B-4.0*A*C))/(2*A)'
->SET A=5.0
->SET B=4.0
->SET C=-5.0
->TYPE %EXP%
```

```
6.770329614269008E-1
->
```

This expression is the general solution to the quadratic equation:

$AX^2+BX+C=0$

The variable EXP is enclosed in percent (%) signs in the above TYPE command in order to have the expression evaluated. Note the difference between this and the simple TYPE EXP which would type the string of characters which represent the formula.

For example:

```
->TYPE EXP
(-B+SQR(B*B-4.0*A*C))/(2*A)
```

These percent (%) signs cannot be nested because there is no concept of left and right percent signs, as there would be in the case of left and right parentheses. Notice the following example:

```
->SET C='A'
->SET D='B'
->SET AB=47
->TYPE %C%%D%
47
->TYPE %C%D%%
<8> VARIABLE AD IS UNDEFINED
->
```

The last TYPE command in the above example is an attempt to nest the %. However, notice that the %C% is interpreted. An A, which is the value of the variable C, can be thought of as replacing the %C%. Thus A and D are now contiguous characters and are interpreted by CTS as the variable AD. This results in a diagnostic declaring that AD is undefined.

As indicated in 8.3.3, the QUERY is another command which can assign a value to a variable. This is done dynamically by typing at the terminal the value to be provided. This can be used very effectively in CTS subroutines by programming the names of programs, files, and character strings as variables.

For example:

```
->BAS
BBASIC 9R1
>>NEW ABC
>>N
100 >PRINT  'I AM A BASIC PROGRAM.'
110 >END
120 >*SAVE
>>FOR F
FD FORTRAN 5R1
>>N
100 >PRINT 10
110 >10 FORMAT (' I AM A FORTRAN PROGRAM.')
120 >END
130 >*SAVE DEF
DO YOU WANT A GLOBAL SCAN?  >YES
```

```
>>CLEAR
->NEW CTSSUB
->N
100 >QUERY PROG WHICH PROGRAM DO YOU WANT TO EXECUTE?
110 >OLD %PROG%
120 >RUN
130 >*SAVE
->CALL CTSSUB
WHICH PROGRAM DO YOU WANT TO EXECUTE?  >ABC
I AM A BASIC PROGRAM.


TIME:     .027
->CALL CTSSUB
WHICH PROGRAM DO YOU WANT TO EXECUTE? >DEF
COMPILING. . .
I AM A FORTRAN PROGRAM.

NORMAL EXIT. EXECUTION TIME:     2 MILLISECONDS
->P A
100         PRINT 10
110 10      FORMAT (' I AM A FORTRAN PROGRAM.')
120         END
END OF FILE
->
```

This example shows the establishment of a BASIC program ABC, and a FORTRAN program DEF. Also, a CTS subroutine called CTSSUB is written with a QUERY command to assign to the variable PROG, the name of the program to be executed. Then the subroutine is called twice with the names ABC and DEF provided to the QUERY and, in both cases, the programs are executed. Note that this concept could be extended considerably to a tutorial approach with, conceivably, many different programs being eventually executed based on the answers provided to various queries in the subroutine.

The example also shows that the changing of the compiler type by the OLD command is not displayed unless an ASSUME SBUG ON has been done. It also shows that the DIAGNOSTIC SCAN query never occurs in a subroutine.

8.3.7. Miscellaneous Commands

8.3.7.1. ENTRY

Syntax: ENTRY

Abbreviation: ENT

Function: To define the entry point of a CTS subroutine.

Generally, in a CTS subroutine the first command of the subroutine is the one to be executed first. That is, the subroutine will start executing at the top. In this case there is no need to define the start of the CTS subroutine. It is assumed to be at the top.

If the first statement is not to be executed first, an entry point can be defined with the ENTRY command. Inclusion of this command causes the next executable command to be the entry point for execution when the subroutine is invoked by the CALL command.

There are no parameters to the ENTRY command. In particular a name cannot be associated with an ENTRY command. Therefore, each subroutine has only one entry point, either the single ENTRY command contained in it, or the top of the subroutine if there is no ENTRY command in the subroutine.

The following is an example of the ENTRY command:

```
->SET A=0
->OLD SUBA
->P A
100 10 SET A=1
110 TYPE 'LINE 110'
120 ENTRY
130 TYPE 'LINE 130'
140 JUMP 10 IF A <>1
150 TYPE 'LINE 150'
END OF FILE
->CALL SUBA
LINE 130
LINE 110
LINE 130
LINE 150
->
```

When this subroutine is executed, the entry is at line 120 and, with A=0, the JUMP in line 140 is effective. Notice that the ENTRY is not an executable statement and, therefore, does not affect execution of any statements on the second time through the loop.

8.3.7.2. RETURN

Syntax: RETURN

Abbreviation: RET

Function: To cause control to exit from a CTS subroutine.

There are four different methods of exiting from execution of a CTS subroutine. One is to ensure that the last command in the CTS subroutine is, in fact, the last command executed. Thus, control can be thought of as dropping out of the bottom of the subroutine. This causes a return back to the CALL of the subroutine.

The second method is to execute a JUMP + command. Think of this as jumping beyond the limits of the subroutine, and thus returning control.

The third is encountering an END command (see 8.3.7.3).

The fourth method is the RETURN command. It is good programming practice to use this command to return control from a CTS subroutine. It is an executable statement and may appear anywhere in the subroutine. It may appear any number of times within the subroutine. The execution of any RETURN statement will cause control to be returned to the place where the CTS subroutine was called.

The following is an example of a RETURN command:

```
100 >JUMP 10
110 >20 TYPE 'LINE 110'
120 >RETURN
130 >10 TYPE 'LINE 130'
140 >JUMP 20
150 >*SAVE
->CALL SUBB
LINE 130
LINE 110
->
```

8.3.7.3. END

Syntax: END

Abbreviation: None

Function: To indicate the last command in a CTS subroutine and to cause control to exit from a CTS subroutine.

The END command may be the last command in a CTS subroutine but it is not needed since an implied END command is automatically supplied by CTS. If the END command is used it must be the last command in the subroutine. If it is not, the lines following the END command are ignored.

When the subroutine is executed, the END command performs the same function as the RETURN command.

For example:

```
->NEW SUB1
->NUM 10,10
10 >T 'FIRST LINE'
20 >T 'SECOND LINE'
30 >END
40 >T 'THIRD LINE'
50 >*SUB
->CALL SUB1
FIRST LINE
SECOND LINE
->
```

8.3.7.4. GENERATE

Syntax: GENERATE ,[h] [, [i] , j] [s]

Abbreviation: GEN

Function: To generate a set of line numbers in the working area f, and to place string s (if specified) into each generated line.

CTS subroutines can program the manipulation of the working area. Unfortunately, the CTS subroutine concept eliminates the fundamental ability to enter lines into the working area f. The GENERATE command retains this capability.

Think of the ways that lines may be entered into f. The commands OLD and MERGE bring lines into f from an existing file. DITTO and MOVE manipulate lines that are already in f. The only way to create a new line is to type it directly. This cannot be done in the middle of a subroutine. The GENERATE command will, however, generate or create any number of new lines in f.

The h parameter in the above syntax specifies how many lines are to be generated. If h is not specified, then 1 is assumed. The i parameter is similar to the i parameter of the NUMBER command — it specifies a starting number. If it is not given, the current line pointer value plus 1 is assumed. The j parameter specifies the increment value for the generation of each line number. Again, this is consistent with the concept of the NUMBER command.

The s parameter is a string of data to be placed into each line generated. The string may be enclosed in quotes but the quotes are not needed unless the string has leading blanks. If s is not specified, then the images will be null. Tab characters in the string s will be effective if used. The string usually uses variable substitution.

The following subroutine includes several examples of the GENERATE command:

```
->OLD GENSUB
->P A
100 NEW GENED
110 GEN 4,100,10 ABC...DEF
120 P A
130 TYPE 'LINE 130'
135 GO 130
140 GEN
150 P A
160 TYPE 'LINE 160'
170 GEN ,200,10 'LINE 200'
180 P A
190 TYPE 'LINE 190'
200 GO 200
210 GEN 2,,10 'NEW LINES'
220 P 200+
END OF FILE
->CALL GENSUB
100 ABC...DEF
110 ABC...DEF
120 ABC...DEF
130 ABC...DEF
LINE 130
100 ABC...DEF
110 ABC...DEF
120 ABC...DEF
130 ABC...DEF
131
LINE 160
100 ABC...DEF
110 ABC...DEF
120 ABC...DEF
130 ABC...DEF
131
200 LINE 200
LINE 190
200 LINE 200
```

```
201 NEW LINES
211 NEW LINES
->
```

Notice that in the GENERATE command in line 110, the quote (') signs have not been included around the string. This is because there is no ambiguity as to what the string is. In the GENERATE command in line 140, all of the parameters are the assumed parameters, generating one null line at the next available line number, 131. The GENERATE command in line 170 has the h parameter assumed, thus generating one line. The GENERATE command in line 210 has the i parameter assumed, and thus begins at the current line pointer value plus 1, or 201.

The line pointer will be set to the last line number generated by the GENERATE command. In the above example there were GO statements in line 135 and line 200. These would not have been necessary except that the intervening PRINT ALL commands alter the line pointer. These were included in order to illustrate step-by-step how the working area f has been altered. In the next example these PRINT ALL commands are omitted. Note that the default values for the generation of the line numbers are correct without the GO statements.

```
->NEW Q1
->N
100 >NEW GENED
110 >GEN 4,100,10 ABC...DEF
120 >GEN
130 >GEN ,200,10 'LINE 200'
140 >GEN 2,,10 'NEW LINES'
150 >P A
160 >*SAVE
->CALL Q1
100 ABC...DEF
110 ABC...DEF
120 ABC...DEF
130 ABC...DEF
131
200 LINE 200
201 NEW LINES
211 NEW LINES
```

8.3.7.5. Setting the Line Pointer – GO

Syntax: GO [L]

Abbreviation: None

Function: Move the line pointer to an existing line or to zero.

The current value of the line pointer, p (see 2.2.7), is used by many commands as a default line specification. The GO command sets this pointer to any existing line in f, or sets it to zero.

The n parameter may be any of the line number specification formats given in 2.2.8.2 but the following are the most useful:

n — set p to n if line number n exists; otherwise, p is unchanged.

null — set p to zero.

+ move to the top of file; p is set to zero.

- move to the end of file; p is set to zero.

+i move forward i existing lines; p is set to this line number unless END OF FILE occurs, in which case then p is set to zero.

-i move backwards i existing lines; p is set to this line number unless TOP OF FILE occurs, in which case then p is set to zero.

For example, assume lines 100, 101, 102, 103, 105, 110, and 115 are in f. The following example illustrates the three forms of GO:

```
->GO 103
->GO +2
->GO -4
->
```

The first GO command sets p to 103; the second, to 110 and the third to 101.

8.3.7.6. Commentary Information

Since the CTS subroutine provides a type of programming language, it is desirable to have comments within the CTS subroutine. There are two methods of doing this; one is to use the REMARK command, and the other is to use a percent sign (%) followed by a space.

8.3.7.6.1. REMARK

Syntax: REMARK [commentary information]

Abbreviation: REM

Function: To provide comments in a CTS subroutine.

CTS performs no function when it encounters a REMARK command. The remark command is used to insert comments in a subroutine which will be displayed when the subroutine is listed.

```
-> NEW AVC
-> N
100 > TYPE 'LINE 100'
110 > REMARK
120 > TYPE 'LINE 120'
130 > REM
140 > TYPE 'LINE 140'
150 > REMARKTYPED'LINE 150'
160 > TYPE 'LINE 160'
170 > *SAVE
-> CALL AVC
LINE 100
LINE 120
LINE 140
LINE 160
```

8.3.7.6.2. Percent-Sign (%)

Syntax: CTS command % [commentary information]

Abbreviation: None

Function: To provide comments in a CTS subroutine after a CTS command.

Comments may also appear on subroutines if the percent sign (%) followed by a space is used after a CTS command. The percent sign (%) is used only when the commentary occurs on a line with a CTS command.

Example:

```
->LIST
100 REM AUTHOR: BLH  LOCATION: CLLK  DATE: 2/14/80
110     SET C = 0              % ZERO OUT COUNTER
120 50  SET C = C + 1          % INCREMENT C BY 1
130     TYPE 'C = ' %C%        % WHAT IS IT'S VALUE?
140 REM NOTE THE DIFFERENT USES OF '%'
150     JUMP 50 IF C < 5       % DO THIS 5 TIMES
160     TYPE 'BYE'
END OF FILE
->SUB PCT
->CALL PCT
C = 1
C = 2
C = 3
C = 4
C = 5
BYE
```

8.3.7.7. Leaving CTS Mode - EXIT

The EXIT command causes an exit from CTS just as the XCTS command (see 1.4.) does, except the automatic reload bit is not cleared. The automatic reload bit forces the Executive to automatically reenter CTS as though a @CTS control statement was given. The Executive will never solicit a control statement when this bit is set.

This command can be used to execute from CTS those Executive control statements which are not allowed on a CSF command. This could be done by saving the control statements as an element and adding this element to the runstream with an ADD command from within a CTS subroutine. An EXIT command following the ADD command would place the user in control mode and execute these control statements before returning to CTS through the automatic reload.

For example:

```
->OLD CONTROL
->LIST ALL
100 @MOVE TAPE.,2
200 @COPIN TAPE.,FILE1
300 @REWIND TAPE.
END OF FILE
->OLD ADDSUB
```

```
->LIST ALL
100 ADD CONTROL
200 EXIT
END OF FILE
->CALL ADDSUB
IN EXEC MODE
```

8.3.8. Removing a Variable or Subroutine — DROP

Syntax: DROP V_1 [, V_2, V_3, ..., V_n]

Abbreviation: DRO

Function: To remove or drop variables and subroutines from CTS operating environment.

The DROP command drops or deactivates variables or subroutines with the names V_1, ..., V_n. Once dropped, these variables or subroutines can no longer be accessed unless they are reestablished by a SET, QUERY, CALL, SUB, or PROC command.

The following procedure is performed to drop V_n:

1. If V_n is an established variable, it is dropped immediately.

2. IF V_n is the name of a currently executing subroutine, then it is not dropped and an error results otherwise, the subroutine is removed.

3. If V_n is neither a variable nor a subroutine name, then an error results.

Any erroneous variable or subroutine name V_n terminates DROP command processing. Any variables or subroutines dropped prior to the error will remain deleted.

Examples:

```
-> SET V=1
-> DELETE ALL
-> 10 TYPE 'SUBROUTINE A'
-> SUB A
-> CALL A
SUBROUTINE A
-> DROP V, A
-> DROP V
<8> VARIABLE V IS UNDEFINED
-> CALL A
<4> ELEMENT A CANNOT BE FOUND
-> SET V=1
-> DROP V,BETA#
<8> VARIABLE BETA# IS UNDEFINED
-> TYPE V
<8> VARIABLE V IS UNDEFINED
```

8.4. Calling a Subroutine

8.4.1. CALL

Syntax: CALL d [s]

Abbreviation: CAL

Function: To call and execute a subroutine d.

Any subroutine which has been previously defined may be called. It may be defined by the SUBROUTINE command from the contents of f, the PROC command from a saved program, or it may be defined implicitly by another CALL command. If d has not been defined as a subroutine already, this is done automatically before it is executed.

Since a subroutine can be used as a variable, the name of the subroutine is only the element name portion of the d parameter. That is to say that the qualifier, file name, and version name are not used in naming the subroutine. For this reason, subroutines should have unique element names.

If the subroutine has been defined by a SUBROUTINE, PROC, or previous CALL command, then d is in an internal format in recovery file CTS$FILE and is named as described in 8.2.2. The subroutine d will remain defined even if the working area f or the file from which d was taken is deleted. If d has not been defined as a subroutine, then the d parameter denotes either the name of an omnibus subroutine element (see 8.5.1) or a symbolic element (see 8.2.2). If an omnibus element with the name d exists, then it is copied as the subroutine definition. If not, then symbolic element d is searched for and used in subroutine definition. Subroutine definition is done automatically (without changing f) before the subroutine is executed. When searching for an element d (done only if d is not already a subroutine) and a file name is specified as part of the d parameter, CTS will only search that file. If a file name is not specified as part of the d parameter, CTS will first search the save file F, the assume call file secondly (see 8.4.1.1), and then the system-maintained file SYS$*CLIB$. The mode of the subroutine definition will be determined by the mode of the omnibus subroutine element or symbolic element d.

NOTE:

The character mode (ASCII or Fieldata) of the lines in a subroutine definition will be determined by the mode of the working area if the subroutine definition was caused by a SUBROUTINE command. Otherwise, the mode will be determined by the mode of the source element specified on the PROC or CALL command. If this mode is not the same as the mode of the working area, each line of the subroutine definition will be translated to conform to the mode of the working area when the line is executed.

The symbol s is a string denoting a parameter to the subroutine. It is referenced within the called subroutine as the subroutine name d (see 8.4.2.).

8.4.1.1. ASSUME CALL FILE

Syntax: ASSUME CALL Fn

Abbreviation: A CAL

Function: To specify a file to be searched on a subroutine CALL.

This feature is useful in setting up project-wide CALL libraries and reduces the duplication of elements.

8.4.2. CALL Parameter

A parameter may be issued to a subroutine by utilizing the string s in the CALL syntax. (See 8.4.1.) This string is separated from d by one or more spaces. If this string is to have leading or trailing spaces, it must be enclosed in quotes. The characters "$" and "-" are legal in subroutine names, but they cannot be used if a parameter is to be passed.

The string may be retrieved within the subroutine by using the name of the subroutine as a variable. The name of the subroutine is the element name from d.

For example:

```
->NEW FILE.ABC
->N
100 >TYPE 'LINE 100'
110 >TYPE ABC
120 >TYPE 'LINE 120'
130 >*SAVE
->CALL FILE.ABC       123
LINE 100
123
LINE 120
->CALL ABC      DEF
LINE 100
DEF
LINE 120
->CALL ABC'     DEF'
LINE 100
    DEF
LINE 120
->
```

In the above example, the first CALL must be FILE.ABC since the element ABC has not been defined as a subroutine. This call also shows that the parameter can be a number and that leading blanks are ignored. The second and third calls need not specify the file name FILE since ABC has been defined as a subroutine. Even if the file name was used, the subroutine from CTS's internal file would be executed since the element name would be the same as a defined subroutine. These two calls also show that to obtain leading spaces, the string must be enclosed in quotes.

Passing the single string as a parameter is not a very complex capability. Consider, however, that CTS is only designed to provide the rudiments of a programming language. The string parameter can be parsed (subdivided into separate variables) within a subroutine. Notice the following subroutine is written to do exactly that. (The TAB command was used to cause the alignment of operations and comments as seen in the example.) It uses a comma as a separator between the

arguments and will allow null arguments within the string or, in fact, as the first or last parameter in the string.

```
->OLD PARSE
->P A
100      SET R=0                              % RIGHT COLUMN LIMIT OF PARAMETER
110      SET A=0                              % INDEX OF CURRENT PARAMETER
120 10   SET L=R+1                            % LEFT COLUMN LIMIT OF PARAMETER
130 20   SET R=R+1                            % INCREMENT RIGHT COLUMN LIMIT
140      JUMP 30 IF R>LEN(PARSE)              % IS RIGHT LIMIT BEYOND STRING?
150      JUMP 20 IF TXT(PARSE,R,R)<>',' % PARAMETER DELIMITER?
160 30   SET A=A+1                            % PARAMETER FOUND.  INCREMENT COUNT
170      SET P%A%=TXT(PARSE,L,R-1)            % SET VARIABLE TO PARAMETER
180      JUMP 10 IF R<LEN(PARSE)              % BEYOND LIMIT OF STRING?
190 40   SET B=1                              % PREPARE TO TYPE VARIABLES.
200 50   JUMP 60 IF B>A                       % LIMIT OF PARAMETERS
210      TYPE P%B%                            % TYPE PARAMETER
220      SET B=B+1                            % INCREMENT LOOP COUNT
230      JUMP 50
240 60   RETURN
END OF FILE
->
```

There are no subscripted variables in CTS. However, the subscripting of variables can be somewhat simulated as is done in line 170 by simply concatenating a variable name (in this case P), with a number, the number being supplied by the variable substitution capability. Thus the various parameters will eventually be established in the variables P1, P2, P3, etc.

The following are some calls on the PARSE subroutine and the resulting print of the individual parameters as detected by the PARSE subroutine:

```
->CALL PARSE ABC,DEF
ABC
DEF
->CALL PARSE ,ABC,

ABC

->CALL PARSE ABC,,DEF
ABC

DEF
```

The first CALL is a relatively straightforward CALL of PARSE with two parameters, ABC and DEF. The second is intended to indicate three parameters, the first and last of which are null. The last is a CALL with the middle of three parameters being null.

Obviously the simple change of one character in the subroutine would allow any character other than the comma to be a parameter delimeter.

8.4.3. Subroutine Debugging

Generally the good debugging practices associated with any programming language will pertain to debugging CTS subroutines. Most CTS subroutines will be short compared to programs written in programming languages.

The error messages associated with CTS commands are generally not printed when they come from a subroutine, because errors can be caught in the subroutine by using the JUMP ERR command. This can be somewhat tricky, since a fatal error can cause the subroutine to terminate with no message being printed. This can be circumvented, however, by the ASSUME SBUG command.

8.4.3.1. ASSUME SBUG

Syntax: ASSUME SBUG [ON/OFF]

Abbreviation: A SBU

Function: To cause error messages to be suppressed or printed during subroutine mode.

ASSUME SBUG ON will cause error messages to be printed during the execution of subroutines. In subroutines, error messages also identify the subroutine in which the error occurred and the text of the line in error, to help the user pinpoint the location of the error.

```
->NEW SBUGON
->100        SET A=1
->110   100  SET P%A%=A
->120        SET A=A+1
->130        JUMP 100 A<3
->140        TYPE 'P1= ' P1 'P2= ' P2 'P3= ' P3
->150        RETURN
->SAVE
->SUB SBUGON
->CALL SBUGON
->ASSUME SBUG ON
->CALL SBUGON
<17> KEY WORD A<3
SBUGON       :JUMP 100 A<3
<8> VARIABLE P2 IS UNDEFINED
SBUGON       :TYPE 'P1= ' P1 'P2= ' P2 'P3= ' P3
->CHANGE /100/100 IF/ 130
130          JUMP 100 IF A<3
->SUB SBUGON
->CALL SBUGON
<8> VARIABLE P3 IS UNDEFINED
SBUGON       :TYPE 'P1= ' P1 'P2= ' P2 'P3= ' P3
->CHANGE /3/4/ 130
```

```
130          JUMP 100 IF A<4
->SUB SBUGON
->CALL SBUGON
P1=1 P2=2 P3=3
->
```

Notice that when the first CALL is made, nothing is typed. Since the SBUG mode was off when an error occurred, no error message was printed.

The SBUG mode is turned on and the subsequent CALL of the subroutine shows that an expression *A<3* is thought to be a keyword. Since the subroutine name is given and the line in error is printed, it is easy to recognize that something is wrong with the syntax of that command. The problem is a missing *IF*. This is inserted with a CHANGE command, and the subroutine is replaced.

A subsequent CALL shows the variable P3 as being undefined. Obviously the loop has not been through the desired number of times, so the loop terminating check is changed from a 3 to a 4 with the CHANGE command, and a subsequent CALL of the subroutine types the desired values.

8.4.3.2. Subroutine Trace

A line-by-line trace of CTS subroutine execution may be attained by first placing the program in the CTS working area and calling SYSTRACE (formerly TRACE/SYS$) before defining the working area as a subroutine with a SUB command. Calling SYSTRACE causes additional information to be attached to each working area line so that when they are defined as subroutine commands and the subroutine is called, the commands are printed out as they are being executed. The trace also prints subroutine line numbers as they are encountered.

Example 1:

```
->OLD TRACE-TEST
->LIST
100     SET A='THE VALUE IS '
110 10  JUMP 30 IF SUB1=''
120     TYPE A SUB1
130     RETURN
140 30  TYPE 'SUB1 WAS NOT SPECIFIED'
150 END
END OF FILE
->CAL  SYSTRACE
->SUB  TRACE-TEST
->CAL  TRACE-TEST
SET A='THE VALUE IS '
LABEL  10
   JUMP 30 IF SUB1=''
LABEL  30
   TYPE 'SUB1 WAS NOT SPECIFIED'
SUB1 WAS NOT SPECIFIED
END
->
```

Note that the JUMP 30 was taken because the next line executed had a CTS subroutine line number (label) of 30.

Example 2 - this time the same subroutine is called specifying a value:

```
->CALL SUB1 1234
SET A='THE VALUE IS '
LABEL 10
JUMP 30 IF SUB1=''
TYPE A SUB1
THE VALUE IS 1234
RETURN
->
```

Note that this time the jump was not taken.

In addition to the full trace, there is a partial trace which traces JUMP instructions. This trace prints only the subroutine line numbers (labels) as they are encountered and not the subroutine commands. This trace can be obtained by placing the program in the working area and calling SYSTRACEJP (formerly TRACEJP/SYS$) before defining it as a subroutine.

Example 3:

```
->OLD TRACE-TEST
->CALL SYSTRACEJP
->SUB SUB1
->DELETE ALL
->CALL SUB1
LABEL 10
LABEL 30
SUB1 WAS NOT SPECIFIED
->CALL SUB1 10000
LABEL 10
THE VALUE IS 10000
->
```

Note that TRACE-TEST must be retrieved via an OLD command again because the working area is changed when calling the trace routines. The unnecessary subroutine line number 10 was included to show that the traced subroutine line numbers need not be the object of a jump command. Also, the DELETE ALL command shows that the working area is no longer needed after the SUB command has been done.

8.4.3.3. ASSUME TRACE

Syntax: ASSUME TRACE [ON/OFF]

Abbreviation: A TRA

Function: The ASSUME TRACE command notifies CTS whether or not to display each line, either from f or from a user-specified element, when a subroutine is created.

ASSUME TRACE ON displays each line as it is converted to internal format. ASSUME TRACE OFF or ASSUME TRACE will suppress the printing of the lines. The initial CTS state is ASSUME TRACE OFF.

For example:

```
->ASSUME TRACE ON
->NEW SUB1
->NUM 10,10
10 >T 'FIRST LINE'
20 >T 'SECOND LINE'
30 >END
40 >*SUB
T 'FIRST LINE'
T 'SECOND LINE'
END
->ASSUME TRACE OFF
->NEW SUB2
->NUM 100,10
100 >T 'THIRD LINE'
110 >T 'FOURTH LINE'
120 >END
130 >*SUB
->CALL SUB1
FIRST LINE
SECOND LINE
->CALL SUB2
THIRD LINE
FOURTH LINE
```

Specifying ASSUME TRACE ON for SUB1 resulted in the display of each line of the subroutine SUB1 upon the transmittal of the *SUB command however, specifying ASSUME TRACE OFF for SUB2 resulted in no display of the lines in the subroutine SUB2 after the *SUB command.

8.4.3.4. ASSUME JUMP

Syntax: ASSUME JUMP i

Abbreviation: A JUM

Function: To specify the maximum number i of JUMP commands which may be executed in a subroutine. Normally i is assumed to be infinite.

The ASSUME JUMP command can be used to find infinite loops in subroutines. It will cause an error message to be printed during the execution of subroutines if the subroutine executes more JUMP commands than are allowed by the ASSUME JUMP. This can be seen in the example which follows.

