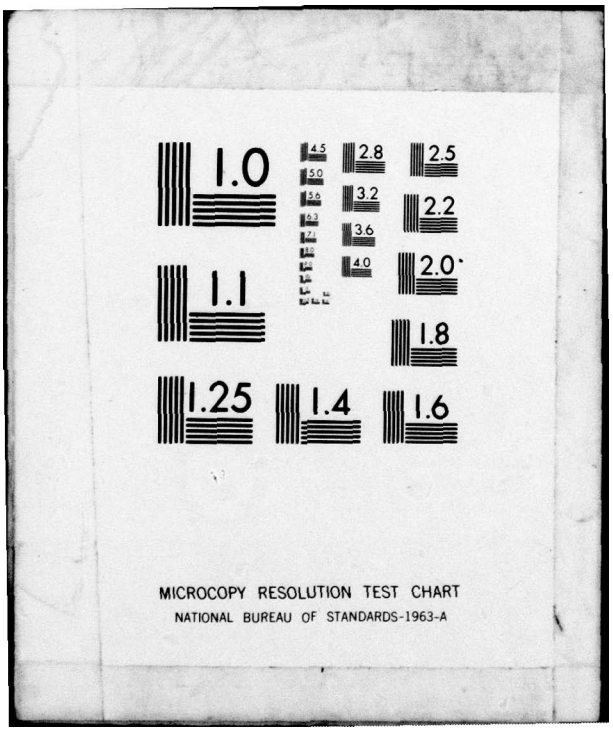
AD-A080 625 PATTERN ANALYSIS AND RECOGNITION CORP ROME N Y F/G 9/2
MULTICS REMOTE DATA ENTRY SYSTEM. VOLUME I.(U)
OCT 79 D BIRNBAUM, J J CUPAK, J D DYAR F30602-77-C-0174
UNCLASSIFIED PAR-79-89 RADC-TR-79-268-VOL-1 NL

1 OF 3
AD
A080625


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

RADC-TR-79-265, Vol I (of two)
Final Technical Report
October 1979

MULTICS REMOTE DATA ENTRY SYSTEM

LEVEL II

Pattern Analysis & Recognition Corporation

David Birnbaum
John J. Cupak, Jr.
Janet D. Dyar
Richard Jackson

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

This report has been reviewed by the RADC Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RADC-TR-79-265, Vol I (of two) has been reviewed and is approved for publication.

APPROVED: Patricia J. Baskinger

PATRICIA J. BASKINGER
Project Engineer

APPROVED: Wendall C Bauman

WENDALL C. BAUMAN, Col, USAF
Chief, Information Sciences Division

FOR THE COMMANDER: John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISCP), Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER: RADC-TR-79-265, Vol I (of two)	2. GOVT ACCESSION NO.
3. RECIPIENT'S CATALOG NUMBER	
4. TITLE *(and Subtitle)*: MULTICS REMOTE DATA ENTRY SYSTEM, Volume I.	5. TYPE OF REPORT & PERIOD COVERED: Final Technical Report.
6. PERFORMING ORG. REPORT NUMBER: PAR-79-59	
7. AUTHOR(s): David Birnbaum, John J. Cupak, Jr., Janet D. Dyar, Richard Jackson	8. CONTRACT OR GRANT NUMBER(s): F30602-77-C-0174
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Pattern Analysis & Recognition Corporation, 228 Liberty Plaza, Rome NY 13440	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 62702F, 55971324
11. CONTROLLING OFFICE NAME AND ADDRESS: Rome Air Development Center (ISCP), Griffiss AFB NY 13441	12. REPORT DATE: Oct 1979