```
->NEW ABC
->N
100 >SET A=0
110 >5 TYPE A
```

```
120 >SET A=A+1
130 >JUMP 10 IF A>5
140 >JUMP 5
150 >10 RETURN
160 >*SAVE
->A JUMP 2
->CALL ABC
0
1
2
<30> ASSUME JUMP MAX. EXCEEDED - SUBROUTINE TERMINATED
ABC             :JUMP 5
->
```

8.4.3.5. Miscellaneous Conditions

It may be necessary for fully debugging a subroutine to program for errors, end-of-file, and end of line. The JUMP command and the LEN function can perform these checks. The following is an example of programming for errors:

```
->NEW ERR1
->N
100 >SET A=NUM(ERR1,2,3)
110 >JUMP 10 ERR
120 >RETURN
130 >10 TYPE 'IMPROPER PARAMETER.'
140 >RETURN
150 >*SAVE
->CALL ERR1 A12
->CALL ERR1 AA2
IMPROPER PARAMETER.
->ASSUME SBUG ON
->CALL ERR1 AA2
<101> ILLEGAL NUMERIC SYNTAX A2
ERR1            :SET A=NUM(ERR1,2,3)
IMPROPER PARAMETER.
->
```

In this case the check is for the validity of numeric form of the parameter issued to the subroutine. The second and third column positions of the parameter should be valid numerics. The first example of the CALL has a valid numeric, 12. The second, however, is invalid with an A2. The subroutine detects this and prints a diagnostic. Notice, in the third CALL, the effect of SBUG ON.

The next example is a subroutine EOF1, which programs for end-of-file. It looks for the five characters "DATA " in the first five columns of any line in the working area, and prints that line when it finds it.

```
->P A
100 GO -
110 ASSUME BRIEF ON
120 10 GO +1
130 JUMP 20 END
140 FIND 'DATA ' +0
150 JUMP 10 NO FIND
```

```
160 PRINT
170 JUMP 10
180 20 ASSUME BRIEF OFF
190 RETURN
END OF FILE
->REP
->OLD DATA
->P A
100 DATA 1,2,3
110 A=1.23
120 DATA 4,5,6
130 B=2.345
END OF FILE
->CALL EOF1
100 DATA 1,2,3
120 DATA 4,5,6
```

Notice the programming for the find/no-find condition.

The above example can be programmed somewhat better by realizing that the FIND command default line numbering is to begin with the line following the current line and continue until a find is successful. Thus, it is not really necessary to program for the find/no-find condition as that is a normal fallout of the FIND command, as in the example which follows:

```
->NEW EOF2
->N
100 >GO -
110 >ASSUME BRIEF ON
120 >20 FIND 'DATA '
130 >JUMP 10 END
140 >PRINT
150 >JUMP 20
160 >10 ASSUME BRIEF OFF
170 >RETURN
180 >*SAVE
->OLD DATA
->CALL EOF2
100 DATA 1,2,3
120 DATA 4,5,6
```

Since the only valuable information provided by this subroutine is the printing of the lines on which the DATA statements occur, it really was unnecessary to have a subroutine at all. In fact, it can be performed with a single statement using the ALL and REPEAT options of the FIND command.

For example:

```
->FIND 'DATA ' ALL R
100 DATA 1,2,3
120 DATA 4,5,6
END OF FILE
->
```

8.4.3.6. Displaying Variables

One very convenient method of debugging subroutines is the simple displaying of variables. Remember that a variable is defined for an entire terminal session. Having been given a value, a variable will retain that value throughout the terminal session until it is changed. Therefore, be careful to provide initialization values to variables in the subroutines. They are not assumed to be zero and a diagnostic will be printed if a variable is used which has not previously been given a value:

```
->TYPE A
<8> VARIABLE A IS UNDEFINED
```

TYPE statements can be imbedded within the subroutine to cause the various variables to be typed. This will accomplish a form of trace within the execution of the subroutine. These can be easily removed after the subroutine is debugged.

The variables can also be displayed after the subroutine has executed and control has returned to the normal CTS command mode. This will be a very important capability but, again, it is important to remember that the variables must be properly initialized in the subroutine. Simply execute the direct TYPE command after the subroutine has finished executing.

8.4.3.7. Subroutine Nesting

CTS is very careful that subroutines which are nested are not improperly called. This can be seen in the following example:

```
->NEW SUB1
->N
100 >CALL SUB1
110 >*SAVE
->CALL SUB1
<32> ILLEGAL CALL NESTING TO SUB1
```

This subroutine calls itself. CTS detects this and gives the diagnostic:

```
<32> ILLEGAL CALL NESTING TO SUB1
```

This same diagnostic may pertain to more than one subroutine, as in the following example:

```
->NEW SUB2
->N
100 >CALL SUB3
110 >*SAVE
->NEW SUB3
->N
100 >CALL SUB2
110 >*SAVE
->CALL SUB2
<32> ILLEGAL CALL NESTING TO SUB3
->CALL SUB3
<32> ILLEGAL CALL NESTING TO SUB2
```

Here two subroutines call each other. Again, CTS detects this and gives a diagnostic. Once the diagnostic is given, CTS drops out of the subroutine mode back to the command mode, soliciting a command. CTS will detect undefined subroutine command numbers within subroutines. This can

be seen in the following example:

```
->NEW SUB4
->N
100 >TYPE 'LINE 100'
110 >SET A=2
120 >JUMP 10 IF A=1
130 >RETURN
140 >*SAVE
->CALL SUB4
<29> STATEMENT NUMBER 10 IS NOT DEFINED
```

Notice that CTS has done a type of syntax analysis. Even though the JUMP to statement 10 would not be taken in the logic of this subroutine, CTS has flagged as an error the fact that statement 10 is not defined. Notice also that this flagging is done prior to the execution of any of the commands, otherwise the first line would have caused a type-out.

8.5. Saving Subroutines Between CTS Sessions

As described in 8.4.1, when a CTS subroutine is first referenced by a CALL statement, it must first be converted into an internal format and placed into CTS$FILE. Two CTS commands allow saving of this internal format as an omnibus element, as later CTS sessions may use the saved CTS subroutines without incurring the overhead of the SUB command operation.

8.5.1. Saving a Subroutine as an Omnibus Element — SSUB

Syntax: SSUB sn [d]

Abbreviation: SSU

Function: To save the internally formatted subroutine for a later CTS session.

The SSUB command saves the internal definition of a CTS subroutine sn as an omnibus CTS subroutine element named d. The omnibus element d is saved in the assumed program file unless d is specified as a file name.element name . If d is omitted, then an element named sn in the assumed program file is created. By saving the internal definition of a subroutine, processing time to SUB or PROC the CTS subroutine in another CTS session can be eliminated. By using an element reference in d that is different from sn (see 8.2.2), multiple copies of subroutine sn can be produced. Each copy can then be referenced as a CTS subroutine by a CALL command.

Examples:

```
-> 15 TYPE 'SUBROUTINE A'
-> SUB A
-> SSUB A B
-> CALL B
SUBROUTINE A
-> CREATE SUBS
IS THIS FILE TEMP, PUBLIC, OR PRIVATE?> PUBLIC
READ AND WRITE KEYS: >
DEVICE CHARACTERISTICS:> FAST
*CRE,PU    SUBS.
-> SSUB B SUBS.X
-> CALL SUBS.X
```

```
SUBROUTINE A

-> SSUB B NEW.Z

<68>  NEW IS NOT CATALOGUED
```

In this example, a new file named SUBS is created so that the CTS subroutine may be saved as the omnibus element X in it. When the user attempted to also save the CTS subroutine as the omnibus element Z in a file named NEW, an error message was received (because the file NEW did not exist).

Since parameters may be passed to a CTS subroutine via the subroutine-named variable (see 8.4.2), care should be taken when saving subroutines via the SSUB command that reference CALL parameters through the associated variable sn under a different element name (d).

Example:

```
-> N
100 > TYPE 'LINE 100'
110 > TYPE ABC
120 > TYPE 'LINE 120'
130 > *MAN
-> SUB ABC
-> SSUB ABC
-> CALL ABC     123
LINE 100
123
LINE 120
-> SSUB ABC DEF
-> CALL DEF    772
LINE 100
123
LINE 120
->
```

When the CTS subroutine is saved as the omnibus element DEF, it cannot pass parameters via the subroutine-named variable ABC anymore (i.e., ABC is not the element name). The command TYPE ABC results in the printing of the value from the execution of the omnibus element ABC. If the call to omnibus element ABC had not been made, no value would have been printed for the command TYPE ABC.

8.5.2. Replacing a Saved CTS Subroutine Element — RSUB

Syntax: RSUB sn [d]

Abbreviation: RSU

Function: To replace a CTS subroutine omnibus element created by SSUB.

The RSUB command replaces the omnibus subroutine element d by the current internal definition of subroutine sn. If d is omitted, then the omnibus subroutine element named sn in the ASSUME PROGRAM file is replaced. To perform RSUB d, an internal subroutine definition must have been previously saved with a SSUB statement as d.

Examples:

```
-> 10 TYPE 'IN SUBROUTINE A'
-> SUB A
-> CAL A
IN SUBROUTINE A
-> SSUB A B
-> CAL B
IN SUBROUTINE A
-> C /A/B/ 10
10 TYPE 'IN SUBROUTINE B'
-> SUB A
-> CAL A
IN SUBROUTINE B
-> RSUB A B
-> CAL B
IN SUBROUTINE A
-> DROP B
-> CAL B
IN SUBROUTINE B
->
```

Note that when subroutine B is called the second time, it still prints the message 'IN SUBROUTINE A' even though an RSUB was done. This is because internally the subroutine remains unchanged. Subroutine B has been internally defined by the CALL command. The DROP command is used to remove it since subroutine names are saved as variables. The final time subroutine B is called, the RSUB omnibus element is executed.

NOTE:

Since parameters may be passed to a CTS subroutine via the subroutine-named variable (see 8.4.2), care should be taken when using the RSUB statement on subroutines, since later use of the omnibus subroutine d may reference CALL parameters through the associated variable sn. See SSUB (8.5.1) for more details.

8.6. Examples

8.6.1. Selective Execution

This subroutine allows the user to choose a program to be executed. A BASIC program and a FORTRAN program are shown for illustrative purposes. Each program communicates with the user by asking for a value.

It would be very easy to extend this subroutine to make logical choices to execute any one or a combination of runs.

```
->NEW WHICH
->N
100 >10 QUERY ANS DO YOU WISH TO RUN A PROGRAM?
110 >JUMP 20 IF TXT(ANS,1,1)='N'
120 >QUERY PROG WHICH PROGRAM?
130 >OLD %PROG%
140 >RUN
```

```
150 >JUMP 10
160 >20 RETURN
170 >*SAVE
->BAS
BBASIC 9R1
>>NEW B1
>>N
100 >PRINT 'INPUT A VALUE'
110 >INPUT A
120 >PRINT A
130 >END
140 >*SAVE
>>FOR F
FD FORTRAN 5R1
>>NEW F1
>>N
100 >WRITE (6,10)
110 >10 FORMAT (' INPUT A VALUE')
120 >READ (5,20)A
130 >20 FORMAT ()
140 >WRITE (6,20) A
150 >END
160 >*SAVE
DO YOU WANT A GLOBAL SCAN?  >N
>>CLEAR
->CALL WHICH
DO YOU WISH TO RUN A PROGRAM?  >YES
WHICH PROGRAM?  >B1
INPUT A VALUE?  >2.3
2.3


TIME:    .015
DO YOU WISH TO RUN A PROGRAM?  >YES
WHICH PROGRAM?  >F1
COMPILING...
INPUT A VALUE
>2.3
    2.3000
NORMAL EXIT.  EXECUTION TIME:        7 MILLISECONDS.
DO YOU WISH TO RUN A PROGRAM?  >NO
->
```

8.6.2. Programmable Editor

The SPERRY UNIVAC Series 1100 UBASIC Compiler allows an instruction for the control of transfer in the form GO TO *+n or GO TO *-n where n is a relative number of lines. This instruction form simplifies programming in that the line number to which control is to transfer need not be known. However, it does make maintenance of a program considerably more difficult. For example, if a line is inserted between the GO TO and the point to which control is to transfer, the programmer must alter the number in the GO TO statement. Thus, this is not a good programming practice and the standardization of BASIC will probably not include this instruction form.

The following is a very simple CTS subroutine which looks for this instruction form in a program and alters it to GO TO n where n is an actual line number. It is over-simplified in the sense that it does

not allow for the more complex form ON e GO TO *+n,*-n and requires specific spacing within the GO TO. It is built, however, to search for more than one GO TO statement on a single line in situations where the nested IF...THEN...ELSE... has been used.

```
->NEW ED1
->N
100 >QUERY PROG WHICH PROGRAM?
110 >OLD %PROG%
120 >ASSUME BRIEF ON
130 >GO -
140 >10 GO +1
150 >JUMP 50 END
160 >SET D=P()
165 >SET C0=0
170 >15 SET E=C0+1
180 >LOC 'GO TO *' +0 (%E%,80)
190 >JUMP 10 NO FIND
200 >SET C0=C()+6
210 >SET C1=C()+7
220 >SET C2=C()+8
230 >SET C=TXT(+0,%C1%,%C2%)
240 >GO %C%
250 >JUMP 30 ERROR
260 >JUMP 30 END
270 >SET P=P()
280 >JUMP 60 IF TXT(C,1,1)='-'
290 >SET C='-' TXT(C,2,2)
300 >JUMP 70
310 >60 SET C='+' TXT(C,2,2)
320 >70 GO %C%
330 >DELETE (%C0%,%C2%)
340 >INSERT '%P%' (%C0%,%C2%)
350 >JUMP 15
360 >50 PRINT ALL
370 >ASSUME BRIEF OFF
380 >RETURN
390 >30 TYPE 'ERROR IN %D%'
400 >P %D%
410 >GO %D%
420 >JUMP 10
430 >*SAVE
->NEW D1
->N
100 >GO TO *+5
110 >GO TO *+3
120 >GO TO *+2 GO TO *-2
130 >GO TO *-3
140 >GO TO *-1
150 >GO TO *-2
160 >*SAVE
->CALL ED1
WHICH PROGRAM?  >D1
100 GO TO 150
110 GO TO 140
120 GO TO 140 GO TO 100
```

```
130 GO TO 100
140 GO TO 130
150 GO TO 130
->NEW D2
->N
100 >A=1
110 >IF A=1 GO TO *+1 ELSE GO TO *+2
120 >A=2
130 >END
140 >*SAVE
->CALL ED1
WHICH PROGRAM?  >D2
100 A=1
110 IF A=1 GO TO 120 ELSE GO TO 130
120 A=2
130 END
```

8.6.3. Starting Batch Runs

One of the standard Executive functions is to allow the starting of a separate batch run via a CSF interface with the Executive. This interface references a file or element within a file containing a standard operating system batch deck.

An installation might provide a subroutine similar to the one in this section which would assure that the program is in an element in a prescribed file, temporarily build a batch deck in the save file, and start it for the user. It is activated by CALL BATCH PROGNAME, where PROGNAME is the name of the program. (That is, the program name is a parameter in the CALL.) This subroutine allows placing a program in the save file, the working area, or a separate file. It insists that a separate file be created, because the batch run may run concurrently with the rest of the demand session that starts the batch run, thus requiring the sharing of a file between two runs. To alleviate any exclusive-use conflicts the subroutine insists on a separate file for the batch run.

The subroutine assumes the program is FORTRAN and specifies a compilation via the FORTRAN V compiler. Any other compiler could be substituted by an installation, or the compiler name could be a variable supplied by the user. After the compilation, of course, the program is executed. The user is allowed the flexibility of providing data for the execution of the program. This data is provided as direct card images at the solicitation of a WHAT IS IT? question. When *NO* is typed in response to that question, the data images are terminated and a @FIN image is provided. The subroutine releases the file, starts the run, and returns control for the remainder of the terminal session.

```
->OLD BATCH
->P A
100 SET ANS1='TXT(ANS,1,1)'
110 QUERY ANS HAVE YOU CREATED A SEPARATE FILE FOR THIS PROG?
120 JUMP 10 IF %ANS1%='Y'
130 TYPE 'CREATE ONE AND CALL SUBROUTINE AGAIN.'
140 RETURN
150 10 QUERY FILE WHAT IS FILE'S NAME?
160 QUERY ANS IS PROGRAM IN THAT FILE?
170 JUMP 20 IF %ANS1%='Y'
180 QUERY ANS IS PROGRAM IN WORKING AREA?
190 JUMP 30 IF %ANS1%='Y'
200 QUERY ANS IS PROGRAM IN SAVE FILE?
210 JUMP 40 IF %ANS1%='Y'
```

```
220 TYPE 'PLEASE PUT IT IN ONE OF THOSE PLACES'
230 TYPE 'AND CALL SUBROUTINE AGAIN.'
240 RETURN
250 40 OLD %BATCH%
260 30 SAVE %FILE%.%BATCH%
270 20 QUERY RUNID WHAT IS RUN-ID?
280 QUERY PROJID WHAT IS PROJ-ID?
290 NEW BATCH$
300 GEN 1,100,10 '@RUN %RUNID%,,%PROJID%,2'
310 GEN 1,110,10 '@ASG,A  %FILE%.'
320 SET PROG=TRM(BATCH)
330 GEN 1,120,10 '@FOR,S %FILE%.%PROG%,TPF$.%BATCH%'
340 GEN 1,130,10 '@XQT'
350 SET LN=140
360 QUERY ANS IS THERE ANY DATA?
370 JUMP 50 IF %ANS1%='N'
380 60 QUERY DATA WHAT IS IT?
390 JUMP 50 IF DATA='NO'
400 GEN 1,%LN%,10 '%DATA%'
410 SET LN=LN+10
420 JUMP 60
430 50 GEN 1,%LN%,10 '@MSG,N NO DECK -- LISTING TO %RUNID%'
440 SET LN=LN+10
450 GEN 1,%LN%,10 '@FIN'
460 SAVE
470 SET FILE1=APF()
480 RELEASE %FILE%
490 CSF 'START %FILE1%.BATCH$'
500 UNSAVE BATCH$
->
```

9. Operating Information and Assistance

9.1. File Information

Section 4 discusses some of the uses of the LIST command for editing and output functions. Here, the emphasis is on the LIST command used for system interrogation. This interrogation can obtain information about the options used when files were assigned, (i.e., cataloged public or private, read only, write only, etc.).

CTS can list information concerning file names specified on the LIST commands or obtain the file name associated with the working area. This information is mandatory for file manipulation, security, and assignment. For additional information regarding the different mass storage files used by CTS, refer to 7.1.2.

9.1.1. LIST CATALOG

Syntax: LIST CATALOG

Abbreviation: LIS C

Function: Lists all files cataloged under the user's project–id, which is the third field on the @RUN control statement.

The LIST command lists files which were cataloged under the user's project–id, even if they have not been previously referenced or assigned in the current run.

NOTE:

These file names are found by searching the Master File Directory which is maintained by the Executive. This search is time consuming and costly. Therefore, this command should be used with discretion.

9.1.2. LIST FILE

Syntax: LIST FILE [F1 [,F2]]

Abbreviation: LIS F

Function: Lists information concerning file name [F] specified. More than one file name may be specified. If no file name is specified, the assumed program file F is used.

Here is an example of LIST FILE:

```
->LIST FILE
FURPUR 27R2  02/17/77  08:25:20
** PROJ: OLSON              ACCNT: ////////////**
MODES: PUBLIC, ASG-D
NO. OF GRANULES ASG-D:        2       GPG=2
HIGHEST GRANULE ASG-D:       28       TOTAL ASSIGNMENTS:     23
HIGHEST TRACK WRITTEN:       28
CAT: 02/15/77  AT 09:11:35, LAST REF: 02/17/77 AT 08:24:54
->
```

9.1.3. CTS Internal File Names

The names of the working area f, the assumed program file F, and the assumed object file can be found by the values returned by the CTS functions DKN( ), APF( ), and OBJ( ). The names are printed by the TYPE command (see functions in 12.1.4 and TYPE in 12.3) in the following examples.

The name of the working area f, if any, is the file name or element name specified on the last OLD, NEW, or RENAME command:

```
->NEW ABC
->T DKN()
ABC
->
```

The name of the assumed program file (save file F) is initially *project-id*run-id* but may be changed by an ASSUME PROGRAM or ASSUME FILE command:

```
->T APF()
CTS*WEST
->
```

The name of the assumed object file is initially TPF$ but may be changed by an ASSUME OBJECT or ASSUME FILE command:

```
->T OBJ()
TPF$
->
```

9.2. Miscellaneous Operating Information

In addition to the file names and file information explained previously, other information about the operating environment can be determined by CTS functions and commands. Some of these commands are described in other subsections such as SYNTAX (see 2.4.7), LIST SAVED and LIST INUSE (see 4.1.4.2 and 4.1.4.3), and the functions DATE( ), LNG( ), and P( ) (see 12.1.4). Other useful commands are explained in this section.

9.2.1. NEWS File

A NEWS file may be established by the site which automatically solicits a response from the user after the @CTS command (see 1.3.2).

If a *NO* answer follows the file inquiry, WOULD YOU LIKE THE NEWS? the system returns to CTS control mode.

The sending of NEWS to CTS terminal users becomes important if the status of the operating system has been changed. It is possible for the console operator to send a message to all active terminal users, but that ability is limited for spontaneous messages, and the broadcast message is "overlayed" whenever another broadcast message is entered (messages may not be stored for users). Broadcast messages are also severely limited as to the amount of text.

9.2.2. Number of Lines in f – LENGTH

Syntax: LENGTH

Abbreviation: LEN

Function: To print the number of lines of data in f and the line number of the last line in f.

Printing the number of lines of data in the working area may be helpful in determining if the previous OLD, MERGE, DITTO, etc., command has done what was expected. LENGTH does this without having to list all the lines. The number of lines of data can also be obtained by using the LNG( ) function (see 12.1.4).

```
->OLD ABC 10,100
->LEN
IMAGES = 57  LAST LINE NUMBER = 100
->
```

9.2.3. DATE

Syntax: DATE

Abbreviation: None (An abbreviation for DATE would interfere with any abbreviation for the DATA command.)

Function: To print the current date and time of day.

In 12.1.4 a date function is described which will format the printing of the current date. However, the DATE command is available as a direct command to time-stamp an output listing. This is the same value as the DATE( ) function prints. Its format is:

```
->DATE
29 MAR 77       11:50:21
->
```

9.2.4. Central Processor Time – CPTIME

Syntax: CPTIME

Abbreviation: CPT

Function: To display the amount of central processor time which has been used during the current terminal session.

The following example illustrates the use of the CPTIME command and the CTS response:

```
->CPT
0M 20S
->
```

The central processor time used so far is twenty seconds.

9.2.5. STATUS

Syntax: STATUS s

Abbreviation: STA

Function: To return the value of a previously set CTS parameter.

The STATUS command returns the current value of the ASSUME, TAB, and SYNTAX commands. This command operates the same as the STATUS function (see Table 12-2), but the value of the = parameters cannot be obtained since the syntax of the command would conflict with that of the implied SET command (see 8.3.2).

The value returned is formulated as if it were part of a command to set the parameter specified by the argument. If a null value is returned, then the system default is in effect.

The argument string can be either the full parameter name or their accepted abbreviations.

The following ASSUME parameters are available through STATUS:

ASCII	AUTO	BREAKPOINT	BRIEF
CALL	CHECKOUT	COMPILER	COPY
CQUE	DEBUG	ECHO	ECOLUMN
EDIT	FILE	FILLER	HEADING
INPUTWIDTH	JUMP	LIBRARIES	LINES
MAIN	MAP	OBJECT	OCCURRENCES
OCOLUMN	PCOLUMN	POLL	PRINTWIDTH
PROGRAM	QUICK	RELOCATABLE	RESEQUENCE
RETURN	SAVELENGTH	SBUG	SCOLUMN
SITE	SPACER	STRING	TRACE
TYPE	XQT		

Other parameters available through STATUS( ) are:

= (see 1.5)
TAB
SYNTAX (ON/OFF status)

Examples:

```
-> STATUS ASCII
ON
-> A ASCII OFF
-> STATUS ASCII
OFF
-> STATUS TAB
; 11,21,39,73
```

9.3. Online Assistance

Two types of online assistance are available when working at a terminal with CTS. One provides assistance with the syntax of CTS commands. This is the HELP module. The second type of aid is the CTS command EXPLAIN which explains the meaning of error messages.

9.3.1. Command Information – HELP

Syntax: HELP

Abbreviation: HEL

Function: To provide an explanation of CTS commands, the syntax of commands, or the meaning of various fields in commands.

When the HELP command is entered, CTS responds with:

```
WHEN YOU NO LONGER NEED HELP, TYPE EXIT
TEACH? - TYPE YES OR TYPE A HELP COMMAND>
```

An answer of *YES* will cause a description of the three most used HELP commands to be printed. Then, after an answer of *YES* to another solicitation, it will describe seven additional HELP commands. After this description or in response to TEACH?, one of the following HELP commands can be entered:

EXIT	Return control to CTS.
EXPLAIN [*CTS-command*]	Print a brief description of the specified CTS command.
SYNTAX [*CTS-command*]	Print the exact syntax of the specified CTS command.
FIELD i [*CTS-command*]	Print the meaning of the i^{th} field of the specified CTS command.
LENGTH *specification*	Print the definition of the line limit specification.
TEACH	Print a description of the HELP commands.
USE	Print an explanation of some typical CTS commands.
COMMANDS	Print all CTS commands, grouped by function.
SYMBOLS	Print an explanation of the symbols used by HELP in defining syntax.
DEFINITIONS	Print an explanation of the terms used by HELP in explaining CTS commands.

If the *CTS-command* is omitted from the EXPLAIN, SYNTAX, or FIELD command, the last CTS command referenced is used.

All of these commands are fully described with examples in SPERRY UNIVAC Series 1100 Introduction to Time Sharing for CTS Users, UP-8117 (current version).

9.3.2. Error Message Information – EXPLAIN

Syntax: EXPLAIN [i]

Abbreviation: EXP

Function: To provide additional information about diagnostic messages.

The EXPLAIN command displays an explanation of error message i or the last message given if i is not specified. The i represents the number within the <> characters which precede most CTS diagnostics.

If message i is not defined or there is no such message, the following message is printed:

<88> ERROR MESSAGE n IS NOT DEFINED

Requesting an error explanation clears the last error message indicator. If i is not specified and no errors have occurred since the last EXPLAIN, the following diagnostic is given:

<85> NO ERRORS SINCE LAST *EXPLAIN

In the following example, EXPLAIN is used without an error number to explain the last diagnostic:

```
>PRINT 100
100 123456789
>INSERT 'XXX' (2,3) C
<24> STRING EXCEEDS COLUMN LIMITS
>EXPLAIN
<24> THE NUMBER OF CHARACTERS SPECIFIED IN THE
INSERT COMMAND WOULD EXCEED THE COLUMN LIMITS.  THE
ACTION WAS NOT PERFORMED.  THE STRING MAY BE EXPANDED
ON INSERT ONLY WITH NO KEY OR KEY=PACK.
>
```

10. User Communications

10.1. General

The first part of this section deals with operator communications. The main purpose of 10.1 and 10.2 is to show how communications may be established between the central site and the user on an individual basis, or from the central site to all terminals. The console operator at the central site can communicate with the remote users. Entering information at the console of the SPERRY UNIVAC Series 1100 System is similar to entering information at the remote terminal. It is not necessary to explain the use of these keyins to the remote terminal user except that a message *TB*.text... is broadcast to all active terminals, whereas a *TM*.text... is broadcast only to the individual user site-id. All messages sent to and from terminals are recorded on the operator's console and are included in the Executive accounting files, in addition to appearing on the screen at the console and the terminal.

10.2. User/Operator Communications

10.2.1. Operating System Message - @MSG

Syntax: @MSG

Abbreviation: None

Function: To communicate to the central site any message of fifty characters or less, spaces included.

Many active terminals may require facilities at any given moment. Therefore, messages to the central site should be marked with a site-id as an identifier. This site-id should be included at the beginning of the message as follows:

```
DCT236
ENTER USERID/PASSWORD
SMITH/HAPPY
```

```
NOTE:
System will feed ten lines.
*DESTROY USERID/PASSWORD ENTRY
*UNIVAC 1100 OPERATING SYSTEM VER XX.XX.XX*
>@RUN JIM,123456,SMITH,15,50/100
DATE: 052474    TIME: 091120
>@MSG..DCT236   I NEED THIS TEL NUMBER ALL MORNING
>@CTS
CTS    8R1    31 OCT 80 AT 09:13:30
>
```

In this example, the user requests the use of a certain telephone connection (number) for the morning. When this request is denied or approved, the central site replies. Since the site-id is in the first part of the message, the operator can reply quickly and accurately:

```
->*TM* THATS OK DCT236 PLEASE DISCONNECT BY 1230PM
>
```

If no action is to be taken until a reply is received, use the W option on the MSG statement. For example, this telephone line may require permission before use. The W option suspends the operation of the terminal until a reply is received. The request for the use of the telephone connection would then be:

```
>@MSG,W   I NEED THIS TEL NUMBER ALL MORNING
SORRY ONLY UNTIL 11AM
>
```

In the MSG command, there are two types of messages: the information type, @MSG; and the question type, @MSG,W. The message must be answered if the W option is used, so use the W to make certain that the message is received by the console operator, who can respond with a very short acknowledgment:

```
>OK
```

Another type of message that may be received is the TB message. This message is a broadcast message sent to all active terminals and would appear as:

```
*TB* THE SYSTEM IS GOING DOWN IN 15 MIN UP AT 10AM
>
```

This message warns all terminal users to complete their programs and terminate or not to begin. The message also informs the users that the system will be back up at 10 a.m. Such a message requires no response.

It is also possible that the central site will initiate a TB message that requires action to be taken by a user unknown to the central site. If the message:

```
*TB* RELEASE PACK JJJ.  THERE IS NO SUCH PACK!!!!!
```

applies to another terminal, do not reply with a MSG, since this would fill up the operator's screen with unnecessary messages.

This is the method that is available to communicate directly with the console operator. Sound reasoning and discretion should be used when communicating directly with the central site. This process is done only in Executive control mode, although messages may be received while in CTS, BASIC, FORTRAN, COBOL, APL 1100, etc.

If the central site must interrupt a terminal user (primarily because the specific user did not reply to a TM message), the operator has available an alternate TM message that will place his message in the output (or input) to a specified terminal.

It is also possible for the central site to communicate to the terminal if the run–id is known. In the example, the run–id JIM could be common, but the site–id is always unique.

10.2.2. CTS Message – OPR

Syntax: OPR [*]

Abbreviation: OPR

Function: This command is used to send a message to the onsite Series 1100 System console while in CTS mode.

There are two formats of the OPR command. Format 1 only sends a message, and Format 2 sends a message and solicits a response. Format 2 messages will not disappear from the screen on the Series 1100 System console until the operator responds to the message. The following examples will illustrate the two formats. At the end of the examples we will XCTS and FIN to illustrate that the messages are included in the accounting at the end of the program session.

1. Description of Format 1:

```
>@CTS,I
CTS 8R1   31 OCT 80 AT 11:40:45
IF YOU NEED ASSISTANCE TYPE  *HELP
FOR NEW FEATURES TYPE  *CALL CTS-COMMANDS
THE ASSUMED MODE IS ASCII
->*OPR DCT236 SENDS MSG WILL RUN UNTIL 12 PM
->
```

2. Description of Format 2:

```
>@CTS,I
CTS 8R1   OCT 31, 1980 AT 07:32:02
IF YOU NEED ASSISTANCE TYPE  *HELP
FOR NEW FEATURES TYPE *CALL CTS-COMMANDS
THE ASSUMED MODE IS ASCII
->OPR*  DCT236 MSG OK TO RUN UNTIL 12 PM
OK DCT236 PLEASE FIN BY 1 PM
->XCTS
IN EXEC MODE
>@FIN

RUNID: JIM      ACCT:  123456      PROJECT: SMITH
DCT 236  SENDS MSG WILL RUN UNTIL 12 PM
O-  DCT236 MSG OK TO RUN UNTIL 12 PM
O   YES ITS OK DCT236 PLEASE FIN BY 1 PM
TIME:    TOTAL: 00:00:00.035    CBSUPS:  000000350
         CAU:   00:00:00.000    I/O:  00:00:00:000
         CC/ER: 00:00:00.035    WAIT: 00:00:37.538
SUAS USED:  $   3.00  SUAS REMAINING:  $200000.00
SRC:     PS=     000000000     ES=      000000000
```

```
IMAGES READ:    4        PAGES:   2
START:    07:32:02 OCT 31,1980   FIN:   07:43:21 OCT 31,1980
*TERMINAL INACTIVE*
```

The messages were printed in the system accounting with the first message which did not solicit a response, the second message which solicited a response with a "0–", and the answer to that message with only a "0." The "0" signifies the number of solicited messages that are unanswered (it may be one or several), and the hyphen (–) indicates the solicitation from the user. The text of the message may not exceed fifty characters, spaces included, per line. Any excess will be truncated.

10.3. User/User Communications

This paragraph shows how to communicate with other CTS users by use of two commands, MAIL and LOOK. This communication may be from any user to another, since messages are established and stored according to the user's run–id, and retrieved by a CTS command. This is done in CTS mode.

10.3.1. MAIL

Syntax: MAIL [run–id]

Abbreviation: MAI

Function: Establish a message file up to ten lines long to be sent to another user under the run–id specified. If the run–id is not specified it will be solicited by CTS.

```
>@CTS,I
CTS 8R1 31 OCT 80 AT 07:37:02
IF YOU NEED ASSISTANCE TYPE  *HELP
FOR NEW FEATURES TYPE  *CALL CTS-COMMANDS
THE ASSUMED MODE IS ASCII
->*MAIL TOM
MAIL** > YOU ARE USING MY ACCOUNT NUMBER WHICH
MAIL** > WILL HAVE TO BE CHANGED UNLESS WE
MAIL** > ARE ASSIGNED THE SAME ACCOUNT.
MAIL** > MY RUN-ID IS JIM AND MY PROJECT-ID IS
MAIL** > SMITH.  MY TELEPHONE NUMBER IS
MAIL** > AREA CODE 120 PHONE 987-6543 CALL ME IF
MAIL** > YOU CAN OR ELSE DO NOT USE THE SAME
MAIL** > NUMBER AS LOGGED ON OCT 08,80.
MAIL** >STOP
>
```

The MAIL file has been created by Jim Smith for the run–id TOM. The same procedure will have to be performed for run–id's DICK and HARRY. The message was also sent to the operator console telling him to contact the three run–id/project–id users to check their LOOK file. Since there are only eight lines of input for MAIL, *STOP* was typed on the ninth line and the solicitation stopped and CTS mode returned to CTS mode solicitation.

The following MAIL command has no run-id to illustrate how the run-id will be solicited:

```
>*MAIL
TO WHOM? >HARRY
MAIL** >
```

It is a convenient way for one user to leave messages for another. It is also possible that two users may have an application that could be facilitated by use of the MAIL command.

10.3.2. LOOK

Syntax: LOOK

Abbreviation: LOO

Function: To receive any messages that may have been sent from another user-id.

After logging on and entering CTS mode, use the LOOK command to see if any messages have been received. The following example shows what was sent by the operator console to run-id HARRY:

```
>@CTS
CTS 8R1 18:51:01
->*LOOK
1 YOU ARE USING MY ACCOUNT NUMBER WHICH
2 WILL HAVE TO BE CHANGED UNLESS WE
3 ARE ASSIGNED THE SAME ACCOUNT.
4 MY RUN-ID IS JIM AND MY PROJECT-ID IS
5 SMITH.  MY TELEPHONE NUMBER IS
6 AREA CODE 120 PHONE 987-6543 CALL ME IF
7 YOU CAN OR ELSE DO NOT USE THE SAME
8 NUMBER AS LOGGED ON OCT 08,79.
9 FROM:  JIM 24 FEB81 AT 09:20:30
->*LOOK
YOU HAVE NO MAIL
>
```

The output that run-id HARRY received after entering a LOOK command was left for him in his mail by another user. All of the lines were numbered 1-8, with line number 9 giving him the information from JIM (date and time stamped). The last LOOK command was followed by the message, YOU HAVE NO MAIL. This means that MAIL messages may be received only once. The mail from all users is printed whenever a LOOK is done.

11. Debugging Techniques

11.1. Program Debugging

The process of finding out why a program does not work as conceived is called debugging. The errors causing it to malfunction are known as bugs.

A program may fail for many reasons. It may be poorly constructed to begin with. Some idiosyncrasy of the language or compiler used, unknown to the novice, may cause unexpected results. The most common cause of failure is a mistake in logic, but it may be something as simple as a typographical error. The process of putting the program together may be causing difficulties.

Several CTS features have already been mentioned to help find errors. The local syntax scan of a prescan module finds many errors in format or typing, and helps correct the lines before they are accepted as part of the program. After the program is completely keyed in, the global scan will find additional kinds of errors. It is pointed out in 2.1 that the process by which a program is created is iterative, consisting of a series of tests, updates, new tests, etc. until the program performs satisfactorily. It also mentions that many of the CTS commands are useful in detecting the existence and isolating the causes of errors, and then in expeditiously making corrections. Besides using these commands to examine the source code, useful information can be found in the object file, the add file, and the scan file.

The LIST SAVED command (see 4.1.4.2) gives a list of each element in the object file and its type. Unexpected elements may be found or, perhaps, expected elements may be missing. Possibly, elements left in the object file from a previous operation interfered with the collection process of a RUN or MAP command, or the unexpected presence of a symbolic NAME$ element may tell that f was not empty as expected when a RUN or COMPILE command was executed.

The first part of the file, CTS$FILE, is where CTS places Executive control statements. It then turns the file over to the Executive to process them. The RUN (see 6.2), COMPILE (see 6.4.1), and MAP (see 6.4.2) commands in particular use this file. The file cannot be examined directly with CTS, but since it is a data file, the command:

```
->OLD CTS$FILE.
```

will place the contents into f, where they may be examined. Of course, a knowledge of the control language of the Executive is necessary to use this technique.

A feature of CTS which is useful for debugging, convenient for operation from a terminal, and efficient involves the use of the scan file (SQUELCH$). Compilations and collections frequently produce a substantial amount of output listings, particularly if the compiler used is designed primarily for batch

mode operation. The partial run stream created by CTS in the add file for the implementation of a RUN, COMPILE, or MAP command diverts the print output of all compilations and collections away from the terminal (the normal output device) to the scan file. At the conclusion of a RUN command, therefore, the scan file will contain the output from each processor in the order they were used. This printed output always starts with the image of the control statement which called the processor. The SCAN command can load the output resulting from each processor call into the working area, f. Once in f, editing commands can examine it.

11.1.1. Examining Processor Output—SCAN

Syntax: SCAN [d] [,C]

Abbreviation: SCA

Function: To move from the scan file into f the print output of one processor.

The SCAN command places CTS in the SCAN mode if it is not already in this mode. While in SCAN mode many of the normal CTS operations cannot be performed. Programs may not be created or changed, for example. The SCAN mode is geared to the perusal of processor output. Once in the SCAN mode, the only way to return to the standard (EDIT) mode is with the EDIT command (see 11.1.2) or a NEW or OLD command. Besides establishing SCAN mode, the SCAN command moves the print output of one processor execution into f. When the SCAN mode is established, the contents (and other properties) of f are not destroyed, but they are not available until EDIT mode is reestablished. In SCAN mode, the output of only one processor execution may be in f at a time.

The processor chosen from the scan file depends on the parameter field. There are four cases:

1. Empty parameter field. CTS selects the next sequential processor output following the one previously selected. If this is the first SCAN command — the one which establishes SCAN mode — the first processor output in the scan file is selected.

2. Only d coded. This parameter is the name of the element (or file, for some processors) input to the processor. CTS selects the output of the next processor which had d as its input. If the processor is a compiler, for example, d is the name of the element compiled. A comma may be coded following d.

3. Only C coded. C is the name of the processor as it appears on the processor call statement. It must be preceded by a comma. CTS selects the next processor output from the specified processor.

4. Both d and C coded. CTS selects the processor output in which both C and d match the corresponding portions of the processor control statement.

A SCAN command does not affect the contents of the scan file. The same processor output may be called into f more than once. A RUN, COMPILE, or MAP command deletes the contents of the scan file and replaces them. The following example illustrates the above points:

```
->COM (FTN,S) A, B, C (MASM,S) D (FTN,L) E
```

This causes the following sequence of processor call statements. Each of these produces output in the scan file, the first line of which is an image of the processor call statement itself.

```
@FTN,S RUNID.A,TPF$.
@FTN,S RUNID.B,TPF$.
```

```
@FTN,S  RUNID.C,TPF$.
@MASM,S  RUNID.D,TPF$.
@FTN,L  RUNID.E,TPF$.
```

The file RUNID is the save file, F. Continuing with the example:

```
COMPILING...
*DIAGNOSTIC SCAN?  >N
```

Answering the above message with *Y* would have displayed selected lines of the scan file. Neither f nor the scan file would be modified, and SCAN mode would not be established. Continuing:

```
->SCAN
->P 1
1 @FOR,S RUNID.A, TPF$.
->SCAN
->P 1
1 @FOR,S RUNID.B, TPF$.
->SCAN ,ASM
->P 1
1 @ASM,S RUNID.D, TPF$.
->SCAN C,
->P 1
1 @FOR,S  RUNID.C, TPF$.
->SCAN ,FOR
->P 1
1 @FOR,L  RUNID.E, TPF$.
->SCAN
END OF PRINT FILE
->SCAN
->P 1
1 @FOR,S  RUNID.A, TPF$.
->SCAN FOR
<186> PROCESSOR, FOR, CANNOT BE FOUND
->EDIT
->
```

In addition to showing each of the forms of the parameter field mentioned above, the example shows an instance of moving backwards in the file (the fourth SCAN command), wrapping around the end to the beginning (the seventh SCAN command), and two diagnostics related to the SCAN command.

11.1.2. Terminating SCAN Mode—EDIT

Syntax: EDIT

Abbreviation: EDI

Function: To terminate SCAN mode and reestablish EDIT mode.

When SCAN mode is established, the current contents of f become unavailable and many CTS functions are inoperative. The EDIT command terminates SCAN mode, discards the processor output in f at the time, and reestablishes EDIT mode. Access to the information in f which was suspended by establishing SCAN mode is now restored, and f contains exactly what it did before the SCAN.

11.2. Debugging Source Code

CTS programs written in BASIC or FORTRAN (RFOR or FTN) can be debugged on a symbolic level (dealing only with the symbols, variable names, and statement numbers which are used to write the program). The contents of variables can be determined and changed during the execution of the program. The logical sequence of steps in the execution of the program can be observed. Some of this action is due to programmed statements, while other action is caused dynamically by the user from the terminal. When the program is debugged, all of the symbolic debugging statements can be ignored for production execution of the program.

11.2.1. Debug Mode – ASSUME DEBUG

Syntax: ASSUME DEBUG [ON/OFF]

Abbreviation: A DEB

Function: To enable or disable the execution-time trace and diagnostic features of BASIC, RFOR, and FTN described in this section.

The appropriate compiler must be assumed. This can be done with the ASSUME COMPILER command or with a BASIC or FORTRAN command. If the compiler is called explicitly with the COMPILE statement and trace and diagnostic features are to be enabled, then the B option must be specified with the compiler on the COMPILE statement.

When ASSUME DEBUG OFF is keyed in, debugging features are disabled. These debugging features may also be disabled by changing the ASSUME COMPILER or by clearing the working area or bringing another program into the working area.

The DEBUG setting is OFF unless otherwise specified.

11.2.2. BASIC

This paragraph describes four different types of commands:

1. The CTS command ASSUME DEBUG ON/OFF (see 11.2.1.)
2. The BASIC statements PAUSE (see 11.2.2.1) and TRACE (see 11.2.2.3)
3. Answers to the question COMMAND? (see 11.2.2.1)
4. The @@X command

11.2.2.1. PAUSE

The execution of a BASIC program may be halted at any point to examine variables with the statement:

n PAUSE

at the specified point in the program.

For example:

```
->BAS
BBASIC 9R1
>>NEW ABC
>>N
100 >PAUSE
110 >END
120 >*SAVE
```

With DEBUG OFF (see 11.2.1), the PAUSE statement has no effect:

```
>>RUN

 TIME :        .012
```

After an ASSUME DEBUG ON, however, the following results:

```
>>ASSUME DEBUG ON
>>RUN
PAUSE AT LINE NO:   100
COMMAND? >RESUME
 TIME :        .018
```

The PAUSE statement caused the message:

```
PAUSE AT LINE NO: 100
COMMAND? >
```

to be printed.

In the preceding example the word, *RESUME* was typed as an answer to the question COMMAND? One of the following may be transmitted as a response to the COMMAND? question:

PRINT vl	Print the values of the listed variables.
SET v=e	Set the variable v to a new value as calculated from the expression e.
VAR=ZERO	Set all algebraic variables to zero and all string variables to blanks (set current length to zero).
DEBUG ON	Turn the debug mode on.
DEBUG OFF	Turn the debug mode off.
RESUME	Resume execution.
STOP	Terminate the executing program.
DUMP	Terminate the executing program with a postmortem dump.
TIME	Print accumulated run time and resume execution.

If one of the first five of the above commands is transmitted, BASIC solicits additional commands until a RESUME, STOP, TIME, or DUMP is sent.

The following are examples of the preceding commands:

```
->BASIC
BBASIC 9R1
>>NEW ABC
>>N
100 >FOR I=1 TO 10
110 >A=I*2
120 >PAUSE
```

```
130 >NEXT I
140 >END
150 >*ASSUME DEBUG ON
>>RUN
PAUSE AT LINE NO: 120
COMMAND? >PRINT A, I
 2                1
COMMAND? >RESUME
PAUSE AT LINE NO: 120
COMMAND? >PRINT A, I
 4                2
COMMAND? >SET I=7
COMMAND? >RESUME
PAUSE AT LINE NO: 120
COMMAND? >PRINT A, I
 16               8                 results of SET
COMMAND? >VAR=ZERO
COMMAND? >RESUME
PAUSE AT LINE NO: 120
COMMAND? >PRINT A, I
 2                1                 results of VAR=ZERO
COMMAND? >SET B=A**2               NOTE: Previously unused variable and more
                                   complex expression.
COMMAND? >PRINT B
 4
COMMAND? >STOP
TIME:     .073
>>115 PRINT A, I                   NOTE: Inclusion of new statement.
>>RUN
 2                1
PAUSE AT LINE NO: 120
COMMAND? >SET I=6
COMMAND? >RESUME
 14               7
PAUSE AT LINE NO: 120
COMMAND? >DEBUG OFF
COMMAND? >RESUME
 16               8                 results of DEBUG OFF
 18               9
 20               10


 TIME:     .028
>>RUN
 2                1
PAUSE AT LINE NO: 120              NOTE: DEBUG OFF as answer to COMMAND?
                                   does not turn ASSUME DEBUG OFF for next
COMMAND? >>                        RUN.
```

11.2.2.2. BREAK

If the DEBUG mode is on (see 11.2.1) depressing the BREAK key effects an orderly break in program execution. Depressing this key immediately causes output to cease (even in the middle of an output line) and the following message to be typed:

```
*OUTPUT INTERRUPT
```

The system returns the cursor to the beginning of the next line or line-feeds the terminal without a solicitation character. The break key does not stop the execution of the program. To interrupt the execution of the program, enter:

```
@@X C
```

If the terminal was in the process of printing a line when the break key was depressed, the system responds by printing that line again. Since the printing of the output may be slower than the execution of the program, other output lines, which were queued before the @@X C was entered, are printed. In fact, if the program is small it may complete before an @@X C can be entered. Assuming this is not the case, the message:

```
BREAK AT LINE NUMBER: xxx
COMMAND? >>
```

is printed. One of the commands described in 11.2.2.1 can be entered.

If a break is desired and the output is not needed, then enter:

```
@@X CO
```

If the terminal was in the process of printing a line when the interrupt key was depressed, the system responds by printing that line again. The O option causes all other output up to and including the COMMAND? query to be discarded. The system will respond with a cursor. A command described in 11.2.2.1 can be entered.

For example:

```
>>NEW P1
>>N
100 >PRINT 'START'
110 >FOR I=1 TO 1000000
120 >A=A+1
130 >PRINT A, I
140 >NEXT I
150 >END
160 >*RUN
START
 1                1
 2                2                    BREAK key depressed
*OUTPUT INTERRUPT
> @@X C
 2                2
>                                      many lines of output have been skipped
 414                414
 415                415
BREAK AT LINE NO: 130
COMMAND? >SET I=4000                   SET command
```

```
COMMAND? >RESUME                          RESUME command
 416              4001
 417              4002
 418              4003                    BREAK key depressed


*OUTPUT INTERRUPT
@@X OC                                    many lines of output are discarded including
                                          BREAK and COMMAND messages
 419              4004
>                                         carriage return follows >
                                          (a command could have been entered)
INVALID COMMAND-VALID COMMANDS ARE:
PRINT,SET,RESUME,STOP,DUMP,DEBUG ON/OFF,TIME,VAR=ZERO
COMMAND? >RESUME
 813              4398
 814              43                      incomplete line when BREAK depressed
*OUTPUT INTERRUPT
@@X OC
 814              4399                    carriage return follows >
>SEI I=A                                  misspelled command
COMMAND ERR-NOT PRINT,SET,RESUME,STOP,DUMP,DEBUG ON-OFF,TIME,VARZERO
COMMAND? >SET I=A
COMMAND? >RESUME
 1211             1211                    note effect of SET command
 1212             1212
 1213             1213
 1214             1214
 1215             1215
*OUTPUT INTERRUPT
@@X OC
 1216             1216
>STOP                                     STOP command
 TIME: 4.309
```

If the ASSUME DEBUG ON command is not given to CTS prior to the run of a BASIC program, the orderly break cannot be effective. Any attempt to interrupt that program will stop execution of it. This has essentially the same effect as the STOP command (see 11.2.2.1), except the BREAK and COMMAND messages do not appear on the terminal.

Rather, the simple statement PROGRAM STOPPED appears:

```
>> ASSUME DEBUG OFF
>>RUN
START
 1                1
 2                2
*OUTPUT INTERRUPT
@@X C
 3                3
>
  PROGRAM STOPPED.
  TIME: 1.076
```

11.2.2.3. TRACE

The logic flow of a BASIC program can be determined by tracing the program execution. To do this, choose the block of statements in the program to be traced and bracket them by TRACE ON/OFF statements as follows:

```
n1   TRACE ON
.
.
statements to be traced
.
.
n2   TRACE OFF
```

These are BASIC statements which must have line numbers and which must be entered into the working area.

When any of the statements to be traced is first executed, BASIC solicits the trace information to be provided for subsequent statement traces. This is done with the messages as follows:

```
TRACE OUTPUT TO FILE? YES OR NO>
```

A *YES* answer causes trace output to go to the file BTRACE$. Otherwise, it will come to the terminal.

```
LINE NUMBER ONLY?  YES OR NO >>
```

A *YES* answer causes execution to resume. A *NO* answer causes the message:

```
ALL VARIABLES?  YES OR NO >>
```

to be typed. A *NO* answer is followed by the question:

```
WHICH VARIABLES?  >>
```

This is answered by the names of the variables separated by commas. Having determined that these are legitimate names, or if the ALL VARIABLES question is answered by *YES*, BASIC asks:

```
ONLY WHEN CHANGED?  YES OR  NO  >>
```

After a *YES* or *NO* answer, BASIC continues execution. Prior to the execution of each statement in the specified block, BASIC prints:

```
TRACE xxx
```

where xxx is the line number. Then all variables, specified variables, or no variables are printed according to the user's specifications.

For example:

```
>BASIC
BBASIC 9R1
>>NEW ABC
>>N
100 >PRINT 'LINE 100'
110 >TRACE ON
120 >PRINT 'LINE 120'
130 >TRACE OFF
140 >PRINT 'LINE 140'
150 >TRACE ON
```

```
160 >PRINT 'LINE 160'
170 >END
180 >*SAVE
>>ASSUME DEBUG ON
>>RUN
LINE 100
TRACE 120 TRACE OUTPUT TO FILE? YES OR NO >>NO
LINE NUMBER ONLY?  YES OR NO >YES
TRACE 120
LINE 120
LINE 140
TRACE 160
LINE 160
TRACE 170
 TIME :      .031
```

In this case, only line number tracing was selected. Notice the interspersed output and the trace messages themselves.

If DEBUG is turned off, TRACE statements have no effect. This can be accomplished through the ASSUME DEBUG OFF command (see 11.2.1) or by answering DEBUG OFF to a BREAK or PAUSE command (see 11.2.2.2 or 11.2.2.1).

For example:

```
>>ASSUME DEBUG OFF
>>RUN
LINE 100
LINE 120
LINE 140
LINE 160

 TIME :      .035
```

This is another run of the previous example, and the ASSUME DEBUG OFF caused the TRACE statements to have no effect.

The following example shows the tracing of the symbolic variables. Notice the option, ONLY WHEN CHANGED, is used so that variables were typed only when they are changed by a previous statement.

```
>>P A
100 PRINT 'LINE 100'
105 TRACE ON
110 FOR I=1 TO 3
120 A=A+1
130 B=A=A+1
140 NEXT I
150 TRACE OFF
160 END
END OF FILE
>>ASSUME DEBUG ON
>>RUN
LINE 100
TRACE 110
TRACE OUTPUT TO FILE? YES OR NO >NO
LINE NUMBER ONLY? YES OR NO >NO
ALL VARIABLES?  YES OR NO >YES
```

```
ONLY WHEN CHANGED?  YES OR NO >YES
TRACE 110
A=0           B=0           I=0

TRACE 120
I=1

TRACE 130
A=1
TRACE 140
A=2           B=2

TRACE 120
I=2

TRACE 130
A=3

TRACE 140
A=4           B=4

TRACE 120
I=3

TRACE 130
A=5

TRACE 140
A=6           B=6


 TIME :        .102
```

Any number of blocks of a program may be traced by bracketing those blocks with TRACE ON/TRACE OFF instruction pairs. If the final TRACE OFF is omitted, then tracing is effective through the end of the program.

To change the trace data specifications (e.g., add a variable to the list or change from line numbers only to all variables), turn DEBUG OFF (after an interrupt or PAUSE), and turn DEBUG ON as follows:

```
PAUSE AT LINE NO: xxx
COMMAND? >>DEBUG OFF
COMMAND? >>DEBUG ON
COMMAND? >>RESUME
```

This causes the next traced line to go through the trace solicitation procedure "first time through", at which time different answers are given. The answer SAME to the question WHICH VARIABLES? results in the same list previously submitted in the current run.

For example:

```
>BASIC
BBASIC 9R1
>>NEW S3
>>N
```

```
100 >TRACE ON
110 >FOR I=1 TO 3
120 >A=A+1
130 >NEXT I
140 >END
150 >*ASSUME DEBUG ON
>>125 PAUSE
>>RUN
TRACE 110
TRACE OUTPUT TO FILE? YES OR NO >NO
LINE NUMBER ONLY?  YES OR NO >YES
TRACE 110
TRACE 120
TRACE 125
PAUSE AT LINE NO: 125
COMMAND? >DEBUG OFF
COMMAND? >DEBUG ON
COMMAND? >RESUME
TRACE 130
TRACE OUTPUT TO FILE? YES OR NO >NO
LINE NUMBER ONLY?  YES OR NO >NO
ALL VARIABLES?  YES OR NO >YES
ONLY WHEN CHANGED? YES OR NO >YES
TRACE 130
A=1           I=1
TRACE 120
I=2
TRACE 125
A=2
PAUSE AT LINE NO: 125
COMMAND? >STOP

 TIME :       .081
```

11.2.3. FTN

Five different statements form the debug facility for FTN (see 11.2.3.1). There is a powerful interactive debugging system in FTN while in checkout mode (see 11.2.3.2). The ASSUME DEBUG ON/OFF command has the same effect in FTN as it has in BASIC and RFOR (see 11.2.1).

11.2.3.1. Debug Facility

Debugging aids available are: subscript checking, label tracing, tracing of changes in values, tracing of entry and exit for subprograms, and simple output.

The DEBUG statement sets the conditions for operation of the debug facility and designates debugging operations that apply to the entire program unit (such as subscript checking). The debug packet identification statement (AT) identifies the beginning of the debug packet and the point in the program unit at which debugging is to begin. The three executable statements (TRACE ON, TRACE OFF, and DISPLAY) designate actions to be taken at specific points in the program unit.

Several packets may appear in the program but only one DEBUG statement may exist in each program unit. The AT, TRACE ON, TRACE OFF, and DISPLAY statements may not appear before the DEBUG statement. Within the program unit, debug packets must be located after all regular code of the

FORTRAN main program or subprogram, but preceding the END statement. Any normal FORTRAN executable, data, or format statement may also occur in a debug packet. The debug packet may be terminated only by another AT statement or the END statement for that FORTRAN main program or subprogram.

The individual DEBUG statements are explained in detail in the following sections.

11.2.3.1.1. DEBUG

The DEBUG statement is used to indicate the existence of a debug facility for the given FORTRAN program or subprogram and to specify the debugging environment. It has the form:

DEBUG [P [,P] ...]

Where P is any of the five debugging environment specifications:

UNIT(c), SUBCHK, TRACE, INIT, and SUBTRACE.

Zero to five of these options may appear in the option list following the DEBUG keyword. They may be given in any order.

There must be a single DEBUG statement for each program unit to be debugged and it must immediately precede the first debug packet.

If the UNIT option is not specified, any debugging output will be put in the standard program output file.

If the TRACE option is omitted from the DEBUG option list, there can be no display of program flow by statement numbers within the program unit.

Examples:

```
      DEBUG
C             Indicates debugging is enabled.  Debug action is
C             specified in an associated AT statement.  Output is
C             put in the standard system output file.
      DEBUG SUBTRACE,UNIT(4),SUBCHK(ARRAY1,BUNCH2,GROUP3),INIT
C             Subscripts are checked for arrays ARRAY1, BUNCH2,
C             and GROUP3.  Changes in values of all variables are
C             noted.  Debug output is put on unit number 4.
      DEBUG TRACE,INIT(C,LIST1,E),SUBCHK
C             Debugging will include subscript checking on all
C             variables, list of program flow by statement number
C             passage, and notation of changes in value of
C             C, LIST1, and  E.
```

- UNIT

  UNIT is used to designate a particular output file for debug information. It has the form:

  UNIT (c)

  where c is an integer constant representing a file reference number.

All debugging output will go to the designated file.

The file number may not change within an executable program; e.g., if the main FORTRAN program specifies UNIT(8), a subprogram called by this main program must specify UNIT(8) if it has a DEBUG statement.

If this option is not present, all debugging will be put in the standard output file.

For example:

```
      DEBUG UNIT (25)
    C              Sends all debug output to file numbered 25.
```

- SUBCHK

SUBCHK is used to check the validity of subscripts of array elements referenced in the program unit.

It has the form:

SUBCHK [(n [,n] ...)]

Where each n is an array name.

If the list of array names is not given following the SUBCHK option, subscript checking is done for all arrays in the program unit.

The check is made by comparing the size of the array with the product of the subscripts. A message will be placed in the debug output file if an out-of-range subscript expression is encountered. The incorrect subscript will still be used in the continued program execution.

If this option is omitted, no subscript checking will be performed.

For example:

SUBCHK

SUBCHK (ARRAY1,LIST2)

- TRACE

TRACE is used to indicate that label tracing is desired in the FORTRAN program or subprogram in which the DEBUG statement appears.

This option only enables label tracing.

Tracing will not actually be performed until a TRACE ON statement is encountered in the program flow. It is terminated upon encountering a TRACE OFF statement. (See 11.2.3.1.4.)

TRACE ON and TRACE OFF statements have no effect on a program unit in which the TRACE option has not been specified.

For example:

```
  DEBUG TRACE
C             The trace debug facility is enabled.
C             It can be activated with a TRACE ON statement.
```

- INIT

  INIT is used to trace the change in value of variables and arrays during execution.

  It has the form:

  INIT [(m [,m] ...)]

  where m is the name of a variable or array in the program unit for which a value trace is to be performed.

  If no list is given after the INIT option, a value trace is done on every variable or array in the program unit. This includes changes in value of any particular element of that array.

  The value trace consists of placing, in the debug output file, a display of the variable name or array element name along with its new value each time it is assigned a value in an assignment statement, a READ statement, a DECODE statement, or an ASSIGN statement.

  For example:

```
  DEBUG INIT (A,VAR1)
C               Starts debug facility and initiates trace of
C               array A and variable VAR1.
```

- SUBTRACE

  SUBTRACE is used to indicate entrance and exit of a subprogram during program execution.

  When the SUBTRACE option is included in the DEBUG statement within a function or subroutine, a trace on entrance to and exit from that subprogram is enabled.

  The name of the subprogram will be placed in the debug output file each time it is entered and the message "RETURN" will be put in the debug output file each time execution of that subprogram is completed.

  For example:

  DEBUG SUBTRACE

11.2.3.1.2. AT

The AT statement identifies the beginning of a debug packet and indicates the point in the program unit at which the packet is to be activated. It has the form:

AT s

where s is an executable statement number in the program or subprogram to be debugged.

There must be one AT statement for each debug packet. Each AT statement indicates the beginning of a new debug packet. The end of the debug packet must be indicated by an END statement if various operations are to be performed within the debug packet.

The operations specified within the debug packet are to be performed whenever s is encountered and prior to the execution of the statement associated with s.

For example:

```
...
DEBUG
AT 100
DISPLAY X,Y,A
END
```

} debug packet

11.2.3.1.3. TRACE ON

The TRACE ON statement initiates display of the flow of execution by statement number.

After TRACE ON has been encountered and until the next TRACE OFF is encountered, a record of the associated statement number is placed in the debug output file each time a labeled statement is encountered in the program unit.

TRACE ON remains in effect through any level of subprogram call or return. If the TRACE option has not been used on a DEBUG statement in a particular program or subprogram, label trace will not occur during execution of that program or subprogram.

TRACE ON may occur anywhere within a debug packet.

There can be no display of program flow by statement number within this program if the TRACE option was omitted from the DEBUG option list.

For example:

```
      DEBUG TRACE, INIT(A,B)
        .
        .
        .
      AT 104
      TRACE ON
C               The flow of execution will be displayed starting at
C               statement 104.
```

11.2.3.1.4. TRACE OFF

The TRACE OFF statement terminates statement label tracing.

It may occur anywhere within a debug packet. Tracing of program flow by statement number is terminated in the program unit by this statement.

For example:

```
        .
        .
        .
      DEBUG TRACE, INIT(A,B)
        .
```

```
      .
      .
      AT 104
      TRACE ON
      .
      .
      .
      AT 950
      TRACE OFF
C               The flow of execution will be displayed from the point
C               104 to the point where statement 950 is executed.
```

11.2.3.1.5. DISPLAY

The DISPLAY statement provides a simple debug output mechanism. It has the form:

```
DISPLAY list
```

where list is a series of variables, arrays with constant subscripts, or array names separated by commas. A formal parameter name of a function or subroutine is not permitted in list.

The DISPLAY statement is equivalent to the following RFOR or FOR V statements:

```
NAMELIST / name / list
WRITE (n,name)
```

where list is as defined above, name is a name generated for DISPLAY which is not a legal symbolic name, and n is the debug file reference number (from the UNIT option). DISPLAY provides a simple means of putting results of debugging operations for the program unit in the debug output file without needing FORMAT, NAMELIST, or WRITE statements. The output to the debug output file is in NAMELIST format. The DISPLAY statement may appear anywhere in a debug packet.

For example:

```
DISPLAY A,B,C,D (1,2) ,E
```

11.2.3.1.6. Debug Facility Example

The following is an example of FTN debug facility usage under CTS:

```
->FOR ASCII
ASCII FORTRAN PRESCAN 2R1A
>>OLD P2
>>LIST
100 10      FORMAT ('  THE VALUE OF I = ',I1)
110 20      I=7
120 30      WRITE (6,10) I
129         J = 9
130 40      CONTINUE
140         DEBUG TRACE
150         AT 20
160         TRACE ON
180         AT 40
```

```
190         TRACE OFF
195         DISPLAY I,J
200         END
END OF FILE
>>ASSUME CHECKOUT OFF
>>RUN
COMPILING...
DEBUG UNIT              -1
TRACE ON
TRACE               20
TRACE               30
 THE VALUE OF I = 7
TRACE OFF
$?
I =       7,J =             9
$END
*DIAGNOSTIC SCAN? >NO
>>
```

11.2.3.2. Eliminating Program Collection – ASSUME CHECKOUT

Syntax: ASSUME CHECKOUT [ON/OFF]

Abbreviation: A CHE

Function: To enable or disable the collection process for FTN programs.

When ASSUME CHECKOUT ON is keyed in, the collection of FTN programs is eliminated. Instead, program execution begins immediately following the compilation process. The mode CTS is initially in is ASSUME CHECKOUT ON.

The SPERRY UNIVAC Series 1100 ASCII FORTRAN compiler can be used as a compile-and-go processor by invoking the checkout mode of operation. ASSUME CHECKOUT ON directs the compiler to generate code into core and immediately execute it when compilation is complete. This mode results in increased throughput in cases where the object program is to be executed only once and when execution is relatively short. No relocatable element is produced.

To enable checkout mode:

```
->ASSUME CHECKOUT ON
->
```

11.2.3.3. Interactive Debugging Mode in the Checkout Compiler

FORTRAN checkout mode also provides a powerful interactive debugging system which is enabled through the use of the Z option. This allows the user to trace the execution, halt the execution, dump variable values, and perform other debugging activities.

If the Z option is used to enable debugging, the following command should be entered:

```
->ASSUME DEBUG ON
->
```

11.2.3.3.1. Entering Interactive Debug Mode

Interactive debug mode in the checkout compiler is entered:

- Before the first executable statement in the FORTRAN program. Debug mode is automatically entered at this point if the Z option is specified on the checkout compiler call.

- When a contingency interrupt occurs during execution of the FORTRAN program. The appropriate message is printed, and then the checkout contingency routine calls debug mode.

  A special case of contingency handling (see 11.2.3.5) occurs if the user enters:

  @@X C

  during execution of the program. In this case, assuming the Z option was specified, the current FORTRAN statement completes execution and then debug mode is entered (i.e., debug mode is not entered in the middle of a statement).

- When the FORTRAN program executes the statement CALL PAUSE. PAUSE has no parameters.

- When execution of the FORTRAN program has reached the END statement of the main program (i.e., just before termination of the program).

  Entry into debug mode at this point allows the user to dump the final values of variables (see 11.2.3.4.3), restore execution to a previous state (see 11.2.3.4.9), or dump the final contents of the program (see 11.2.3.4.12).

  The message:

  ```
  END PROGRAM EXECUTION
  ```

  is printed before debug mode is entered.

- When the special RESTART processor (FNTR) is invoked to reenter a previous debugging session (see SPERRY UNIVAC Series 1100 FORTRAN (ASCII), Programmer Reference UP-8244 (current version)).

- When a step break has been set at the current statement using the STEP command (see 11.2.3.4.12). The message:

  ```
  STEP BREAK AT LINE n
  ```

  is printed on entry to debug mode, where n is the current line number.

- When a line number break has been set at the current statement using the BREAK command (see 11.2.3.4.1). The message:

  ```
  BREAK AT LINE n
  ```

  is printed on entry to debug mode, where n is the current line number.

- When a statement label break has been set at the current statement using the BREAK command (see 11.2.3.4.1). The message:

  ```
  LABEL BREAK AT n
  ```

is printed on entry to debug mode, where n is the statement label associated with the current statement. Another line follows the above message, stating which program unit in the FORTRAN program contains the label.

- When the subprogram called by the CALL command (see 11.2.3.4.2) returns. The message:

  ENTER DEBUG MODE (RETURN FROM CALL COMMAND)

  is printed on entry to debug mode.

For the first five cases, the message:

ENTER DEBUG MODE AT LINE n

is printed on entry to debug mode. In the message, n is the line number of the statement where execution in the FORTRAN program was interrupted.

11.2.3.3.2. Soliciting Input

When the checkout compiler is in interactive debug mode, commands are solicited with the solicitation message:

C:

To leave debug mode, the GO, EXIT, and CALL commands are used. The debug commands are discussed individually in 11.2.3.4.

11.2.3.4. Debug Commands

All debug command names may be abbreviated to one letter (the initial one), except for SAVE, SNAP, CALL, and STEP, which may be abbreviated to the two-letter abbreviations CA, SA, SN, and ST, respectively; and SET BP, which may be abbreviated to SET B.

The following syntax rules apply for all debug commands:

- No blank characters are allowed inside a field of a debug command.

  The only exception to this rule occurs when a character variable is specified in the v subfield of the first field of the SET command. In this case, blanks may appear inside quotes in the character constant in the c field.

  This rule applies when a command contains a p subfield. This subfield, if specified in a command, is part of the first field of the command. Therefore, no blanks should appear before or after the slash (/) which separates the p subfield from the previous subfield, or before or after the colon (:) separator in the p subfield.

- Any number of blank characters (including zero) may appear between fields.

  The only command with more than one field is the SET command, which has three. All other commands have either one field or none.

- The *v* (variable name) subfield in the DUMP, SET, and SETBP commands must be one of the following:

  - scalar variable name (including a function subprogram entry point name)

- array name
- array element name (with constant subscripts)

Note that a scalar or array subprogram parameter may be specified using one of these forms.

The variable *v* must appear in an executable statement in the designated program unit *p*, unless *p* is the main program. If the COMPILER statement option DATA=AUTO or DATA=REUSE appeared in the program, then *v* must be a variable appearing in a COMMON block.

- The p (program unit) field in the PROG command and the p subfield in the DUMP, SET, SETBP, BREAK, CLEAR, and GO commands have the following format:

  progname [:extname]

  where progname represents the desired program unit in the ASCII FORTRAN program. It may be specified as (1) "*" (to represent the main program), (2) a FORTRAN subprogram (subroutine or function) name, or (3) an unsigned positive integer n (to represent the n^{th} block data program in the FORTRAN source program).

  The parameter extname represents the program unit name of the external program unit corresponding to the internal subprogram progname. Therefore, extname may be specified only if progname is specified as a subprogram name which represents a FORTRAN internal subprogram. The parameter extname may be specified as (1)"*" (if the external program unit is the main program) or (2) a FORTRAN subprogram name (if the external program unit is a subprogram).

  If progname is specified as a subprogram name and extname is not specified, then the external subprogram with name progname is taken. If no such external subprogram exists, then the first internal subprogram with name progname is taken.

11.2.3.4.1. BREAK

The BREAK command is used to set a breakpoint at a label or internal statement number (ISN). It has the format:

BREAK n[L [[/ p]] [, n [L [/p]]] . . .

where n is a positive integer, and p represents a program unit in the FORTRAN source program. The p subfield is described under syntax rules (see 11.2.3.4).

This command specifies a label or a line number as a point at which execution of the FORTRAN program is to be interrupted and interactive debug mode is entered. This point is referred to as a breakpoint.

If an L immediately follows n, then the breakpoint is the beginning of the FORTRAN statement with statement label n. The parameter p determines in which program unit in the FORTRAN symbolic element that the breakpoint is set. If p is not specified, then the breakpoint is label n in the program unit set by the PROG command. (See 11.2.3.4.10.)

If no L follows n, then the breakpoint is the beginning of the FORTRAN statement with line number n. Only line numbers which appear in the left column of the source listing may be used in a BREAK command.

For example, if the two commands BREAK 9 and GO are entered, then execution of the FORTRAN program will resume, and debug mode will be reentered before execution of the statement with ISN 9.

A maximum of eight label breaks and eight line number breaks may be set at any one time.

Two other debug commands are used in connection with the BREAK command. The CLEAR command is used to clear one or more breakpoints. The LIST command is used to list all breakpoints.

11.2.3.4.2. CALL

This CALL command calls a FORTRAN subprogram with the given arguments. It has the format:

CALL s[(a [,a]...)]

where s is a subprogram entry point name, and "a" is an actual argument that is passed to the subprogram.

This allows the user to test only a given subprogram without having to execute the entire FORTRAN program. For example, a subprogram could be repeatedly called with different sets of arguments.

The parameter s has the following format:

ent [:extname]

where ent is the entry point to be called; ent may be any entry point in any subprogram in the FORTRAN program, except for an alternate entry point (i.e., an entry point specified in an ENTRY statement) in an internal subprogram.

The parameter extname represents the program unit name of the external program unit corresponding to the internal subprogram ent. Therefore, extname may be specified only if ent is specified as an internal subprogram name. The parameter extname may be specified as (1)"*" (if the external program unit is the main program) or (2) a FORTRAN subprogram name (if the external program unit is a subprogram).

If extname is not specified, then the external subprogram entry point with name ent is taken. If no such external subprogram entry point exists, then the first internal subprogram with name ent is taken.

Each "a" is an actual argument and must match the corresponding formal parameter of s in type and usage. (In this way, the CALL command closely resembles a subprogram reference in a FORTRAN program.)

The parameter a must be specified in one of the following forms:

- A FORTRAN constant.
- A variable in program unit p, where p is the default program unit set by the PROG command. It must be specified as either a scalar variable name, an array name, or an array element name (with constant subscripts).
- A subprogram entry point name, immediately preceded by "*". If the entry point specified exists as an external subprogram entry point, then that one is taken. If no such external entry point exists, then the first internal subprogram with the specified name is taken.

Note that a statement number may not be passed as an actual argument via the CALL command. Therefore, a subprogram with any RETURN i statements (i.e., a subprogram with * as any formal argument) may not be called with this command.

A maximum of 20 arguments is allowed.

When the subprogram returns (via the RETURN statement), control is transferred back to interactive debug mode. The message:

ENTER DEBUG MODE (RETURN FROM CALL COMMAND)

is printed. In addition, if the subprogram called was a function, the message:

FUNCTION VALUE RETURNED:

is printed, followed by the actual value.

When debug mode is reentered on return from the CALL command, the user may not resume normal execution of the program (at the ISN where the CALL command was executed) using the GO command. Instead, the SAVE and RESTORE commands must be used for this purpose, since the CALL command interrupts normal execution.

For example, if the user wishes to execute a portion of the program, interrupt execution to test subprogram SUB (using the CALL command), and then resume normal execution of the program, the following commands could be entered:

```
SAVE
CALL SUB
RESTORE
```

11.2.3.4.3. CLEAR

The CLEAR command is used to clear breakpoints set by the BREAK command. It has the format:

```
CLEAR [ { n [ L [/ p] ] }
        { LABEL         }
        { LINE          }
        { BRKPT         }
        { ALL           } ]
```

where n is a positive integer, and p represents a program unit in the FORTRAN source program. The p subfield is described under syntax rules (see 11.2.3.4).

This command clears one or more breakpoints established by the BREAK command.

CLEAR n[L [/ p]] is the same as the BREAK command format. The parameter n may be immediately followed by L. This format clears a single statement label breakpoint in program unit p (if L is specified) or a single internal statement number break point.

The rest of the formats are used to clear one or both break lists or the SET BP break. CLEAR LABEL clears all label breakpoints CLEAR BRKPT clears the 1110 breakpoint register set by the SETBP command. CLEAR LINE clears all line number breakpoints. CLEAR and CLEAR ALL clears all label and line number breaks and the SETBP break.

LABEL, ISN, BRKPT, and ALL may be abbreviated to L, I, B, and A, respectively.

11.2.3.4.4. DUMP

The DUMP command is used to print the values of FORTRAN variables. It has the format:

```
DUMP [,opt]   { V [/p] }
              { /p     }
              {   !    }
```

where:

opt is an option letter; A or O is allowed.

V is the name of the variable that has been declared in the FORTRAN program. It must be specified in one of the following forms:

- scalar variable name
- array name
- array element name (with constant subscripts)
- function subprogram entry point name

  (Note that a scalar or array subprogram parameter may be specified using one of the first three forms.)

p represents a program unit in the FORTRAN source program. The p subfield is described under syntax rules (see 11.2.3.4).

This command prints the current value of one or more FORTRAN variables.

If the O option is specified on the DUMP command, the values are printed in octal format. If the A option is specified, they are printed in ASCII character format. If neither O nor A is specified, they are printed in a format corresponding to the variable's data type (INTEGER, REAL*4, COMPLEX*16, etc.).

Whenever the value of a variable is printed, it is preceded by a heading line in the format:

```
v  /p
```

where v is the variable name and p is the program unit name. (See the description of the p subfield in 11.2.3.4.)

If an entire array is dumped, the values of all elements in the array are printed in column-major order.

The formats are described as follows:

- DUMP [,opt] v[/p]

  This format prints the value of variable v in program unit p.

  If v is a scalar, array element, or function entry point, then one value is printed. If v is an array name, then the values of all elements in the array are printed.

If p is not specified, the variable v is taken from the default program unit set by the PROG command.

- DUMP [,opt] /p

  This format prints the values of all variables in program unit p.

- DUMP [,opt] !

  This format prints the values of all variables in all program units in the FORTRAN program.

If the second or third formats are used, then the order that the variables appear in the output is as follows:

- In a program unit, the variables are listed in alphabetical order.
- In the FORTRAN program (format 3), the program units are listed in the order that they appear in the source input.

11.2.3.4.5. EXIT

The EXIT command terminates the SPERRY UNIVAC Series 1100 FORTRAN (ASCII) processor, with a call to the FEXIT$ system routine. This routine terminates all input/output and does an ER EXIT$.

11.2.3.4.6. GO

The GO command resumes execution of an ASCII FORTRAN program. It has the format:

GO [n L [/p]]

where n is an positive integer, and p represents a program unit in the FORTRAN source program. The p subfield is described under syntax rules (see 11.2.3.4).

The GO command causes an exit from interactive debug mode; execution of the FORTRAN program is then resumed.

If 'GO' is specified (i.e., no command fields), then execution of the program continues at the point at which it was interrupted to go into debug mode.

If the n L [/p] field is specified, execution of the program continues at statement label n in program unit p. If p is not specified, the default program unit set by the PROG command is assumed.

The user should be cautious when specifying the n L [/p] format, since registers may not be set up correctly when jumping to a statement label. For instance, jumping to a label inside a DO loop or jumping to a label in another program unit (i.e., not the one currently being executed) could cause execution problems.

11.2.3.4.7. HELP

The HELP command (available in level 7 of FTN) prints information about debug commands, thereby about debug commands, thereby allowing the user to continue debugging without having to consult a manual about command descriptions or formats. It has the format:

$$\text{HELP} \left[\left[\text{[,opt]} \left\{ \begin{matrix} \text{ALL} \\ \text{cmd} \end{matrix} \right\} \right] \right]$$

where opt is an option letter (F or D is allowed), and cmd is one of the checkout debug command names. No abbreviations are allowed.

The format HELP lists all of the debug command names.

The format HELP cmd prints all available information about the designated debug command cmd, including a list of all formats, a description of the individual items specified in the formats, and a general description of the command.

The format HELP,opt cmd prints more specific information about debug command cmd. HELP,F cmd lists all command formats only. HELP,D cmd prints a command description only.

The format HELP ALL lists all available information for all debug commands. Note that a large amount of output is generated.

11.2.3.4.8. LINE

The LINE command prints the line number of the statement in the FORTRAN program where execution was interrupted to go into debug mode.

Line numbers are listed in the leftmost field of an ASCII FORTRAN source code listing.

11.2.3.4.9. LIST

The LIST command lists all breakpoints set by the BREAK command. This includes all statement label breaks and all line breaks.

The default program unit set by the PROG command is also listed.

11.2.3.4.10. PROG

The PROG command is used to set the default program unit for variables and statement labels. It has the format:

```
PROG p
```

where p represents a program unit in the FORTRAN source program. The parameter p is described under syntax rules (see 11.2.3.4).

This command sets the default program unit in the FORTRAN symbolic element that is implied for variables (in the DUMP, SETBP, and CALL commands) and statement labels (in the BREAK and CLEAR, and GO commands) to p. The following determines the default:

If no PROG command has been entered in debug mode during execution of the FORTRAN program, then the first program unit in the FORTRAN symbolic element is set as the default.

The default program unit set by this command may be overridden in an individual command (DUMP, SET, SETBP, GO, BREAK, or CLEAR) by specifying a program subfield p in that command.

The LIST command will print the default program unit set by the PROG command.

For example:

```
      PROG SUB1
C             Set subroutine or function SUB1 as the default program
C             unit.
      DUMP X
C             Print the value of variable X in the subprogram SUB1.
      DUMP X/2
C             Print the value of the variable X in the
C             block data program.
      BREAK 10L
C             Set a break at statement label 10 in subprogram SUB1.
      BREAK 10L/*
C             Set a break at statement label 10 in the main program.
```

11.2.3.4.11. RESTORE

The RESTORE command restores the state of the user's program which a previous corresponding SAVE command preserved (see 11.2.3.4), essentially restarting his program at the state it was in at the SAVE point. It has the format:

RESTORE [n]

where n is an integer consisting of 1 to 12 digits.

The optional version number n can be used to keep several stages of execution around when debugging.

When reentering the user program, the following message is printed:

ENTERING USER PROGRAM prog name [VERSION version-no]

Note the following limitation. The user is responsible for the assignment of files and their positioning. File contents, assignments, and positioning (tapes) are not saved or restored. Only the user's variables and point of execution are saved and restored, along with several debug mode parameters: breakpoints set by the BREAK and STEP commands, the default program unit set by the PROG command, and the trace mode value set by the TRACE command. Also, the same level of ASCII FORTRAN must have been used to do the corresponding SAVE command.

Examples:

```
@FTN,SCZ IN.ELT
@EOF
C             State is automatically saved into omnibus element
C             IN.ELT before entry to debug mode
      BREAK 7
      GO
C             FTN responds with BREAK AT ISN 7
      SAVE 2
C             State saved into omnibus element IN.ELT/2
      RESTORE
C             State restored from omnibus element IN.ELT
```

```
C                FTN responds with ENTERING USER PROGRAM       ELT
      BREAK 3
      GO
C                User program now restarts execution from the
C                original save point.
C                FTN responds with BREAK AT ISN 3
      RESTORE 2
C                State restored from omnibus element IN.ELT/2
C                FTN responds with ENTERING USER PROGRAM  ELT  VERSION:
2
      GO
C                User program resumes execution at ISN 7, where the
C                SAVE was done.

@FTN,NCZ.  IN.GAMES,SAVE.GAMES
C                The state is automatically saved in SAVE.GAMES before
C                entry to debug mode.
      GO
      ...
C                User program executes.
      ...
@FIN
      ...
C                Next day the user desires to do more testing on GAMES.
@RUN  ...
@FTNR  SAVE.GAMES
C                FTNR responds with a sign-on line and the state of
C                GAMES is restored from yesterday's save.
      GO
      ...
C                User GAMES program now executes again.
```

11.2.3.4.12. SAVE

The SAVE command is used to save the present state of the user's program for later resumption. It has the format:

SAVE [n]

where n is an integer consisting of 1 to 12 digits.

This command saves the present state of the user's program by writing it out to an omnibus element in his relocatable output (RO) file. The element name used is the RO element name. The version name used is either the user's RO version or the up-to-12-digit field on his SAVE command.

Only an all-digit field may be used on the SAVE command. Since the element created is typed as omnibus, this command does not destroy the user's symbolic, relocatable, or absolute elements of the same name in his RO file.

The RESTORE command may be used to restore the FORTRAN program to the state of execution of the last SAVE command. (See 11.2.3.4.11.)

An automatic SAVE command is done for the user just before initially entering debug mode after the END FTN message.

Examples:

```
@FTN,SCZ   IN.ELT
@EOF
SAVE
C              This will save state into omnibus element IN.ELT .

@FTN,NCZ   IN.ELT,OUT.ELT/TEST
@EOF
SAVE
C              This will save state into omnibus element OUT.ELT/TEST .

@FTN,NCZ   IN.ELT,OUT.ELT/TEST
@EOF
SAVE 99
C              This will save state into omnibus element OUT.ELT/99 .
```

11.2.3.4.13. SET

The SET command changes the value of a FORTRAN variable. It has the value:

```
SET  v [ /p ] = c
```

where:

- v is a name of a variable that has been declared in the FORTRAN program. It must be specified in one of the following forms:
  - scalar variable name
  - array element name (with constant subscripts)
  - function entry point name

  (Note that a scalar or array subprogram parameter may be specified using one of the first two forms.)
- p represents a program unit in the FORTRAN source program. The p subfield is described under syntax rules (see 11.2.3.4).
- c is a FORTRAN constant.

This command sets the value of variable v in the FORTRAN program to the constant e.

The parameter p determines which program unit in the FORTRAN symbolic element that v comes from. If p is not specified, then v is from the program set by the PROG command.

The parameter e must be the same data type as v. There are no conversions between data types for the SET command. For example, if v is declared as type complex*16 in program p, then e must be a double-precision complex constant.

The parameter c must be the same data type as v. There are no conversions between data types for the SET command. For example, if v is declared as type COMPLEX*16 in program p, then c must be a double-precision complex constant.

If v is a character variable, then c must be a character constant. Hollerith constants are not allowed.

11.2.3.4.14. SETBP

SETBP sets a breakpoint so that debug mode is reentered when a variable is set or referenced. It has the format:

SETBP [,opt] v [/p]

where:

opt is an option letter; R or W is allowed.

v is the name of a variable that has been declared in the FORTRAN program. It must be specified in one of the following forms:

- scalar variable name
- array element name (with constant subscripts)
- function subprogram entry point name

(Note that scalar or array subprogram parameters may be specified using one of the first two forms.)

p represents a program unit in the FORTRAN SOURCE program. The p subfield is described under syntax rules (see 11.2.3.4).

The SETBP command sets a breakpoint so that debug mode is reentered whenever the designated FORTRAN variable in program unit p is set or referenced during execution of a FORTRAN program. If p is not specified, v is taken from the program unit set by the PROG command.

The SETBP command may be used only if execution is on the SPERRY UNIVAC 1110, since the ER SETBP$ mechanism is used. This Executive Request sets the 1110 programmable breakpoint register, which causes a breakpoint interrupt whenever the specified condition is met. Checkout debug mode is reentered at the beginning of the next executable FORTRAN statement after the specified variable has been set or referenced.

If the R option is specified on the SETBP command, then debug mode is reentered whenever the designated variable v is read from storage. This occurs when the variable is referenced in an assignment statement (on the right side of the assignment "=" operator) or an I/O write statement.

If the W option is specified, the debug mode is reentered whenever the variable v is stored into. This occurs when the variable is set in an assignment statement (on the left side of the assignment '=' operator) or an I/O read statement.

If neither the R nor the W option is specified, both are assumed, i.e., debug mode is reentered whenever the variable is set or referenced.

Note that a breakpoint interrupt will occur whenever the storage that the variable occupies is involved in a load (R option) or store (W option) instruction. Therefore, the FORTRAN statement where the breakpoint interrupt occurs (i.e., the executable statement immediately preceding the statement where debug mode is reentered) may not actually reference the variable name specified in the SETBP command; the interrupt may have been caused by the setting or referencing of a variable that occupies the same storage as the variable designated in the command. Variables which may be

overlapped in storage in a FORTRAN program include those used in EQUIVALENCE or COMMON statements or those passed as subprogram parameters.

The breakpoint set by the SETBP command will remain in effect during execution until it is cleared by the CLEAR command (either CLEAR or CLEAR BRKPT format).

11.2.3.4.15. SNAP

The SNAP command dumps all or part of the FORTRAN program. The Executive routine ER SNAP$ is used to dump the contents of one location counter at a time. It has the format:

SNAP [{ I / D / R }]

The SNAP format dumps the entire FORTRAN program. The SNAP I format dumps contents of location counter 1 of the program. $(1) contains all program instructions not resulting from input/output lists. The SNAP D form dumps the contents of all location counters except 1. The SNAP R format causes all registers to be dumped.

Since ER SNAP$ lists absolute addresses only, an L option checkout compiler listing of the FORTRAN program may be helpful if the user wishes to decode the information that is dumped. This listing includes the location counters and relative addresses for variables and code in the program.

11.2.3.4.16. STEP

The STEP command is used to set a breakpoint at a certain point ahead in the program. It has the format:

STEP [n]

where n is a positive integer.

The STEP command specifies a breakpoint at which execution of the FORTRAN program is to be interrupted and interactive debug mode reentered.

After the GO command is entered, n FORTRAN statements are executed and then debug mode is reentered. If n is omitted, one is assumed.

For example, if the two commands STEP 3 and GO are entered, then execution of the FORTRAN program will resume. After three FORTRAN statements have been executed, debug mode will be reentered.

11.2.3.4.17. TRACE

Either TRACE or TRACE ON turns trace mode on. TRACE OFF turns trace mode off. It has the format:

TRACE [{ ON / OFF }]

If trace mode is on, then a message in the form:

```
LINE n
```

is printed at the start of execution of each FORTRAN statement, where n is the internal statement number of the statement. Trace mode is initially off.

11.2.3.4.18. WALKBACK

The WALKBACK command is used to trace the general flow of program execution through FORTRAN subprograms.

It gives a step-by-step trace of FORTRAN subprogram references that have occurred during program execution. The trace begins at the current statement in the subprogram which is executing (i.e., the point in the user program at which execution was interrupted to go into debug mode) and ends at the main program.

During execution of the walkback trace, one line is printed at each step indicating which FORTRAN subprogram (subroutine or function) was referenced at a certain line number of another subprogram (or the main program). Walkback may occur over any number of subprograms.

Any FORTRAN subprogram named in a walkback message will be a main entry point, regardless of which entry point in that subprogram was actually referenced.

For example:

```
110    I = 5
120    CALL S(I)
130    END

140    SUBROUTINE S(I1)
150    J = F(I1)
160    PRINT * ,J
170    RETURN
180    END

190    FUNCTION F(I2)
200    F = 12**3
210    RETURN
220    END
```

Assume that a break is set in the above program at internal statement number 11 during checkout execution (using the debug command BREAK 11). When debug mode is reentered at line 11, execution of the WALKBACK command causes the following lines to be printed:

```
WALKBACK INITIATED AT ADDRESS 031470 IN USER PROGRAM
THIS ADDRESS IS AT LN.    11 OF F
F     REFERENCED AT LN.     5 OF S
S     REFERENCED AT LN.     2 OF MAIN PROGRAM
```

11.2.3.4.19. Interactive Debugging Example

```
-> FOR ASCII
ASSUME ASCII ON
ASCII FORTRAN PRESCAN 2R1A
```

```
>>ASSUME DEBUG ON
>>ASSUME CHECKOUT ON
>>OLD EXP
>>LIST
100 10     FORMAT (' THIS IS LINE ONE ')
110 20     FORMAT (' THIS IS LINE TWO ')
120 30     FORMAT (' THE VALUE OF J =  ',I2)
122        WRITE (6,10)
124        WRITE (6,20)
130        DO 50 I=1,3
140        J=I*3
148 40     K=J**2
150        WRITE(6,30) J
160 50     CONTINUE
170        END
END OF FILE
>>RUN
FTN 9R1  *03/24/77-08:45(0,)

END FTN 43 IBANK 44 DBANK
ENTER DEBUG MODE AT ISN    4
C:>BREAK 40L/*
C:>GO
THIS IS LINE ONE
THIS IS LINE TWO
LABEL BREAK AT   40L
  IN MAIN PROGRAM
C:>LINE
LINE    8
C:>LIST
PROG:MAIN PROGRAM
LINE:  NONE
LABELS:
 40L
  IN MAIN PROGRAM
C:>DUMP I
I       /*
000000                1
C:>DUMP J
J       /*
000000                3
C:>DUMP K
K       /*
000000                0
C:>GO
THE VALUE OF J = 3
LABEL BREAK AT   40L
  IN MAIN PROGRAM
C:>DUMP I
I       /*
000000                2
C:>DUMP J
J       /*
000000                6
C:>DUMP K
```

```
K       /*
000000                  9
C:>GO
THE VALUE OF J = 6
LABEL BREAK AT    40L
  IN MAIN PROGRAM
C:>DUMP I
I       /*
000000                  3
C:>DUMP J
J       /*
000000                  9
C:>DUMP K
K       /*
000000                 36
C:>GO
THE VALUE OF J = 9
END PROGRAM EXECUTION
ENTER DEBUG MODE AT ISN    11
C:>DUMP I
I       /*
000000                  5
C:>DUMP J
J       /*
000000                 12
C:>DUMP K
K       /*
000000                144
C:>GO
>>
```

11.2.3.5. Contingencies and Restrictions in Checkout Mode

Most contingencies are captured by the compiler even if the Z option is omitted. After an appropriate message, the interactive debugger will solicit commands. The user may then attempt to find the problem by dumping variables. Most contingencies are such that the user program cannot continue execution. In debugging mode, however, a BREAK keyin (@@X C) causes temporary suspension of the user program. The user may continue the program execution by typing GO to the interactive debugger. In this way, he may interrupt infinite loops in his program and probe for their cause using interactive commands.

Due to the generation of code in core, only simple program structure can be provided. Links cannot be generated to subprograms that are not physically in the source program. In addition, multibanking and segmentation are not provided (e.g., the BANK and COMPILER statements cannot be used). The maximum size for a user program is somewhat smaller due to the addressing space used by checkout execution time requirements. A warning message will be printed if this maximum size is exceeded.

11.2.3.6. Walkback and the Interactive Postmortem Dump

When a contingency interrupt occurs during execution of an ASCII FORTRAN program, the following debugging aids will automatically be executed:

- the ASCII FORTRAN walkback process (FTNWB)
- the ASCII FORTRAN interactive postmortem dump (FTNPMD)

The walkback mechanism gives a step-by-step trace of FORTRAN subprogram references that have occurred during program execution, from the point of the error condition back to the main program. Only subprograms in relocatable elements generated by F option ASCII FORTRAN compilations can be traced by the walkback process.

FTNPMD allows the user to interactively dump the current values of FORTRAN variables in the executing program. The variables which may be dumped are those which exist in elements which were generated with F option ASCII FORTRAN compilations. If the program is running in batch mode (see 13.1), FTNPMD will be executed only if the F option was specified on the @XQT control card.

In addition to being called by the contingency routine, both FTNWB and FTNPMD may be initiated by calls from the user program.

For more information on walkback, interactive PMD, and interactive debugging, see SPERRY UNIVAC Series 1100 FORTRAN (ASCII) Programmer Reference, UP-8244 (current version).

11.2.4. RFOR

Symbolic level debugging in RFOR is very similar to that in BASIC. The syntax of the commands is identical. There are some differences in the messages and operations, however. (For example, since two variables can have the same name in different subroutines in RFOR, these must be distinguished in some way.)

The ASSUME DEBUG ON/OFF command has the same effect in RFOR as it has in BASIC (see 11.2.1).

11.2.4.1. PAUSE

The execution of a RFOR program may be halted at any point to examine variables with the statement:

n PAUSE

at the specified point. Notice that *n* is a CTS line number. If the ASSUME DEBUG is on when this statement is executed, FORTRAN prints the message:

PAUSE n IN d

where n is the line number in octal and d is the program element name.

For example:

```
->FORTRAN F
FD FORTRAN 5R1
>>NEW P1
>>N
100 >10 FORMAT (' LINE ONE')
110 >WRITE (6, 10)
120 >PAUSE
```

```
130 >20 FORMAT (' LINE TWO')
140 >WRITE (6,20)
150 >END
```

With ASSUME DEBUG OFF (see 11.2.1), the PAUSE statement has no effect:

```
160 >*RUN
DO YOU WANT A GLOBAL SCAN?  >YES
COMPILING...
 LINE ONE
PAUSE          . TYPE S TO RESTART. >S
 LINE TWO
NORMAL EXIT.  EXECUTION TIME:              5 MILLISECONDS.
*DIAGNOSTIC SCAN?   >NO
```

After an ASSUME DEBUG ON, however, the following results:

```
>>ASSUME DEBUG ON
>>RUN
DO YOU WANT A GLOBAL SCAN?  >NO
COMPILING...
 LINE ONE

PAUSE    120 IN   NAME$
COMMAND? > >RESUME
 LINE TWO
NORMAL EXIT.  EXECUTION TIME:              5 MILLISECONDS.
*DIAGNOSTIC SCAN? >NO
```

The PAUSE statement causes the following message to be printed:

```
PAUSE n IN d
COMMAND? > >
```

In the preceding example the word *RESUME* was typed as an answer to the question, COMMAND?. Any of the following may be transmitted as responses to the COMMAND? question.

PRINT vl	Print value of variable list vl.
SET v=x	Set variable v to a new value where the new value x is of the same type (REAL, INTEGER, ...) as v.
RESUME	Resume execution.
STOP	Stop execution of program.
DUMP	Stop execution of program and provide a register contents dump to the terminal.
CHANGE	Solicit new trace options as though the first TRACE ON statement is encountered (see 11.2.4.3).
DEBUG ON	Reinstate the execution of the TRACE ON and PAUSE statements. This command is used in conjunction with the break key following the use of the DEBUG OFF command. Program execution trace output resumes when a TRACE ON statement is encountered (see 11.2.4.2 and 11.2.4.3).

DEBUG OFF Terminate all program execution trace output and disregard all following TRACE ON and PAUSE statements.

If TRACE output to the file TRACE$ (see 11.2.4.3) is also in effect, this information is written in the trace file, as well as typed on the terminal. After processing all but the STOP, DUMP, CHANGE, and RESUME commands, the executing program again solicits input with the statement:

```
NEXT? >
```

The following are examples of the acceptable commands:

```
>>P A
80        DO 30 I=1,3
90        J=2*I
100 10     FORMAT (' LINE ONE')
110        WRITE (6,10)
120        PAUSE
130 20     FORMAT (' LINE TWO')
140        WRITE (6,20)
145 30     CONTINUE
150        END
END OF FILE
>>RUN
DO YOU WANT A GLOBAL SCAN? >YES
COMPILING...
 LINE ONE

PAUSE  120   IN   NAME$

COMMAND? > >PRINT I

                    I =                 1

               NEXT? > >PRINT J

                    J =                 2

               NEXT? > >SET J=40
               NEXT? > >PRINT J

                    J =                 40

               NEXT? > >PRINT J,I

                    J =                 40        I =          1

               NEXT? > >RESUME
 LINE TWO
 LINE ONE

PAUSE  120   IN   NAME$
COMMAND? > >STOP
 TRACE EXIT.  EXECUTION TIME:              31 MILLISECONDS
*DIAGNOSTIC SCAN?  >NO
```

11.2.4.2. BREAK

If ASSUME DEBUG is ON (see 11.2.1) depressing the BREAK key can effect an orderly break in program execution at any time. Depressing this key immediately causes output to cease (even in the middle of a line), and the following message appears:

```
*OUTPUT INTERRUPT
```

The system returns without a solicitation character. Enter:

```
@@X C
```

If the terminal was in the process of printing a line when the break key was depressed, the system responds by printing that line again. Since the printing of the output may be slower than the execution of the program, other output lines, which were queued before the @@X C was entered, are printed. In fact, if the program is small it may complete before an @@X C can be entered. Assuming this is not the case, the message:

```
BREAK AT LINE NUMBER: xxx
COMMAND? >>
```

is printed. One of the commands described in 11.2.2.1 can be entered.

If a break is desired and the output is not needed, then enter:

```
@@X CO
```

If the terminal was in the process of printing a line when the interrupt key was depressed, the system responds by printing that line again. The O option causes all other output up to and including the COMMAND? query to be discarded. The system will respond with a cursor. A command described in 11.2.2.1 can then be entered.

For example:

```
>>NEW P2
>>N
100 >DO 30 I=1, 100000
110 >10 FORMAT (' LINE 110')
120 >WRITE (6,10)
130 >A=A+1
140 >30 CONTINUE
150 >END
160 >*RUN
DO YOU WANT A GLOBAL SCAN?  >YES
COMPILING...
  LINE 110
  LINE 110
  LINE 110
  LINE 11

*OUTPUT INTERRUPT*
@@X OC
  LINE 110
>
COMMAND? > > PRINT I        I =           650
```

```
                NEXT? > >PRINT A
                          A = 649.00000
                NEXT? > >PRINT A, I
                          A = 649.00000          I =            650
                NEXT? > >SET A=7000.
                NEXT? > >RESUME
  LINE 110
  LINE 110
  LINE 110

*OUTPUT INTERRUPT*
@@X OC
  LINE 110
>
COMMAND? > >DEBUG OFF
                NEXT? > >RESUME
  LINE 110
  LINE 110
  LINE 110

*OUTPUT INTERRUPT*
@@X OC
  LINE 110
>
COMMAND? > >SET B=4
ILLEGAL VARIABLE NAME.
                NEXT? > >STOP
  TRACE EXIT. EXECUTION TIME:          2412 MILLISECONDS.
*DIAGNOSTIC SCAN? >NO
```

11.2.4.3. TRACE

The logic flow of a RFOR program may be determined by bracketing the block of statements to be traced with TRACE ON/OFF statements as follows:

```
n1   TRACE ON
 .
 .
statements to be traced
 .
 .
n2   TRACE OFF
```

Notice that n1 and n2 are CTS line numbers. These TRACE statements may have RFOR statement numbers.

There is a minor difference between RFOR and BASIC concerning the initial execution of the TRACE. In BASIC, all lines are numbered and, therefore, all lines may be the object of GOTO statements. This is not true in RFOR. Thus, the TRACE ON statement is treated as an executable statement. That is, for the TRACE to be effective, the TRACE ON statement itself must be executed, not just any statement within the TRACE ON/OFF block. Thus, in RFOR any transfer into the TRACE block does not have the trace effect.

When the TRACE ON statement is executed, the following messages appear:

```
TRACE n IN d

OUTPUT TO FILE? > >
```

where n is the line number (octal) of the TRACE statement, and d is the name of the program element.

A *YES* answer causes subsequent trace information to be sent to the file TRACE$. A *NO* answer causes the trace information to be sent back to the terminal.

This is followed by the question:

```
LINE NOS ONLY?  > >
```

A *YES* answer causes execution to resume with only the line numbers as trace output. A *NO* answer causes the message:

```
ALL VARIABLES?  > >
```

A *YES* answer causes execution to be resumed with a trace of all program variables. A maximum of 100 variables may be displayed. A *NO* answer causes the question:

```
WHICH VARIABLES? > >
```

Answer this with the names of up to ten variables separated by commas. RFOR resumes execution after this response.

Notice that, unlike BASIC, RFOR does not offer the question:

```
ONLY WHEN CHANGED?
```

Prior to the execution of each statement in the specified block, RFOR prints:

```
TRACE n IN d
```

where n is the line number (octal) and d is the program element name, then all variables, specified variables, or no variables are printed according to specifications. Notice the following example:

```
>>P A
100 10     FORMAT (' LINE ONE')
110        WRITE (6,10)
120        TRACE ON
130        DO 30 I=1,3
140 20     FORMAT (' LINE TWO')
150        WRITE (6,20)
160 30     CONTINUE
170        END
END OF FILE
>>ASSUME DEBUG ON
>>RUN
DO YOU WANT A GLOBAL SCAN?    >YES
COMPILING...
 LINE ONE
R73R1Q TRACE 120 IN   NAME$
```

```
OUTPUT TO FILE? >    >NO
LINE NOS ONLY? >    >YES

 LINE TWO
 LINE TWO
 LINE TWO
NAME$    130    140    150    130    140    150    130    140    150    160
NORMAL EXIT.  EXECUTION TIME:              15 MILLISECONDS.
*DIAGNOSTIC SCAN?  >NO
```

In this case only line number tracing is selected. Notice the trace messages following the output.

If DEBUG is turned OFF, TRACE statements have no effect. This can be accomplished through the ASSUME DEBUG OFF (see 11.2.1) or by answering DEBUG OFF to a BREAK or PAUSE command (see 11.2.2.1 or 11.2.4.1).

The following example shows the tracing of the symbolic variables:

```
>>P A
100 10     FORMAT (' LINE ONE')
110        WRITE (6,10)
120        TRACE ON
130        DO 30 I=1,3
135        A=A+1
140 20     FORMAT (' LINE TWO')
150        WRITE (6,20)
160 30     CONTINUE
170        END
END OF FILE
>>ASSUME DEBUG ON
>>RUN
DO YOU WANT A GLOBAL SCAN?  >YES
COMPILING...
 LINE ONE

R73R1Q TRACE 120 IN    NAME$
OUTPUT TO FILE?  >    >NO
LINE NOS ONLY?   >    >NO
ALL VARIABLES?   >    >YES
NAME$     190
                          I =          0          A = .00000000
NAME$     130
                          I =          1          A = .00000000
NAME$     135
                          I =          1          A = 1.0000000
NAME$     140
                          I =          1          A = 1.0000000
 LINE TWO
NAME$     150
                          I =          2          A = 1.0000000
NAME$     130
                          I =          2          A = 1.0000000
NAME$     135
                          I =          2          A =
```

```
*OUTPUT INTERRUPT
@@X OT
                                I =          2          A = 2.0000000
*BREAK
->
```

Any number of blocks of a program may be traced by bracketing those blocks with TRACE ON/OFF instruction pairs. If the final TRACE OFF is omitted, then tracing is effective through the end of the program. To change the trace data specifications (add a variable to the list or change from line numbers only to tracing variables), use the CHANGE command (see 11.2.4.1) in response to a PAUSE or BREAK sequence. Notice that this is somewhat different from the BASIC procedure. This may be seen in the following example:

```
>>P A
100 10    FORMAT (' LINE ONE')
110       WRITE (6,10)
120       TRACE ON
130       DO 30 I=1,3
135       A=A+1
137       PAUSE
140 20    FORMAT (' LINE TWO')
150       WRITE (6,20)
160 30    CONTINUE
170       END
END OF FILE
>>RUN
DO YOU WANT A GLOBAL SCAN? >NO
COMPILING...
 LINE ONE

R73R1Q TRACE 120 IN   NAME$

OUTPUT TO FILE?  >   >NO
LINE NOS ONLY?  >   >YES

NAME$     130

PAUSE       137 IN    NAME$

COMMAND?   >  >CHANGE
OUTPUT TO FILE? >   >NO
LINE NOS ONLY? >   >NO
ALL VARIABLES? >   >YES
NAME$     137
                                I =          1          A = 1.0000000

NAME$     140
                                I =          1          A = 1.0000000
 LINE TWO
NAME$     150


*OUTPUT INTERRUPT*
@@X
                                I =          1          A = 1.0000000
```

```
*BREAK
->
```

Following program execution termination, if trace output were to file TRACE$, output file action would be solicited via:

```
SEND OUTPUT TO SITE? > >
```

A *YES* answer solicits an end-of-trace output message to be inserted in file TRACE$ via:

```
MSG: >
```

The trace output is then directed to an onsite printer. Regardless of the answer, the file TRACE$ can be accessed with CTS. Use the commands:

```
>>USE SQUELCH$, TRACE$
>>SCAN
```

For example:

```
>>P A
100 10     FORMAT (' LINE ONE')
110        WRITE (6, 10)
120        TRACE ON
130        DO 30 I=1,3
135        A=A+1
140 20     FORMAT (' LINE TWO')
150        WRITE (6, 20)
160 30     CONTINUE
170        END
END OF FILE
>>ASSUME DEBUG ON
>>RUN
DO YOU WANT A GLOBAL SCAN? >NO
COMPILING ...
 LINE ONE

R73R1Q TRACE 120 IN   NAME$
OUTPUT TO FILE? >  >YES
LINE NOS ONLY? >  >NO
ALL VARIABLES? >  >YES
 LINE TWO
 LINE TWO
 LINE TWO
SEND OUTPUT TO SITE? >  >NO
NORMAL EXIT.  EXECUTION TIME:        60 MILLISECONDS.
*DIAGNOSTIC SCAN?  >NO
>>USE SQUELCH$, TRACE$
>>SCAN
<185> INCOMPLETE PRINTFILE
```

```
>>P 1,5
1 R73R1Q RFOR TRACE OUTPUT 02/25-16:10
2 NAME$     120
3                              I =          0          A = .00000000
4 NAME$     130
5                              I =          1          A = .00000000
>>T LNG ( )
29
>>P 25, 30
25                             I =          3          A = 3.0000000
26 NAME$     150
27                             I =          3          A = 3.0000000
26 NAME$     160
29                             I =          3          A = 3.0000000
END OF FILE
>>
```

Notice that after the SCAN command any editing commands reference the TRACE information. Here the LNG function is used to determine the length of the file. Use the EDIT command (see 11.1.2) to reestablish the previous program from the working area.

12. Desk Calculator

12.1. Expressions

CTS can compute the value of complex expressions including most common mathematical and trigonometric functions, CTS variables, integer, real, and string values. The resulting values can be formatted using CTS editing functions. They are automatically concatenated when placed in the TYPE command for printing. There also is a command which prints a table of expression values which are computed by reevaluating the expression while iterating variables over a specified range of values. Another command prints the summation values of this table.

The simplest way of using CTS as a desk calculator is to use the TYPE command to print an expression value as follows:

```
->TYPE 12.0+SQR(10)/152.1
12.020790780145749
```

An expression is composed of a series of terms and operators.

12.1.1. Integer Constants

An integer constant is a sequence of 1 to 18 decimal digits. It must be preceded by a minus sign in order to be a negative integer. Positive constants may be preceded by a plus sign. An integer without a sign is assumed to be positive. Here are some examples of integer constants:

```
1
0
-1
+32768
-4096
```

12.1.2. Real Constants

Real constants are distinguished from integer constants by the decimal point and the optional power of 10. Generally a real constant is represented by an integer constant, followed by a decimal point, followed by the fractional part of the constant. This is sometimes followed by the letter E and the power of ten which is represented by an integer constant. There may be up to 18 digits in the integer and fractional parts of a number. The power of ten must be between -308 and +308.

Here are some permissible real constants:

```
1.13
-.333333333333333333
+4096.
10.E50        (equivalent to 10^51)
.1E-100       (equivalent to 10^-10)
0.0E50        (equivalent to 0)
```

12.1.3. String Constants

A string constant is a sequence of characters delimited by the single quote character. Thus:

```
'(-B+SQR(B*B-4.0*A*C))/(2*A)'
```

is a string constant. A variable can be set equal to a string, evaluating the string as an expression.

12.1.4. Functions

As with constants, there are two types of built-in functions in CTS. These functions operate on zero, one, or more arguments and produce a value. The value produced may be either an arithmetic value or a string value. The functions are distinguished by the type of value they produce.

A function appearing in an expression may be thought of as being reduced to its equivalent. For example: SQR(4) appearing in an expression is equivalent to 2. Consequently, functions may be nested. For example:

```
->T   SQR(SQR(ABS(-4)))
1.414213562373095
->
```

Some functions do not require an argument. Nevertheless, parentheses must appear in all function calls. This is in order to distinguish the function from a simple variable name with the identical spelling. For example, the LNG function, which has no argument, must be written as LNG( ) to distinguish it from a variable LNG.

Generally arithmetic functions accept arguments of type real or integer and return a real value. Individual exceptions to this rule are shown in the Tables 12-1 and 12-2.

Table 12-1. Numeric Functions

Function Reference	Numeric Value Returned	Examples
ABS(x)	The absolute value of x.	-> *T ABS(7.5)* 7.5 -> *T ABS(-7.5)* 7.5 ->
ATN(x)	*The arctangent of x in the range of [-pi/2,pi/2]. x may be any value.	-> *T ATN(1.2)* 8.760580505981934E-1 ->
C( )	The number of the left column of the last find in execution of the LOCATE command.	-> *100 ABCDEFGHIJKL* -> *LOC 'DEF' 100* 100 ABCDEFGHIJKL -> *T C( )* 4 -> *LOC 'GHI' 100* 100 ABCDEFGHIJKL -> *T C( )* 7 ->
COL( )	The column position as an integer within a string expression.	-> *SET S1='ABCDEF' COL( ) 'DEF'* -> *T S1* ABCDEF7DEF -> *T 'AB' TAB(4) 'CD'* AB CD -> *T 'AB' TAB(COL( )+4) 'CD'* AB CD ->
COS(x)	*The cosine of x in the range of [-1,1]. $\|x\| < 2^{56}$.	-> *T COS(1.2)* 3.623577544766735E-1 ->
COT(x)	*The cotangent of x (any value). $\|x\| < 2^{56}$ and not equal 0.	-> *T COT(1.2)* 3.887795693682049E-1 ->
CR( )	The number of the right column limit of the last find in execution of the LOCATE command.	-> *100 ABCDEFGHIJKL* -> *LOC 'DEF' 100* 100 ABCDEFGHIJKL -> *T CR( )* 6 ->
EXP(x)	*The exponential (e^x) of x (any value). X<709.089.	-> *T EXP(1.2)* 3.3201169227365475 ->

Table 12–1. Numeric Functions (continued)

Function Reference	Numeric Value Returned	Examples
LEN(s)	The number of characters currently in string s. Zero is returned if s is null.	*-> SET B='ABC'* *-> T LEN(B)* 3 ->
LGT(x)	*The base 10 logarithm of x (any value). x > 0	*-> T LGT(1.2)* 7.918124604762483E–2 ->
LNG( )	The number of lines currently in the working area f.	*-> T LNG( )* 7 ->
LOG(x)	*The natural logarithm of x (any value). x > 0	*-> T LOG(1.2)* 1.823215567939546E–1 ->
NUM(n,c1,c2)	The numeric value of the strings specified by column limits c1 and c2 within the line of f specified by n. If n is 0, then the current line of f is implied. If n is a string, the substring denoted by c1 and c2 is converted to a number.	*-> SET C='ABC123'* *-> T NUM(C,4,6)* 123 *-> P 120* 120 JUMP 20 NO FIND *-> T 4+NUM(120,6,7)* 24 ->
OCC( )	The number of lines the string occurred in after a LOCATE or FIND command or the number of occurrences of the string after a CHANGE command is printed.	*-> TYPE OCC( )* 4
P( )	The current position of the line pointer of the working area f.	*-> T P( )* 120 ->
SIN(x)	*The sine of x in the range of [–1,1]. $\|x\| < 2^{56}$.	*-> T SIN(1.2)* 9.320390859672264E–1 ->
SQR(x)	The square root of x. x must be greater than or equal to zero.	*-> T SQR(1234)* 35.128336140500592 ->
TAN(x)	*The tangent of x (any value). $\|x\| < 2^{56}$.	*-> T TAN(1.2)* 2.5721516221263189 ->

* *For a full explanation of these functions, see SPERRY UNIVAC Series 1100 Mathematical Function Library, Programmer Reference, UP–8007 (current version).*

Table 12-2. String Functions

Function Reference	String Value Returned	Examples
APF( )	The current name of the assumed program file F. This file is initially proj-id*run-id but may be changed by an ASSUME PROGRAM or an ASSUME FILE command.	-> *T APF( )* KMB ->
ASC( )	A string which indicates if the working area is ASCII or Fieldata	-> *T ASC( )* ASCII OFF -> *ASSUME ASCII ON* -> *T ASC( )* ASCII ON ->
COMP( )	A string containing the name and options for the assumed compiler.	-> *ASSUME COMPILER RFOR* -> *T COMP ( )* RFOR,RS ->
CSF(e)	The CSF function passes to the Executive control language statements via CSF$ (see SPERRY UNIVAC Series 1100 Executive System, Vol. 2 EXEC, Programmer Reference, UP-4144.2 (current version)). The expression e is a string expression representing the control statement or a variable that has been previously set to that string value. A status code from the operating system is returned.	-> *CSF (@ASG,UP BREAK.)* ->
DATE( )	A string containing the date and time of day in the following form: dd mon yy hh:mm:ss	-> *T DATE( )* 01 APR 77 15:43:37 -> *T TXT(DATE( ),1,9)* 01 APR 77 -> *T TXT(DATE( ),14,21) ON ' TXT(DATE( ),1,9)* 15:48:00 ON 01 APR 74 ->
DKN( )	The current name of the working area, f.	-> *NEW ABC* -> *TYPE DKN( )* ABC ->

Table 12-2. String Functions (continued)

Function Reference	String Value Returned	Examples
FILE('s')	The fully qualified file name associated to the internal file name provided as the function's string argument. If the desired file is not currently assigned, then a null string is returned as the value of the function.	-> *CRE,A ABC*DEF.* *CRE,A ABC*DEF. -> *USE X,ABC*DEF* -> *TYPE FILE('X') ',' FILE('DEF')* ABC*DEF(1),ABC*DEF(1) -> *REL X* -> *TYPE FILE ('X') ',' FILE('DEF')* , ->
FMT(x,w,d ['s'])	A string containing the numeric expression x converted as directed by the remaining arguments. The primary use of FMT is for building an output line. The arguments have the following meanings:	-> *SET A=10123.45* -> *SET B=* *-.000987654321*
	w – The total number of characters (field width including nonnumerics) returned by FMT. If w=0, the converted number is left justified and of variable length. If w is not 0, it must be large enough to contain the entire field; that is, this function will not truncate a field.	-> *T FMT(A,15,2,'*$,')* *****$10,123.45 -> *T FMT(A,5,2)* <24> STRING EXCEEDS COLUMN LIMITS. ->
	d – If d>0, then d decimal digits to the right of the assumed decimal point will be converted.	-> *T FMT(A,0,2,'*$,')* $10,123.45 -> *T FMT(B,0,6)* -0.000988 ->
	If d=0, then only the integer part of x is printed and no decimal point is printed.	-> *T FMT(A,0,0)* 10123 ->
	If d<0 then |d| decimal digits are converted in scientific notation. That is, the converted number consists of a fractional part and a power of 10 exponent.	
	s – is a string consisting of from one to three characters enclosed in quotation marks: 'ABC'.	-> *T FMT(B,17,-8)* -9.8765432E-4 ->
	A is the character to be filled in on the left of the field (for example, the asterisk). The assumed value of A is a blank.	-> *T FMT(A,10,0,'*$,')* ***$10,123 ->

Table 12-2. String Functions (continued)

Function Reference	String Value Returned	Examples
	B is a prefix character. This character will be printed in the prefix or sign position unless the field is negative. If the field is negative a minus sign will appear in the prefix position. For example, B could be the $.	*-> T FMT(B,17,12,' ! ')* -0.00098 76543 21 *-> T FMT(-B,17,12,' ! ')* !0.00098 76543 21 *-> T FMT(B,17,12,' + ')* -0.00098 76543 21 ->
	C specifies a digit group separator. If C is a comma, then every three digits of the number are separated by a comma. If C is any other character, then every five digits are separated by that character. If no character is specified, no grouping will occur.	*-> T FMT(A,0,-7,' + ')* +1.01234 5E4
ID('s')	A string containing run-related information as specified by the input argument string. The valid input arguments and their function values are: GRUN – The unique run-id as generated by the operating system. This value is usually the user-specified run-id from the @RUN Executive command, but it may vary under certain conditions. For a more detailed explanation of those conditions, see SPERRY UNIVAC Series 1100 Executive System, Volume 2 EXEC, Programmer Reference, UP-4144.2 (current version). RUN – The user-specified run-id from the @RUN command. PROJ – The user-specified project-id from the @RUN command. IDEN – The user-specified run-id, unless the F-option was specified on the @CTS initiation command. If the F-option was used, then the value returned is the unique identifier entered by the user. In this way, a subroutine can obtain the default assume program file at CTS initiation time.	(Assume the Executive control statement is: @RUN JONES,555555,JAY) *-> TYPE ID('PROJ')* JAY *-> TYPE ID('RUN')* JONES *-> TYPE ID('GRUN')* JONES

Table 12-2. String Functions (continued)

Function Reference	String Value Returned	Examples
	ACCT – The user-specified account number from the @RUN command.	->TYPE ID('ACCT') 555555
LOWER('s')	A string with all uppercase alphabetic characters changed to lowercase. This function is ignored in ASSUME ASCII OFF mode.	-> *T LOWER ('AaBb')* aabb ->
OBJ( )	The current name of the assumed object file. The file is initially TPF$, but may be changed by an ASSUME OBJECT or ASSUME FILE command.	-> *TYPE OBJ( )* TPF$ ->
STATUS('s')	The value of an ASSUME, TAB, or SYNTAX command parameter is returned as a string. The values for the CTS control characters (see 1.5) can also be requested. See 9.2.5 for a list of values for s.	-> *TYPE STATUS('ASCII')* ON -> *TYPE STATUS('TAB')* 11,21,39,73 -> *TYPE STATUS('=')* *,%,'
TAB([n])	Blanks are produced to position the next string expression to column n if n is given. If n is omitted, the string expression will be positioned to the next column, as indicated by the last TAB command. Only rightward positioning is possible.	-> *T 'AB' TAB(12) 'CD'* AB CD -> *T TAB(12) 'AB'* AB -> *TAB 10,20* -> *T 'AB' TAB( ) 'CD'* AB CD -> *T TAB( ) 'AB'* AB
TRM(s)	A string equal to string s with all trailing blanks removed.	-> *SET S4='ABC '* -> *T LEN(S4)* 6 -> *T LEN(TRM(S4))* 3 -> *T LEN(S4)* 6 -> *SET S5=TRM(S4)* -> *T LEN(S5)* 3 ->

Table 12-2. String Functions (continued)

Function Reference	String Value Returned	Examples
TXT(n,c1,c2)	A substring from a line of f specified by n. If n is zero, then the current line of f is implied. The string is obtained from the line within column limits specified by c1 and c2. If the specified line n does not exist or if the column limits are illegal, then an error message is returned. Let k be the length of line n. If c2 > k, then k is used as the value for c2. If c1 > k, then the null string is returned.	*-> P 120* 120 JUMP 20 NO FIND *-> T TXT(120,1,4)* JUMP *-> 100 ABCDEF* *-> T TXT(100,4,3)* <125> BACKWARD COLUMN LIMIT IS INVALID *-> T LEN(TXT(100,4,3))* <125> BACKWARD COLUMN LIMIT IS INVALID *-> T TXT(100,5,7)* EF *-> T TXT(100,7,8)* *-> T LEN(TXT(100,7,8))* 2
TXT(s,c1,c2)	A substring from the string s as specified by the column limits c1 and c2. Let k be the length of string s. If c2 > k, then k is used as the value for c2. If c1 > k, then the null string is returned.	*-> SET A='ABCDEFGHI'* *-> T TXT(A,4,6)* DEF *-> T TXT(A,8,10)* HI *-> T TXT(A,10,12)* *-> T LEN(TXT(A,10,12))* 2 ->
UPPER('s')	A string with all lowercase alphabetic characters changed to uppercase. This function is ignored in ASSUME ASCII OFF mode.	*-> T UPPER ('AaBb')* AABB

12.1.5. CTS Variables

A variable's value may be set and referenced by CTS commands. The value of a variable may be changed at any time by a CTS command. Its value remains until it is changed. A variable name is composed of one to twelve alphanumeric characters, the first of which must be alphabetic. No distinction is made between upper and lower case alphabetic characters. A variable may be assigned an integer value, a real value, or a string value. A variable is defined or changed by a SET command (or a QUERY command in a subroutine).

12.1.6. Operators

Numeric terms of an expression (constants, variables, or functions) may be separated by one of the following arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
**	exponentiation

String or numeric terms may be separated by one of the following relational operators:

<			Less than
>			Greater than
=			Equal to
<>	or	><	Not equal to
<=	or	=<	Less than or equal to
>=	or	=>	Greater than or equal to

The terms on either side of the operator must be the same type (string or numeric). The result of the operation is either one (condition is true) or zero (condition is not true). The relational operations are done after all arithmetic operations have been completed but before functions are evaluated. The order of evaluation may be changed by using parentheses for grouping.

The result of an arithmetic operation is a real number if either of the terms is real. If both terms are integer, the result is also an integer. This means that the fractional part of the result (if any) is discarded. For example:

```
->T  5/2
2
->
```

12.2. Variable Definition - SET

Syntax: SET v=e

Abbreviation: v=e

Function: To evaluate the expression e and store the result into variable v.

The variable name v must begin with an alphabetic character and may contain from one to twelve alphanumeric characters.

NOTE:

Some variable names are reserved for internal use. These variables always begin with the letters SYS. Therefore, do not use a name starting with these characters.

Variables may be referenced by the TYPE command and they may be inserted into a CTS command using the percent sign (%) (see 8.3.6).

A SET command may be abbreviated S in a subroutine, but the command must appear. When in desk calculator mode and not in a subroutine, the command may be dropped, leaving only v=e to define the variable v.

12.3. Evaluating and Printing Expressions – TYPE

Syntax: TYPE e1 [e2 e3...]

Abbreviation: T

Function: The expressions e1, e2... are evaluated and the results are printed on one line.

The TYPE command is analogous to an output command like PRINT in BASIC or WRITE in FORTRAN. Since the expressions are evaluated first, complex expressions containing constants and variables may be specified in the TYPE command.

If the terms are not separated by an operator, concatenation is performed. To separate expression values with a blank in the print, place a blank character string between them on the command.

For example:

```
->R=5
->PI=4*ATN(1)
->C=2*PI*R
->T 'CIRCUMFERENCE = 'C
CIRCUMFERENCE = 31.415926535897932
->T R' 'FMT(c,6,-2)
5 3.1E1
->
```

If numeric results are not printed through use of the FMT function, then numbers (N) are converted and printed as follows:

- If N is of type integer, then N is printed as an integer.
- If N is of type real and $10^{16} > |N| > 1$, then the integral part of N is printed, followed by a decimal point, possibly followed by a fractional part.
- If N is of type real and $|N| > 10^{16}$ or $|N| < 1$, then N is printed in floating point, i.e., up to 16 decimal digits, with the decimal point following the first digit, followed by the signed power of ten exponent in the form:

  E±ddd

  where ddd is the unsigned power of ten.

  For real numbers, trailing zeros are automatically truncated, and in no case will more than 16 significant digits be printed. In all cases above, the number will be preceded by a minus sign if the number is negative.

12.4. Iterative Expression Evaluation – DISPLAY

Syntax: DISPLAY $V_1(S_1;E_1;[I_1;])\ V_2(S_2;E_2;[I_2;])\ V_3(S_3;E_3;[I_3;])$ EN=EXP [;]

Abbreviation: DIS

Function: To print in table form V_1, V_2, V_3 and EN, where EN is the value of the expression EXP for each possible combination of values for V_1, V_2, and V_3.

When DISPLAY prints this table, each column is headed by the appropriate variable name. On normal termination the values of V_1, V_2, V_3 and EN are the same as the last line of the table. With the exception of the blank between DISPLAY and V_1, blanks are optional.

V_1, V_2, V_3	CTS variable names.
S_1, S_2, S_3	Start values.
E_1, E_2, E_3	End values.
I_1, I_2, I_3	Increment values (if not specified, one is assumed).
EN	CTS variable name which will be defined by CTS to have an initial value of zero.
EXP	A CTS expression which has a numerical result.

Since the start, end, and increment values may be any CTS expression which has a numerical result, these values must be separated by semicolons.

The initial value of each variable is its respective start value. Each time the variable is incremented, its value changes to its current value plus the increment. The value of a variable is always less than or equal to its end value.

The variable V_3 will be incremented over its range, before the variable V_2 is incremented. The variable V_2 will in turn be incremented over its range before the variable V_1 is incremented. For each new value of V_2, the parameters S_3, E_3, I_3 and V_3 are reevaluated and V_3 is incremented over its range again. For each new value of V_1, the parameters S_2, E_2, I_2, V_2, S_3, E_3, I_3 and V_3 are reevaluated and the process starts over. When V_1 has been incremented over its range the command terminates.

This is an example of a table printed by a DISPLAY command:

```
->DISPLAY  A(1;4;)  B(3;5;)  HYP=SQR(A**2+B**2)
                A              B              HYP
     1.0000000E0    3.0000000E0    3.1622776E0
     1.0000000E0    4.0000000E0    4.1231056E0
     1.0000000E0    5.0000000E0    5.0990195E0
     2.0000000E0    3.0000000E0    3.6055512E0
     2.0000000E0    4.0000000E0    4.4721359E0
     2.0000000E0    5.0000000E0    5.3851648E0
     3.0000000E0    3.0000000E0    4.2426406E0
     3.0000000E0    4.0000000E0    5.0000000E0
     3.0000000E0    5.0000000E0    5.8009518E0
     4.0000000E0    3.0000000E0    5.0000000E0
     4.0000000E0    4.0000000E0    5.6568542E0
     4.0000000E0    5.0000000E0    6.4031242E0
->
```

The DISPLAY or SUM command may cause the following diagnostic messages:

```
<193>  *ERROR - THE INCREMENT FOR [variable name] IS ZERO
<195>  *ERROR - THE INCREMENT FOR [variable name] IS NEGATIVE BUT THE
START VALUE IS LESS THAN THE END VALUE
<194>  *ERROR - THE INCREMENT FOR [variable name] IS POSITIVE BUT THE
START VALUE IS GREATER THAN THE END VALUE
<192>  *ERROR - INVALID EXPRESSION OF UNDEFINED VARIABLE FOR [variable
name or expression name]
```

12.5. Iterative Expression Summation – SUM

Syntax: SUM $V_1(S_1;E_1;[I_1;])$ $V_2(S_2;E_2;[I_2;])$ $V_3(S_3;E_3;[I_3;])$ EN=EXP [;]

Abbreviation: None

Function: To set the expression name, EN, to the sum of the values of the expression, EXP, for each possible combination of values for V_1, V_2 and V_3.

The parameters are the same as for DISPLAY. Since the start, end, and increment values may be any CTS expression which has a numerical result, these values must be separated by a semicolon.

For example:

```
->SUM A(1;4;) B(3;5;)  HYP=SQR(A**2+B**2)
HYP = 57.980825905866712
->
```

On normal termination the values of V_1, V_2, and V_3 are the same as they would be for the DISPLAY command. See the DISPLAY command (12.4) for a list of possible error messages.

12.6. Removing a Variable or Subroutine – DROP

Syntax: DROP V_1 [, V_2, V_3, ..., V_n]

Abbreviation: DRO

Function: To remove or drop variables from CTS operating environment.

This command will drop or deactivate variables with the names V_1, ..., V_n. Once dropped, these variables can no longer be accessed unless they are reestablished by a SET or QUERY command.

If V_n is an established variable, it is dropped immediately. If V_n is not a variable, an error results.

Any erroneous variable name V_n terminates DROP command processing. Any variables dropped prior to the error will remain deleted.

Example:

```
-> SET A=1
-> TYPE A
1
-> DROP A
-> SET B=A+2
<8> VARIABLE A IS UNDEFINED
```

13. Batch Mode

13.1. General

Much of this manual has discussed the entering, creating, editing, and output of programs from a terminal. This implies that all work has been performed in demand and conversational modes. It is important to establish that batch mode of operation is not in contrast to, but is a part of, the SPERRY UNIVAC Series 1100 Conversational Time Sharing System. Batch processing is the technique of executing a set of computer programs such that each is completed before the next program of the set is started. The term "batch processing" therefore implies the serial execution of programs. Because of this concept of running programs serially, programs performed at the central site are considered to be batch; and are usually read to the computer in the form of punched cards or magnetic tape; and the output produced by these same runs is contained on magnetic tape, paper tape, punched cards, microfilm, or printed output.

Clarifying what is meant by executing programs serially will help explain how terminal input can be treated as if it were an entire program read in through a card reader.

The Executive receives input that is acted upon when the entire program has been read in. The input has a @FIN control statement at the end. All of the instructions and control statements (which may number in the thousands) are stored until the last image has been read in. A job (or run) does not "start" until the @FIN statement has been entered.

The batch run is designed primarily so that input devices such as card readers, magnetic tape drives, paper tape readers, etc., reach a high speed of transfer (input).

The batch job may be few or many statements, instructions, or control commands. These programs usually are not entered a line at a time at the terminal, but are brought into the run stream as an entire program or group of instructions. By building program files and having the @RUN control statement as the first image of the program file, the entire program is read in from the terminal in the form of paper tape, cassette input, keyboard entry, magnetic tape, or disk or drum storage under terminal file control. In either Executive mode or CTS mode, the option of batch input with the @START command must be used. As stated earlier, the first statement in the file must be a @RUN with the necessary parameters (run-id, account number, project-id, etc.,) or these parameters must be supplied with the @START command, in which case the account number that is used in the @START*name, set, run-id, account number, project-id, run-time* statement is used. The end of the file or element denotes a @FIN control statement.

To avoid confusion, the statement format as it is used in Executive mode is illustrated first, followed by the *CSF command in CTS mode. The same @RUN card is used as in Sections 9 and 10, but notice the addition of the "B" option as @RUN,/B JIM,123456, SMITH,15,50. The following subsection illustrates the @START command in both Executive mode and CTS mode.

13.2. Starting a Batch Job from the Terminal

13.2.1. In Executive Mode – @START

The example starts as if logging-on the system with TSS (Terminal Security System). The B option is used on the original log-on statement as follows:

```
DCT236
ENTER USERID/PASSWORD
*SMITH/HAPPY
*DESTROY USERID/PASSWORD ENTRY*
*UNIVAC 1100 OPERATING SYSTEM VER.xx.xx.xx*
>@RUN,/B JIM,123456,SMITH,15,50,100 (Batch mode--enter stream.)
>@ASG,A FIL3.
>@START FIL3. BLOOD
>@FIN
```

Immediately after the @RUN, the cataloged file FIL3. is assigned to the run. The first command in BLOOD must be a @RUN JIM,123456,SMITH,25, statement. The output from BLOOD should be returned to JIM SMITH.

The following example uses the @START with a different format because the next element does not have a @RUN statement as the first command. The same file, FIL3 is used, but a program contained in element SWEAT is executed:

```
>@HDG HOLD SWEAT OUTPUT FOR TOM BROWN IN OUTPUT BIN
>@START FIL3.SWEAT,,TOM,123456,BROWN,10,15/25
```

In the event that the element SWEAT contains a @RUN statement, the parameters supplied by @START FIL3.SWEAT,,TOM,123456,BROWN would be overwritten, or have priority over the @RUN control card in SWEAT. A message is not needed because the heading explains that the printout is to be held for Tom Brown.

Notice the format of the @START statement where two commas were used together (,,) after FIL3.SWEAT. The second field after the file-element name (FIL3.SWEAT) is for SET which specifies an octal number to be placed for altering the normal execution sequence of the run. SET was eliminated by placing two consecutive commas after the first field.

13.2.2. In CTS Mode – CSF 'START'

Syntax: CSF 's'
s= START

Abbreviation: None

Function: To initiate dynamically an independent run and permit the user to schedule independent batch runs where the run streams for these runs have been created and entered previously into the system.

Runs scheduled by this control statement must be in SDF (see 7.1) and must be cataloged as either data files or data elements. The following illustrates the same format as in Executive control mode:

```
>@CTS,I
CTS 8R1 31 OCT 80 AT 07:37:02
```

```
IF YOU NEED ASSISTANCE TYPE  *HELP
FOR NEW FEATURES TYPE  *CALL CTS-COMMANDS
THE ASSUMED MODE IS ASCII
->*CSF  'START FIL3.TEARS'
```

A @RUN statement was not necessary in the example just shown, assuming that the @RUN source statement was the first control command in element TEARS of file FIL3. In the following example, we supply the @RUN for the program in element, TEARS:

```
>@CTS,I
CTS 8R1 31 OCT 80 AT 07:37:02
IF YOU NEED ASSISTANCE TYPE  *HELP
FOR NEW FEATURES TYPE  *CALL CTS-COMMANDS
THE ASSUMED MODE IS ASCII
->*CSF  'START FIL3.TEARS,,HARRY,123456,HORN'
->
```

The @RUN statement supplied with the CSF @START command would have precedence over a @RUN statement if there were one present as the first source statement in element, TEARS. The double commas voided the second field of the @RUN statement so that the execution sequence would not be altered.

If no messages are printed, the operation is successful. The output from the program of element TEARS in file name FIL3, goes to the primary printer at the central site:

```
->@FIN
RUNID: JIM  ACCT: 123456  PROJECT: SMITH
SMITH*MSG SMITH STARTING A BATCH RUN FROM SITE ID DCT236
TIME:    TOTAL:  00:02:33.722
         CAU:    00:00:00.403     I/O:  00:00:25.205
         CC/ER:  00:02:08.113     WAIT: 00:15:27.992
SUAS USED: $     3.00   SUAS REMAINING:  $200000.00
SRC:     PS=        468533       ES=      0453091
IMAGES   READ:   4          PAGES:  2
START:   07:40:02  Oct 31,1980  FIN: 07:46:32  Oct 31, 1980
*TERMINAL INACTIVE*
>
```

The terminal does not return to a terminal inactive mode until a @FIN command has been entered. The @FIN is implied for any runs that were dynamically initiated by the START command by either the end of file or end of element. Another @RUN statement could be entered while in batch mode, but it is also treated as a batch run.

NOTE:

It is recommended than a @FIN command be used to terminate all batch runs that call on CTS. If not used, unpredictable results may occur.

The many applications of the @START command are limited only by limitations of the program that has been cataloged.

Debug work is also greatly enhanced by being able to perform tests and print the entire output at the local site. It is also very easy to perform the same tests repeatedly for this same purpose.

Prestored utility routines and standard production runs are of particular benefit when used with the START command.

13.2.3. In Either Mode – @@START

A special mode of processing directs the operating system to process control statements immediately after they are received from a remote terminal. The processing called for by the control statement is done independently of any current program execution or control statement processing in the run stream. This mode of executing a control statement is specified by a special character, a second @ in column 2 in the control statement. This mode of operation is called transparent mode, and control statements which can direct or specify this mode of operation are called transparent control statements.

Transparent control statements are a subset of the control statement set. The syntax rules for normal control statements, with the following exceptions, also apply to transparent control statements. The exceptions are as follows:

1. The identification of a transparent control statement consists of a double @@ versus a single @ for a normal control statement.

2. The use of a label on a transparent control statement, while not prohibited, is meaningless.

3. Transparent control statements may be entered only in demand run mode.

4. Transparent control statements may not be entered from any means other than the primary input device, (i.e., at the keyboard only, not on paper tape, full screen, or cards).

5. Processing of any previous transparent control statement from the same terminal must have been completed.

6. Any run initiated from a demand terminal using the @START or @@START control statement is scheduled as a batch run with its output going to the onsite peripherals.

See Appendix A for a list of transparent control statements.

13.3. Batch/Time Sharing Compatibility

CTS maintains a compatibility with the Executive wherever possible. The files created in CTS are created through the Executive. Consequently, these files are registered with the Executive exactly as if they had been created outside of CTS in either demand or batch mode. Whether they are used as data files or program files, a strict adherence to the respective formats is maintained.

Many operations are performed by the Executive itself, even when interfacing through CTS. Compilations, collections, executions, and many file manipulations are examples. In a sense, CTS really acts as a convenient interface between the programmer and the Executive.

This compatibility has some far-reaching implications. It is entirely possible, and sometimes advantageous, to use the Executive for part of an effort and CTS for the rest. Source code may be created via the normal batch mode operation, using either a compiler (getting the benefit of a debugging run at the same time) or the ELT processor. This can take advantage of the skill of a keypunch operator. For creating large programs, this may be the most efficient method. Once the symbolic element is in a program file, CTS can edit, modify, test, and update it. Depending on the type of program, it can be executed in the future in batch mode. The fact that a file or element was created in one mode does not prevent it from being used in another mode.

Appendix A. Transparent Control Statements

Command	Mode	Terminal	Meaning
@@CDI	Demand Remote Batch Inactive	DCT 1000	Turn on card input.
@@CDO	Demand Remote Batch	DCT 1000	Turn on card output.
@@CONT	Demand Remote Batch Inactive	ALL	Continue. Used after a break-key (interrupt of output) to signify that no special action is desired.
@@CQUE	Demand	ALL	Circumvent input solicitation requirement. This allows several input images to be buffered in main storage before the terminal is placed in a wait condition.
@@DCT	Demand Remote Batch Inactive	TTY DCT 500	Changes terminal operating characteristics. Parameters are required.
@@END	Demand Remote Batch	ALL	Terminates any special input mode; i.e., @@CQUE, @@INQ, @@FUL.
@@ESC	Demand	ALL	Allows input to be passed to the requestor unaltered from the format in which it was entered.
@@FUL	Demand Remote Batch Inactive	UNISCOPE 100/200 UNISCOPE 300	Puts terminal in full-screen input mode.

Command	Mode	Terminal	Meaning
@@FRZ n	Demand Remote Batch Inactive	UNISCOPE 300	Allows the uppermost n lines to be unaffected by UNISCOPE scrolling.
@@HI	Demand Remote Batch Inactive	UNISCOPE 300	Sets high-speed (normal) output rate.
@@INQ	Demand Remote Batch Inactive	ALL	Directs the Executive to buffer all input to mass storage until an @@END is received. If the @@INQ is entered when the terminal is inactive, the next input should be a @RUN image. The following input will be treated as a remote batch input.
@@INS n	Demand Remote Batch Inactive	UNISCOPE 100/200 UNISCOPE 300	Set insert point at line n.
@@LOW	Demand Remote Batch Inactive	UNISCOPE 300	Sets low-speed output rate. This is always assumed when a PAGEWRITER printer is active.
@@MED	Demand Remote Batch Inactive	UNISCOPE 300	Sets medium-speed output rate.
@@NOPR	Demand Remote Batch Inactive	DCT 1000 UNISCOPE 300 DCT 500 S/A UNISCOPE 100/200	Release assigned PAGEWRITER printer or Communications Output Printer (COP).
@@PRNT n	Demand Remote Batch Inactive	DCT 1000 UNISCOPE 100/200 UNISCOPE 300 DCT 500	Select the printer as the output device. For UNISCOPE 100 and UNISCOPE 300, turns COP or PAGEWRITER printer on or off. The COP number n must be specified for UNISCOPE 100/200 and UNISCOPE 300.
@@PTI	Demand Remote Batch Inactive	TTY DCT 500 DCT 1000	Select paper tape as input. If @@PTI is entered when the terminal is inactive, the first image on paper tape should be a @RUN image. The input will then be treated as a batch run.
@@PTO	Demand Remote Batch Inactive	DCT 500 DCT 1000	Select paper tape punch as output device.

Command	Mode	Terminal	Meaning
@@PTP	Demand Remote Batch Inactive	DCT 1000	Enter point-to-point mode. Optional with configuration.
@@RLD	Demand Remote Batch Inactive Inactive	UNISCOPE 100/200 UNISCOPE 300	Roll screen down.
@@RLU	Demand Remote Batch Inactive	UNISCOPE 100/200 UNISCOPE 300	Roll screen up.
@@RQUE	Remote Batch	ALL	Stop printing batch output file but save file for later.
@@SEND	Inactive	ALL	Print batch output files.
@@SKIP n	Demand	ALL	Skip n lines of output (n < 64).
@@TCT n,TRACK	Demand Inactive	UNISCOPE 100/200	Position tape cassette UNIT n to top of TRACK (1 or 2).
@@TCM	Demand	UNISCOPE 100/200	Tape mark to deselect tape cassette unit.
@@TCI n,NUMBER	Demand Inactive	UNISCOPE 100/200	Select tape cassette UNIT n to input number of blocks or until tape mark if number is not specified.
@@TCO n,MARGIN	Demand Inactive	UNISCOPE 100/200	Select tape cassette UNIT n for output. Format one image/block unless lines specifies the maximum number of lines to output/block.
@@TERM	Demand Remote Batch Inactive	ALL	Terminate site.
@@TTY	Demand Remote Batch Inactive	TTY DCT 500	See @@DCT.
@@X [param] param = I = O = T = C	 Demand Demand Demand Demand	 ALL ALL ALL ALL	 Release backed-up input. Release backed-up output. Terminate executing task. Control to contingency.

Appendix B. Explanation of CTS Messages

This appendix lists the messages and diagnostics generated by CTS. Most of the diagnostics are preceded by a number within <> characters. This number is the message number which may be specified on an EXPLAIN command. The explanation is the one which would be given by EXPLAIN. The miscellaneous messages do not have message numbers because they are printed by a CTS internal subroutine or because they are not diagnostics.

B.1. Miscellaneous Messages

ALL MAIL FILES IN USE
No additional messages may currently be sent to the run-id.

ASSUMED COMPILER?
Self explanatory.

Action: Specify an assumed compiler in response to above query and the COMPILE or RUN will continue.

AUTO
This simply indicates that CTS has just done an auto save and that the ASSUME AUTO with the print has been turned on.

Action: None required.

BAD SCAN FILE
CTS could not read the ASSUME SCAN file either because it did not exist or it is improperly formatted.

Action: To see the exact reason, execute ASSUME SBUG ON, followed by a SCAN command.

*BREAK
CTS has terminated the current command on a break contingency. (@@X C will cause this message to be printed if CTS is not finished with the command being interrupted.)

CANNOT RESTORE ENVIRONMENT – CTS REINITIALIZED
Recovery file exists, but cannot be used to restore environment because of file errors, or because an abnormal exit was taken from CTS and an auto save had not been done.

CANNOT SAVE WORKAREA
Self explanatory.

Action: An attempt to list the working area will usually show the exact reason.

CTS RESTART
The recovery file exists and was used to reestablish the environment following an abnormal run termination.

DECK NAME LOST
Self explanatory.

END OF FILE
An editing command has encountered the end (a line number greater than any existing line) of f. The end flag is set.

END OF PRINT FILE
The end of the scan file has been detected. The next scan will start at the beginning of the scan file.

ERROR DO YOU WANT A DUMP?
Of the large variety of potential error interrupts, CTS checks for the errors it can take action on. The rest will cause this message.

Action: Respond *YES* to this, and CTS will query for identification of the dump and send a diagnostic dump to the onsite printer. The dump and the interrupt can be used by the site analyst to correct the error.

FILE IS BUSY—WAITING
Another CTS user is writing into the file, or waiting for his write operation to finish.

Action: The request has been queued. Wait for it to be performed or abort the requested action, with a @@X C.

IN EXEC MODE
CTS has saved its environment and is exiting to execute another program. This means control is leaving CTS.

WARNING—THE WORK AREA WAS NOT SAVED/REPLACED
If the work area has been edited but not SAVED or REPLACED before existing CTS, the user is warned.

INCOMPLETE PRINTFILE
This is a warning only. The run producing the print file was abnormally terminated. Information that is in the scan file SQUELCH$ is available.

NO SCANNER FOR THIS PROCESSOR
The scan file includes a listing from a processor for which there has not been developed a diagnostic scan routine.

**OPERATOR INTERRUPTED PROGRAM
The operator executed an II keyin at the console. It interrupts CTS in a manner similar to the break contingency.

PROCESSOR, (NAME), CAN NOT BE FOUND
The processor specified on a SCAN command was not in the scan file.

PROGRAM, (NAME), CAN NOT BE FOUND
The program specified on a SCAN command was not in the scan file.

STATEMENT NOT IN CLIST
An Executive control statement or a line preceded by an @ has been entered.

Action: If this message occurs and CTS commands are not acceptable, then type @ENDX.

THE WORK AREA IS EMPTY
Self explanatory.

TOP OF FILE
An editing command has encountered the top (a line number smaller than any existing line) of f. The end flag is set.

B.2. CTS Diagnostic Messages

The explanations for the following messages are those that are printed by the EXPLAIN command.

<1> VARIABLE NAME LENGTH EXCEEDED (CHARACTER STRING)
A variable name greater than 12 characters has been defined.

<2> COMMAND LENGTH EXCEEDED (CHARACTER STRING)
A command has been entered which became larger than a CTS internal buffer when the character substitution was done for a variable reference or when defining a subroutine. Break the command into smaller commands.

<3> STRING LENGTH EXCEEDED (CHARACTER STRING)
The characters specified in the diagnostic exceed the length of an internal buffer. This occurs when an editing command encounters an invalid line. If an OLD or MERGE is terminated, the invalid line, which is the last one copied, may be listed by *PRINT !-1.

<4> ELEMENT (FILE NAME.ELEMENT NAME) CANNOT BE FOUND
The element referenced does not exist. Check spelling.

<5> DUPLICATE NAME (FILE NAME.ELEMENT NAME) - PROGRAM NOT SAVED
An attempt has been made to save a program but a program already exists by that name. Either specify a different program name with the SAVE command or use the REPLACE command.

<6> PROGRAM NOT SAVED
A SAVE or REPLACE was interrupted by a *@@X C.

<7> EXPRESSION IS TOO COMPLICATED
There are too many terms in the expression and an internal buffer has been exceeded.

<8> VARIABLE (VARIABLE) IS UNDEFINED
Self-explanatory - Check spelling.

<9> ILLEGAL OPERATOR (CHARACTER STRING)
A character has been entered which is not a valid operator or operand in an expression. Check typing or expression syntax.

<10> Not Used.

<11> UNBALANCED PARENTHESIS
Self explanatory – Check typing.

<12> UNBALANCED DELIMITER
The character string has no closing delimiter, usually a quote or the first character in the string.

<13> I/O ERROR WHILE READING ADD FILE
The Executive was unable to read the add file. Check creation of the file.

<14> ARGUMENT COUNT TO (FUNCTION NAME) IS BAD
The wrong number of arguments in the function was specified.

<15> UNKNOWN FUNCTION (FUNCTION NAME)
Self explanatory – Check spelling or documentation for valid CTS functions.

<16> ARGUMENT MODE TO (FUNCTION NAME) IS BAD
A numeric or string argument has been specified when the opposite mode is required, or when the argument exceeds the numeric range of the function.

<17> KEYWORD (CHARACTER STRING)
A keyword expected for this command was not given (missing) or incorrectly given. Check spelling and/or command syntax.

<18> (FILE NAME) IS NOT A DATA FILE
Self explanatory – Check command syntax, DATA mode, or file generation.

<19> (FILE NAME) IS NOT A PROGRAM FILE
Self explanatory – Check file generation.

<20> ILLEGAL COMMAND SYNTAX (CHARACTER STRING)
Self explanatory – Check spelling or type HELP.

<21> LINE (LINE NUMBER) DOES NOT EXIST
Self explanatory.

<22> REQUIRED SYNTAX IS MISSING
A required field was left off a command. See command syntax.

<23> ILLEGAL FILE OR PROGRAM NAME SYNTAX (CHARACTER STRING)
A file or program exceeds 12 characters, contains illegal characters, or has an invalid format.

<24> STRING EXCEEDS COLUMN LIMITS
The number of characters specified in the INSERT command exceeds the column limits. The action was not performed. The string may be expanded on insert only with NO KEY or KEY = PACK.

<25> NUMERIC CONVERSION ERROR
A number specified cannot be represented as floating point.

<26> CANNOT COMPARE STRING TO NUMBER
The two expressions for a relational operator were not of the same type. If part of a JUMP instruction, the jump was not taken.

<27> COMMAND IS LEGAL ONLY IN EDIT MODE
This command requires the working area images which were saved by the SCAN command. They are restored by EDIT.

<28> STATEMENT NUMBER (NUMBER) IS MULTIPLY DEFINED
Two or more statements within a subroutine have the same statement number. The subroutine definition was not performed.

<29> STATEMENT NUMBER (NUMBER) IS NOT DEFINED
Statement number has been specified in a JUMP or BRANCH command, but it is not defined in the subroutine. The subroutine definition was not performed.

<30> ASSUME JUMP MAX. EXCEEDED – SUBROUTINE TERMINATED
The maximum number of allowable JUMPs specified by ASSUME JUMP has been exceeded.

<31> COMMAND IS LEGAL ONLY IN SUBROUTINE MODE
The JUMP, BRANCH, RETURN, ENTRY, and END commands are legal only within a subroutine.

<32> ILLEGAL CALL NESTING TO (SUBROUTINE NAME)
A subroutine cannot call itself either directly or indirectly. The call was not performed and the calling subroutine is terminated.

<33> FILE (FILE NAME) IS FULL
All of the available space in the file is used. If it is a program file, pack it. If not, it must be released and created with a larger max size specification.

<34> (FILE NAME) IS TOO SMALL TO BE A PROGRAM FILE
At least 29 tracks must be specified for a program file. The default of 128 tracks is suggested. PURGE and recreate the file.

<35> DIVIDE OVERFLOW
The result of a divide was too large.

<36> FLOATING POINT OVERFLOW
The result of a multiply exceeded arithmetic limits of 10**308.

<37> ILLEGAL READ/WRITE KEYS SYNTAX (CHARACTER STRING)
File read/write keys may each be up to six characters in length and consist of any character except a period, comma, slash, or blank. Check typing.

<38> AFTER LINE (LINE NUMBER) THERE IS A BAD CONTROL WORD (CHARACTER STRING)
The file is damaged or is not in SDF format. The specified line number was the last line copied.

<39> TAB TABLE IS FULL
Either more than 4 tab characters or more than 12 tab stops were specified, or a combination of the 2 specifications caused tab limits to be exceeded.

<40> A PERIOD IS NOT ALLOWED IN A HDG
Self explanatory.

<41> FILE (FILE NAME) IS NOT FASTRAND
The file referenced is probably tape or a word addressable file.

<42> READ OR WRITE PERMISSION DENIED ON (FILE NAME)
The file is read inhibited or is write inhibited and the command requires the missing permission. The file may have been created as read or write only, or the requested keys were not specified.

<43> COMMAND IS LEGAL ONLY IN SCAN MODE
CTS was already in edit mode.

<44> END-OF-FILE ENCOUNTERED ON (FILE NAME)
Self explanatory.

<45> TAPE (FILE NAME) IS AT LOAD POINT
Self explanatory.

<46> MISSING SENTINEL ON (FILE NAME)
A mass storage search operation has failed.

<47> NON-INTEGRAL BLOCK WAS READ ON (FILE NAME)
A partially full tape block was read. This may be valid but quite often indicates the tape has been damaged or was created on a noncompatible tape drive.

<48> BAD ADDRESS LINK ON (FILE NAME)
A portion of the specified file has been lost or moved by a pack. Use PRINT to find the line pointer position where the bad link occurs. The lines preceding and/or following any line with a bad link may be saved by deleting the erroneous line.

<49> CSF SYNTAX ERROR IN IMAGE...
The specified image is not a valid Executive control card. This normally denotes an Executive error return after a CSF command. Check CSF syntax.

<50> LOSS OF POSITION ON TAPE (FILE NAME)
The console operator has entered a B response to an I/O error message on tape or mass storage. The operation is terminated.

<51> FILE (FILE NAME) REQUIRED BUT HAS BEEN FREED
CTS$FILE or the file read by the last OLD command has been freed. The file name specified is an internal name. If the file is not CTS$FILE, either reassign the file, if possible, or delete the working area. If the file is CTS$FILE, exit with an EXIT command and CTS will reinitialize.

<52> F-CYCLE CONFLICT ON (FILE NAME)
CSF status bit 5 –
(A) cataloging of the requested f-cycle would force deletion of a currently assigned f-cycle.
(B) F-cycle generation inhibited due to existence of +1 file.
(C) F-cycle requested is not within currently acceptable range.

<53> LINE (LINE NUMBER) IS TOO LONG
(A) A line's length exceeds the ASSUME SAVE LENGTH limit (normally 162 characters). The image is entered correctly but will be truncated during a save.
(B) A very large image has been encountered during an OLD or MERGE, probably due to file damage or the file not being in SDF format. The operation is terminated.

<54> ERROR – JUMP PAST END OF SUBROUTINE
A relative jump is beyond the limits of the subroutine. This is an abnormal subroutine termination and control is returned to command mode.

<55> (FILE NAME) IS DISABLED
CSF status bit 8 or 6 – The file is not accessible because the links to the master file directory items have been destroyed or because the file has been rolled out and the backup copy is unrecoverable.

<56> WARNING – DEVICE FOR (FILE NAME) IS DOWN
CSF status bit 9 – The site operator has downed the equipment.

<57> WARNING – FILE (FILE NAME) IS READ INHIBITED
CSF status bit 10 – The file is a write-only file cataloged with a W option.

<58> WARNING – FILE (FILE NAME) IS WRITE INHIBITED
CSF status bit 11 – The file is a read-only file cataloged with an R option.

<59> WARNING – (FILE NAME) IS TAPE
CSF status bit 12 – Self explanatory. This is only a warning.

<60> (FILE NAME) IS PRIVATELY CATALOGUED
CSF status bit 13 – File has been cataloged by a different project-id as a private file. Your request is denied.

<61> (FILE NAME) HAS BEEN DELETED
CSF status bit 14 – File has been marked for deletion in the master file directory and will be decataloged when no run has it assigned. The request is denied.

<62> EXCLUSIVE USE OF (FILE NAME) DENIED
CSF status bit 15 – The file is assigned to another run. The reference requires exclusive use. The command is not executed.

<63> SOMEONE ELSE HAS EXCLUSIVE USE OF (FILE NAME)
CSF status bit 16 – Self explanatory. The request is denied.

<64> ILLEGAL ATTEMPT TO ALTER (FILE NAME)
CSF status bit 17 – Option conflict for a cataloged file. The file was already cataloged when the C or U option was used, either alone or in combination with P, R, or W. The request is denied.

<65> (FILE NAME) IS TEMPORARILY UNAVAILABLE
CSF status bit 18 – The file is not accessible because someone has assigned it for exclusive use, the file has been rolled out, or the physical unit is not available. The action is not queued. Wait and try again.

<66> (FILE NAME) HAS BEEN ROLLED OUT
CSF status bit 19 – The file has been rolled out on tape. This reference causes a request, to roll the file in to mass storage, to be queued. Wait and try again.

<67> WRONG REEL NUMBER FOR (FILE NAME)
CSF status bit 20 – The specified file is cataloged on a different tape reel or removable disk than specified on the create statement.

<68> (FILE NAME) IS NOT CATALOGUED
CSF status bit 32 – The file does not exist. Check spelling.

<69> WARNING – FILE (FILE NAME) HAS NOT BEEN CREATED
The file referenced in the second field of the USE command does not exist. Check spelling or typing.

<70> ASCII IMAGES ENCOUNTERED AND IGNORED
The file being read by OLD or MERGE contains ASCII images which were not copied. The Fieldata images were copied.

<71> WARNING – READ KEY MISSING ON (FILE NAME)
CSF status bit 24 – This is a warning that the file was created with a read key and none has been specified on the command.

<72> WARNING – WRITE KEY MISSING ON (FILE NAME)
CSF status bit 25 – This is a warning that the file was created with a write key and none has been specified on the command.

<73> PROGRAM NOT REPLACED
The data file name specified on a REPLACE command does not exist. Check DATA mode, or spelling, or use the SAVE command.

<74> PROGRAM HAS NO END CONTROL IMAGE
The data file or program element is damaged or incomplete. All of the lines may have been read successfully. Use PRINT !–1 to see the last line read.

<75> (FILE NAME.ELEMENT NAME) IS AN ASCII ELEMENT – NOT COPIED
Self explanatory – The program can be read into the CTS working area by specifying NUMBER mode with a NUM command and keying in @ADD (FILE NAME.ELEMENT NAME).

<76> WARNING – FILE NAME (FILE NAME) IS NOT UNIQUE
CSF status bit 29 – This is only a warning that two files exist with the same name but different qualifiers or f–cycles.

<77> ILLEGAL LINE LIMIT SYNTAX (CHARACTER STRING)
Self explanatory for most commands. The valid line limit syntax on OLD or MERGE commands is more restrictive than for other commands and this diagnostic is given for those not allowed. This message is also given if a command would generate a line number of zero.

<78> WRONG DEVICE TYPE FOR (FILE NAME)
CSF status bit 31 – User has specified sector–addressable device or a tape unit device when referencing a cataloged file on the other type device.

<79> FILE (FILE NAME) IS ALREADY CATALOGUED
CSF status bit 32 – A file already exists by the name specified.

<80> I/O ERROR (OCTAL NUMBER) ON (FILE NAME)
An I/O operation has terminated with the specified error condition. Check the description in UP–7924 or see the Sperry Univac site analyst.

<81> FORMAT OR OPTION ERROR IN CONTROL STATEMENT
CSF status bit 34 – There is an error, other than syntax, in the control statement submitted to the Executive. The error is an option conflict (MHL,OE, or IB), or noise constant specification error, or the requested hardware is not currently part of the system.

<82> MISSING TERMINATION CHARACTER: (CHARACTER)
An INLINE editing command must be terminated by the specified character.

<83> COMMAND WOULD CAUSE ASSUME DUP RES OF A BASIC PROGRAM
The specified DITTO or GENERATE command would cause a line number conflict when ASSUME DUPLICATE RESEQUENCE mode is set. BASIC program images can only be

resequenced with a MOVE or RESEQUENCE command and the BASIC prescanner must be active. Any other resequence is invalid because the program statement number references would not get resequenced. Either specify ASSUME DUPLICATE DELETE and lose the conflicting lines or use RES prior to the DITTO or GENERATE.

<84> BAD EDIT CHARACTER
The only valid edit characters on an INLINE command are I, D, and R.

<85> NO ERRORS SINCE LAST EXPLAIN
No diagnostics which have explanations have been given by CTS or BFOR (FORTRAN prescanner) in this terminal session or since the last error explanation was requested. RFOR errors can be explained but do not set the default last error condition.

<86> NO EDIT CHARACTER
An INLINE editing line with no nonblank characters has been entered.

<87> INVALID COMMAND
CTS does not recognize the initial characters of a line as a valid command or valid data line number.

<88> ERROR MESSAGE (DECIMAL NUMBER) IS NOT DEFINED
Either there is no error message with the specified number or its explanation has not been entered in the system.

<89> COMPILE INVALID FOR BASIC – USE *RUN
BASIC is an processor which does not produce object code and therefore does not require COMPILE, MAP, and XQT to execute a program.

<90> DEPRESS PUNCH ON
Turn the paper tape punch on at the terminal.

<91> THE ADD FILE DOES NOT EXIST
SYMB 02 – Self explanatory. Check spelling or file generation.

<92> DEBUG ONLY FOR BASIC AND RFOR
The assumed compiler must be BASIC or RFOR to specify ASSUME DEBUG ON. They are the only processors with the run-time debug features.

<93> CARD LIMIT EXCEEDED
SYMB 42 – Self explanatory.

<94> Not Used.

<95> OUTPUT FILE IS NOT AVAILABLE
The file specified in the second field of COPY or TRANSFER command is not available. The reason was given in the preceding messages. The command was not performed.

<96> DUPLICATE DATA FILE NAME – NOT SAVED
An attempt was made to save into a data file which already exists and already contains data. Use REPLACE.

<97> BASIC PROGRAM MOVE REQUIRES BASIC PRESCAN MODE
The BASIC prescanner must be active to cause the BASIC program statement number references to be changed along with the statement numbers. Execute a *BASIC command before the MOVE.

<98> BASIC PROGRAM RESEQUENCE REQUIRES BASIC PRESCAN MODE
The BASIC prescanner must be active to cause the BASIC program statement number references to be changed along with the statement numbers. Execute a *BASIC command before the RESEQUENCE.

<99> INVALID FIELDATA STOP-CODE DETECTED IN IMAGE
The underscore is an illegal character in Fieldata mode.

<100> UNKNOWN DEVICE TYPE
The device specified on ASSUME SITE or in response to the SITE? query on a SITE or CARDS command is not a valid symbiont device type. Try PR for the onsite printer and CP for the onsite card punch.

<101> ILLEGAL NUMERIC SYNTAX (CHARACTER STRING)
The column limits fields on a TXT function, the number field on a NUM function, or the message number field on an EXPLAIN command contains nonnumeric characters.

<102> LINE NUMBER LIMIT EXCEEDED
A CTS line number larger than 262141 has either been entered or generated by a NUMBER command or an editing command. NUMBER mode is terminated or the editing command is terminated.

<103> SYNTAX SCAN IF OFF
The SEND-WORK-SPACE, SWS command, is used to cause the prescanner to scan the specified portion of the working area. The syntax scanner has been turned off. Enter *SYNTAX ON.

<104> Not Used.

<105> FILE (FILE NAME) IS EMPTY
Self explanatory. The file name specified is an internal name. If the name is not familiar, use LIST F to find the external name.

<106> Not Used.

<107> INTEGER OVERFLOW
Integer exponentiation has caused a number to exceed arithmetic limits.

<108> ILLEGAL VARIABLE NAME SYNTAX (CHARACTER STRING)
A variable which has a nonalphanumeric character or a nonalphabetic first character, or exceeds 12 characters in length has been referenced or defined. This can also occur when attempting to use the variable delimiter for command termination if the commentary information is not preceded by a blank.

<109> THE REQUESTED CYCLE FOR (FILE NAME.ELEMENT NAME) DOES NOT EXIST
The element referenced on the OLD or MERGE exists, but not with the cycle requested. OLD, with no cycle specified, will read the latest cycle of the element.

<110> SPECIFIED LINES DO NOT EXIST
There are no lines within the range of line numbers specified.

<111> ILLEGAL OR CONFLICTING OPTION SYNTAX (CHARACTER) STRING
Options other than A, R, S, I, or O have been specified or another option was used with the I on a COPY or TRANSFER command. The command was not performed.

<112> COMMAND WOULD DELETE ASSUMED PROGRAM FILE
A TRANSFER or UNSAVE command would cause the assumed program file to be purged. The command was not performed.

<113> TRANSFER PARAMETERS INVALID IN (ELEMENT OR DATA) MODE
A data file name must be specified in both the input and output fields of the transfer command when in DATA mode. The command was not performed.

<114> COPY PARAMETERS INVALID IN (ELEMENT OR DATA) MODE
A data file name must be specified in both the input and output fields of the COPY command when in DATA mode. The command was not performed.

<115> INPUT FILE IS NOT AVAILABLE
The file specified in the first field of a COPY or TRANSFER command is not available. The reason was given in the preceding messages. The command was not performed.

<116> Not Used.

<117> ILLEGAL SUBROUTINE COMMAND SYNTAX (CHARACTER STRING)
The specified line was not a valid CTS command. This error occurred while scanning the subroutine lines for valid syntax, prior to defining the subroutine. The CALL, SUB, BEGIN, or PROC was not performed.

<118> ILLEGAL RUN-ID SYNTAX (CHARACTER STRING)
A run-id cannot exceed six alphanumeric characters. The MAIL command was not performed.

<119> WORKSPACE WAS NOT CHANGED
The editing command was terminated when the line limit was exceeded and the work space was returned to the same state as before the command. Use a different command or resequence first.

<120> LINES EXCEEDING 262141 ARE LOST
The editing command was performed until completion and the resulting lines which exceed the maximum line number are lost. If there is a backup copy of the lost lines, they can be appended to the working area by first resequencing and then using MERGE.

<121> AN ELEMENT MAY NOT BE SPECIFIED
Only file names can be specified on this command because it is a file manipulation command or because CTS is in DATA mode. The operation was not performed.

<122> FILE (FILE NAME) IS ALREADY RELEASED
An attempt was made to release a file which was not assigned. Check spelling.

<123> FILE (FILE NAME) IS ALREADY CREATED
An attempt was made to create a file and there already was a file by that name.

<124> BACKWARD LINE LIMIT IS INVALID
Lines must be referenced in ascending order on an OLD or MERGE. They may be referenced in descending order on other commands. The OLD or MERGE was not performed.

<125> BACKWARD COLUMN LIMIT IS INVALID
Self explanatory.

<126> ILLEGAL COLUMN LIMIT SYNTAX (CHARACTER STRING)
A column limits specification contains invalid characters or references a column number not in the range of 1 to 132.

<127> A FILE NAME MAY NOT BE SPECIFIED

<128> TOO MANY FILLER CHARACTERS IN SECOND STRING (S2)

<129> A BLANK DATA LINE IS NOT ALLOWED

<130> PRESCANNER REQUIRES ASCII ON MODE

<131> PRESCANNER REQUIRES ASCII OFF MODE

<132> ASCII IMAGES WERE TRANSLATED

<133> FIELDATA IMAGES WERE TRANSLATED

<134> LINE LIMIT TOO SMALL

<135> LINE LIMIT TOO LARGE

<136> SUBROUTINE (SUBROUTINE NAME) IS ALREADY ACTIVE

<137> COMMAND (COMMAND) IS UNDEFINED

<138> ARGUMENT SYNTAX TO (FUNCTION NAME) IS BAD

<139> ZERO IS AN INVALID COLUMN LIMIT

<140> WARNING – AUTOMATIC RESEQUENCE THROUGH THE LAST LINE

<141> ILLEGAL TAPE TRANSLATION OR DATA CONVERTER SPECIFICATION

<142> UNABLE TO CONVERT ASCII CHARACTER CODE ; *character*

<143> END-OF-FILE CONDITION ENCOUNTERED ON QUERY

<144> SUBROUTINE *subroutine name* IS NOT DEFINED

<145> MAXIMUM PAGE LIMIT EXCEEDED

<146> INVALID READ/WRITE KEY COMBINATION

<147>NUMERIC ONLY, LESS THAN OR EQUAL TO 63

.

.

<170> INVALID FILE NAME SYNTAX ON @CTS CONTROL CARD

<171> CTS AUTOFILE IN USE BY ANOTHER RUN — AUTO RECOVERY DISABLED

.

.

<183> ENTER A UNIQUE IDENTIFIER FOR AUTO FILE

<184> STATEMENT NOT IN CLIST

<185> INCOMPLETE PRINTFILE

<186> PROCESSOR (PROCESSOR NAME) CAN NOT BE FOUND

<187> CANNOT SAVE WORK AREA

<188> UNABLE TO COMPLETE MAIL

<189> ALL MAIL FILES IN USE

<190> CANNOT RESTORE WORK AREA

<191> PROGRAM (PROGRAM NAME) CAN NOT BE FOUND

<192> *ERROR – INVALID EXPRESSION OR UNDEFINED VARIABLE FOR (VARIABLE NAME)

<193> *ERROR – THE INCREMENT FOR (VARIABLE NAME) IS ZERO

<194> *ERROR – THE INCREMENT FOR (VARIABLE NAME) IS POSITIVE BUT THE START VALUE IS GREATER THAN THE END VALUE

<195> *ERROR – THE INCREMENT FOR (VARIABLE NAME) IS NEGATIVE BUT THE START VALUE IS LESS THAN THE END VALUE

<196> NO SCANNER FOR THIS PROCESSOR

<197> CTS$FILE OVERFLOWED.
Reserve CTS$FILE space is being used to allow the current command to complete without loss of the work area. The user should save the work area, and reinitialize CTS (i. e., @CTS,I) so a fatal CTS$FILE overflow can be avoided.

<198> BAD SCAN FILE

<199> CANNOT ASSIGN SPECIFIED FILE

<200> INTERNAL ERROR AT (ADDRESS): A9 = (OCTAL NUMBER)

Index

Term	Reference	Page
A		
ABS(x) function	12.1.4	12-2
ADD command	7.5.7	7-13
Alternate print file	4.3.8	4-16
APF function	9.1.3	9-2
APF( ) function	Table 12-2	12-5
APL		
access to	2.4.4.1	2-38
processor options	2.4.4.2	2-38
running with CTS	2.4.4	2-37
statements	2.4.4.3	2-39
Arithmetic operations	12.1.6	12-10
ASCII command	2.4.2	2-19
ASSUME commands		
ASCII	2.2.9	2-9
AUTO	2.2.10	2-9
BREAKPOINT	4.3.8	4-16
BRIEF	5.1.3	5-7
CALL FILE	8.4.1.1	8-25
CHECKOUT	11.2.3.2	11-18
COLUMN	2.2.1	2-4
COMPILER	6.2.1	6-9
COPY	4.3.6	4-15
DEBUG	11.2.1	11-4
ECHO	5.1.5	5-8
ECOLUMN	2.2.2	2-4
EDIT	2.3.5	2-16
FILE	6.2.2	6-10
FILLER	5.1.6	5-9
	5.1.7	5-9
HEADING	4.3.4	4-15
INPUTWIDTH	4.2.3.2	4-13
JUMP	8.4.3.4	8-30
LIBRARIES	6.4.2.2	6-18
MAIN	6.4.2.1	6-18
MAP	6.4.2.3	6-19
OBJECT	6.2.3	6-11
OCOLUMN	2.2.3	2-5
PCOLUMN	2.2.4	2-5
POLL	2.3.1	2-11
PRINTWIDTH	4.3.1	4-14
PROGRAM	6.2.4	6-11
QUICK	4.3.2	4-14
RELOCATABLE	6.2.5	6-13
RESEQUENCE	5.3.6	5-26
RETURN	4.3.5	4-15
SAVELENGTH	3.2.3	3-9
SBUG	8.4.3.1	8-27
SCOLUMN	2.2.5	2-5
SITE	4.3.3	4-14
STRING	5.1.8	5-9
TRACE	8.4.3.3	8-29
TYPE	4.3.7	4-15
XQT	6.3.2	6-15
Assumed object file name of	9.1.3	9-2
Assumed program file, name of	9.1.3	9-2
ATN(x) function	12.1.4	12-2

Term	Reference	Page
Attached names	4.1.4.3	4–9
Automatic line numbering		
generation	2.3.2	2–12
termination	2.3.3	2–15
B		
BASIC debugging		
BREAK key	11.2.2.2	11–7
PAUSE statement	11.2.2.1	11–4
TRACE statements	11.2.2.3	11–9
Batch processing	13.1	13–1
BBASIC (BASIC prescan module)	2.4.1	2–17
BCOB prescanner	2.4.3	2–27
automatic continuation	2.4.3.1	2–27
conversational mode	2.4.3.2.2	2–32
diagnostics	2.4.3.1	2–27
Edit mode	2.4.3.2.1	2–31
margins	2.4.3.1	2–27
operational description	2.4.3.1	2–27
program file mode	2.4.3.2.3	2–37
BCOB (COBOL prescanner)	2.4.3	2–27
BFOR (Fieldata FORTRAN prescan module)	2.4.2	2–19
BFTN (ASCII FORTRAN prescan module)	2.4.2	2–19
Blank lines, in output listing	4.1.3	4–5
BRANCH command	8.3.5.3	8–12
C		
CALL command	8.4.1	8–24
CALL FILE	8.4.1.1	8–25
Calling CTS	1.3	1–1
CARDS command	4.2.2	4–11
Central Processor Time accumulated	9.2.4	9–4
CHANGE command	5.2.2	5–14
CHECK command	2.4.2.5	2–26
	2.4.3.2.3	2–37
Checkpoint saves	2.2.10	2–9
CHOP command	2.4.3.1	2–27
CLEAR command	2.4.7	2–42
COBOL prescanner	2.4.3	2–27
Collection	6.1.1	6–1
	6.1.2	6–2
COL( ) function	12.1.4	12–2
Combining programs	3.6	3–17
Comments	8.3.7.6	8–21
Compatibility, CTS and batch processing	13.3	13–4
COMPILE command	6.4.1	6–16
Compile, collect, and execute a program	6.2	6–4
Compile–and–go	6.1.1	6–1
Compressing terminal output	4.3.2	4–14
Compression of spaces	4.1.2	4–5
COMP( ) function	Table 12–2	12–5
Control characters	1.5	1–6
Control statements submitting via CSF$ interface	7.6	7–14
Controlling the solicitation sequence	2.3.1	2–11

Term	Reference	Page
COPY command	7.5.4	7–10
Copying a line	5.3.3	5–24
COS(x) function	12.1.4	12–2
COT(x) function	12.1.4	12–2
CPTIME command	9.2.4	9–4
CREATE command	7.5.1	7–7
Creating a program	2.1	2–1
Creating an absolute element	6.4.2	6–17
Creating lines of data in f	2.3	2–10
Creating relocatable elements	6.4.1	6–16
Creation of a file	3.2.2	3–7
CR( ) function	12.1.4	12–2
CSEQ command	2.4.3.1	2–27
CSF command	7.6 13.2.2	7–14 13–2
CSF(e) function	Table 12–2	12–5
CTS messages	Appendix B	
Current date and time	9.2.3	9–4
Cycle	4.1.4.3	4–9
C( ) function	12.1.4	12–2

D

Term	Reference	Page
DATA command	3.2 3.7	3–1 3–21
Data files	3.1	3–1
DATE command	9.2.3	9–4
DATE( ) function	Table 12–2	12–5
DEBUG mode	11.2.1	11–4
Debugging source code	11.2	11–4
Defining an onsite device	4.3.3	4–14
Defining the length of a displayed line	4.3.1	4–14
Delete a copy	3.4	3–11
DELETE command	5.2.1	5–11
Deleting files	7.5.2	7–8
Deleting lines	5.2.1	5–11
DIAG command	2.4.2.6	2–26
Diagnostics	Appendix B	
Discarding a copy	3.4	3–11
DISPLAY command	12.4	12–12
Displaying a program	4.1.1	4–1
Displaying line numbers of matched lines	5.1.4	5–8
Displaying matched lines	5.1.3	5–7
Displaying names of saved elements	4.1.4.2	4–6
Displaying the names of assigned files	4.1.4.3	4–9
DITTO command	5.3.3	5–24
DKN( ) function	9.1.3 Table 12–2	9–2 12–5
DML COBOL	2.4.3.1	2–27
DROP command	8.3.8 12.6	8–23 12–14

Term	Reference	Page
E		
ECHO command	2.4.2.6	2–26
EDIT command	11.1.2	11–3
Editing a line	5.2.3	5–18
Elements	7.1.1	7–1
END command	8.3.7.3	8–18
Entering data images	2.1	2–1
Entering FTN interactive debug mode	11.2.3.4.1	11–21
ENTRY command	8.3.7.1	8–16
Equipment type	4.1.4.3	4–9
ERASE command	7.5.8	7–14
ERROR command	8.3.5.1	8–9
Examining processor output	11.1.1	11–2
Executing a program	6.1.1	6–1
Executing an absolute element	6.3.1	6–14
EXIT command	8.3.7.7	8–22
Exiting from CTS	1.4	1–5
Explanation of commands	9.3.1	9–5
Expressions, evaluating and printing	12.3	12–11
EXP(x) function	12.1.4	12–2
F		
FD command	2.4.2	2–19
Fieldata character mode	2.2.9	2–9
File information	9.1	9–1
FILLER	5.1.1 5.2.2	5–1 5–14
FILLER default for LOCATE command	5.1.6	5–9
FIND command	5.1.2	5–5
Finding a string	5.1.1 5.1.2	5–1 5–5
FMT(x,w,d[,'s']) function	Table 12–2	12–5
Forms of L	2.2.8.2	2–7
FORTRAN prescanner		
abbreviated key words	2.4.2.3	2–23
automatic formatting	2.4.2.1	2–21
automatic global syntax analysis	2.4.2.4	2–24
continuation lines	2.4.2.2	2–22
controlling global syntax analysis	2.4.2.6	2–26
global syntax analysis	2.4.2.5	2–26
FOUND command	8.3.5.2	8–11
FTN	2.4.2	2–19
FTN CHECKOUT mode	11.2.3.2	11–18
contingencies	11.2.3.5	11–34
interactive postmortem dump	11.2.3.6	11–35
restrictions	11.2.3.5	11–34
walkback	11.2.3.6	11–35
FTN debug facility	11.2.3.1	11–12
AT statement	11.2.3.1.1	11–13
DEBUG statement	11.2.3.1.1	11–13
DISPLAY	11.2.3.1.5	11–17
example	11.2.3.1.5	11–17
INIT	11.2.3.1.1	11–13
SUBCHK	11.2.3.1.1	11–13
SUBTRACE	11.2.3.1.1	11–13
TRACE	11.2.3.1.1	11–13
TRACE OFF statement	11.2.3.1.4	11–16
TRACE ON statement	11.2.3.1.3	11–16
UNIT	11.2.3.1.1	11–13

Term	Reference	Page
FTN interactive debug mode		
BREAK command	11.2.3.4.1	11-21
CALL command	11.2.3.4.2	11-22
CLEAR command	11.2.3.4.3	11-23
DUMP command	11.2.3.4.4	11-24
EXIT command	11.2.3.4.5	11-25
GO command	11.2.3.4.6	11-25
HELP command	11.2.3.4.7	11-26
ISN command	11.2.3.4.8	11-26
PROG command	11.2.3.4.10	11-26
RESTORE command	11.2.3.4.11	11-27
SAVE command	11.2.3.4.12	11-28
SET command	11.2.3.4.13	11-29
SETBP command	11.2.3.4.14	11-30
SNAP command	11.2.3.4.15	11-31
Soliciting input	11.2.3.3.2	11-20
STEP command	11.2.3.4.16	11-31
TRACE command	11.2.3.4.17	11-31
WALKBACK command	11.2.3.4.18	11-32
FTN interactive debug mode, entering	11.2.3.3.1	11-19
Functions		
general	12.1.4	12-2
numeric	Table 12-1	12-3
string	Table 12-2	12-5

G

GENERATE command	8.3.7.4	8-18
GLOBAL command	2.4.2.6	2-26
Global Scan	2.4.2.4	2-24
GO command	8.3.7.5	8-20

H

HELP command	9.3.1	9-5
HELP subcommands		
COMMANDS	9.3.1	9-5
EXIT	9.3.1	9-5
EXPLAIN	9.3.1	9-5
	9.3.2	9-6
FIELD i	9.3.1	9-5
LENGTH	9.3.1	9-5
SYNTAX	9.3.1	9-5
TEACH	9.3.1	9-5
USE	9.3.1	9-5

I

Information	9.3.1	9-5
Initialization of CTS	1.3.1	1-2
Initialization subroutine	1.3.3	1-5
INLINE command	5.2.3	5-18
INSERT command	5.2.4	5-19
Inserting strings	5.2.19	5-19
Integer constants	12.1.1	12-1
Internal file name	7.5.5	7-11
	9.1.3	9-2
Interpreter	6.1.1	6-1
Iterative expression		
evaluation	12.4	12-12
summation	12.5	12-13

J

JUMP command	8.3.5	8-9

K

Keys	7.4	7-7

L

LENGTH command	9.2.2	9-3
LEN(s) function	12.1.4	12-2
LGT(x) function	12.1.4	12-2
Line Numbers	2.2.6	2-6
specifying a range	2.2.8.2	2-7
specifying a sequence	2.2.8.7	2-7
Line numbers, display of	4.3.7	4-15

Term	Reference	Page
LIST command		
CATALOG	9.1.1	9–1
FILE	9.1.2	9–2
LIST INUSE	4.1.4.3	4–9
LIST L	4.1.4.1	4–6
LIST SAVED	4.1.4.2	4–6
LNG( ) function	12.1.4	12–2
LOAD command	2.4.3.1	2–27
LOCATE command	5.1.1	5–1
Logical file name	7.5.5	7–11
LOG(x) function	12.1.4	12–2
LOOK command	10.3.2	10–5
LOWER('s') function	Table 12–2	12–5
M		
MAIL command	10.3.1	10–4
Making a permanent copy	3.2	3–1
MANUAL command	2.3.3	2–15
MAP command	6.4.2	6–17
MAP directives	6.4.2.3	6–19
Mapping	6.1.1	6–1
Mass storage files		
drum or disk	7.3	7–3
examples of file usage	7.7	7–15
private	7.2	7–3
program file	7.1.1	7–1
Public	7.2	7–3
SDF	7.1.1	7–1
tape	7.3	7–3
temporary	7.2	7–3
use by CTS	7.1.2	7–2
MERGE command, line numbers	3.6.1	3–18
MOVE command	5.3.4	5–25
Moving a line	5.3.4	5–25
N		
NAME$	6.2	6–4
Naming a program	5.3.1	5–22
Naming relocatable element	6.2.5	6–1
NEW command	5.3.1	5–22
News File	1.3.2	1–4
NOCHOP command	2.4.3.1	2–27
NODIAG command	2.4.2.6	2–26
NOECHO command	2.4.2.6	2–26
NOGLOBAL command	2.4.2.6	2–26
NSEQ command	2.4.3.1	2–27
NUMBER command	2.3.2	2–12
Number of lines in f	9.2.2	9–3
NUM(n,c1,c2) function	12.1.4	12–2
O		
OBJ function	9.1.3	9–2
OBJ( ) function	Table 12–2	12–5
OLD command		
assumed compiler and options	3.5	3–13
changes to working area	3.5	3–13
Omnibus element	8.5.1	8–34
Online assistance	9.3	9–5
Operating information	9.2	9–3
Operator communications	10.2.1	10–1

Term	Reference	Page
Operators		
arithmetic	12.1.6	12–10
relational	12.1.6	12–10
OPR command	10.2.2	10–3
Output to paper tape	4.2.3	4–12
Output to punched cards	4.2.2	4–11
P		
PACK command	7.5.6	7–12
Paper tape input	4.2.3.1	4–13
Paper tape line length	4.2.3.2	4–13
Paper tape output	4.2.3	4–12
Positioning parameter, INSERT command	5.2.4	5–19
Preparing libraries	6.4.2.2	6–18
Prescan module		
controlling syntax checking	2.4.6	2–41
terminating control	2.4.7	2–42
Prescan modules	2.4	2–17
PRINT command	4.1.1	4–1
PROC command	8.2.3	8–6
Processor call, from within CTS	6.5	6–19
Processor options	6.1.2	6–2
Program failure	11.1	11–3
Pseudo-text, in COBOL	2.4.3.1	2–27
PTI command	4.2.3.1	4–13
PUNCH command	4.2.3	4–12
PURGE command	7.5.2	7–8
PXQT command	6.5	6–19
P( ) function	12.1.4	12–2
Q		
Qualifier	4.1.4.3	4–9
QUERY command	8.3.3	8–8
	12.1.5	12–9
QUICK command	4.1.2	4–5
R		
Real constants	12.1.2	12–1
RELEASE command	7.5.3	7–9
Relocatable elements	6.1.1	6–1
Removal of a variable or subroutine	8.3.8	8–23
RENAME command	5.3.5	5–26
Renumbering line numbers	5.3.2	5–23
REPLACE command	3.3	3–9
Replacing strings	5.2.2	5–14
Reprinting of lines keyed into CTS	5.1.5	5–8
RESEQUENCE command	5.3.2	5–23
Resolving line number conflicts	5.3.6	5–26
Retrieving a copy	3.5	3–13
RETURN command	8.3.7.2	8–17
RFOR	2.4.2	9–4
RFOR debugging		
BREAK key	11.2.4.2	11–38
PAUSE statement	11.2.4.1	11–35
TRACE statements	11.2.4.3	11–39

Term	Reference	Page
S		
SAVE command	3.2	3–1
Save file	6.2.4	6–11
Saving a data file	3.2.2	3–7
Saving a program	3.2.1	3–2
SCAN command	11.1.1	11–2
Scan file	7.1.2	7–2
	11.1	11–1
Security	7.4	7–7
Send print to an alternate file	4.3.8	4–16
Sending output to an onsite device	4.2.1	4–10
SET command	8.3.2	8–7
	12.1.5	12–9
	12.2	12–10
Setting the character mode	2.2.9	2–9
Setting the maximum length of a saved line	3.2.3	3–9
SIN(x) function	12.1.4	12–2
SITE command	4.2.1	4–10
SKIP command	4.1.3	4–5
SPACER	5.1.1	5–1
	5.2.2	5–14
SPACER default for LOCATE command	5.1.7	5–9
Specifying libraries	6.4.2.2	6–18
Specifying Part of a Line	2.2.1	2–4
Specifying the absolute element	6.3.2	6–15
Specifying the assumed compiler	6.2.1	6–9
Specifying the main program	6.4.2.1	6–18
Specifying the object file	6.2.3	6–11
Specifying the SAVE and OBJECT file	6.2.2	6–10
SQR(x) function	12.1.4	12–2
SQUELCH$	6.2	6–4
	11.1	11–1
SSUB command	8.5.1	8–34
@@START command	13.2.3	13–4
@START command	13.1	13–1
	13.2.1	13–2
STATUS command	9.2.5	9–4
String constants	12.1.3	12–2
SUBROUTINE command	8.2.2	8–3
Subroutines	8.1	8–1
ASSUME SBUG	8.4.3.1	8–27
BRANCH command	8.3.5.3	8–12
building	8.2	8–2
CALL command	8.4.1	8–24
CALL Parameter	8.4.2	8–25
comments	8.3.7.6	8–21
debugging	8.4.3	8–27
defining entry point	8.3.7.1	8–16
displaying variables	8.4.3.6	8–33
END command	8.3.7.3	8–18
ENTRY command	8.3.7.1	8–16
ERROR command	8.3.5.1	8–11
execute	8.4.1	8–24
EXIT command	8.3.7.7	8–22
exiting	8.3.7.2	8–17
FOUND command	8.3.5.2	8–11
GENERATE command	8.3.7.4	8–18
GO command	8.3.7.5	8–20
JUMP command	8.3.5	8–9
nesting	8.4.3.7	8–33
programmable editor	8.6.2	8–37
programming	8.3	8–7
QUERY command	8.3.3	8–8

Term	Reference	Page
RETURN command	8.3.7.2	8–17
saving	8.2.1	8–3
	8.5	8–34
SET command	8.3.2	8–7
setting the line pointer	8.3.7.5	8–20
starting batch runs	8.6.3	8–39
subroutines calling other subroutines	8.2.2	8–3
subscripting variables	8.4.2	8–25
trace	8.4.3.2	8–28
transferring control	8.3.5	8–9
	8.3.5.3	8–12
TYPE command	8.3.4	8–9
variable substitution	8.3.6	8–13
variables	8.3.1	8–7
Subscripting variables	8.4.2	8–25
Symbolic element types	3.2.1	3–2
SYNTAX command	2.4.6	2–41
System stop	2.2.10	2–9
T		
TAB characters	2.3.4	2–16
TAB command	2.3.4	2–16
TAB stops	2.3.4	2–16
TAB([n]) function	Table 12–2	12–5
TAN(x) function	12.1.4	12–2
TB message	10.2.1	10–1
Terminating scan mode	11.1.2	11–3
Termination character	5.2.3	5–18
TM message	10.2.1	10–1
Tracing, FTN program execution	11.2.3.1.3	11–16
Tracing, in BASIC program	11.2.2.3	11–9
Transparent control statements	13.2.3 Appendix A	13–4
TRM(s) function	Table 12–2	12–5
TXT(n,c1,c2) function	Table 12–2	12–5
TXT(s,c1,c2) function	Table 12–2	12–5
TYPE command	8.3.4	8–9
	12.1	12–1
	12.3	12–11
U		
UNSAVE command	3.4	3–11
Updating a copy	3.3	3–9
UPPER('s') function	Table 12–2	12–5
USE command	7.5.5	7–11
USER$	1.3.3	1–5
User/operator communications		
@MSG	10.2	10–1
OPR command	10.2.2	10–3
User/user communications		
LOOK command	10.3.2	10–5
MAIL command	10.3.1	10–4
V		
Variable definition	12.2	12–10
Variable delimiter character	5.2.2	5–14
Variable substitution	8.3.6	8–13
Variables	12.1.5	12–9
W		
Working area, name of	9.1.3	9–2

Term	Reference	Page	Term	Reference	Page
X					
XCTS command	1.4	1–5			
XQT command	6.3.1	6–14			

UNISYS

USER COMMENTS

We will use your comments to improve subsequent editions.

NOTE: Please do not use this form as an order blank.

_____

(Document Title)

_____ _____ _____

(Document No.) *(Revision No.)* *(Update Level)*

Comments:

From:

_____

(Name of User)

_____

(Business Address)

Fold on dotted lines, and mail. (No postage is necessary if mailed in the U.S.A.)
Thank you for your cooperation

CUT

FOLD

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 1145 ST. PAUL, MN

POSTAGE WILL BE PAID BY ADDRESSEE

Unisys Corporation
Large Systems Product Information
P.O. Box 64942
St. Paul, MN 55164-0942

FOLD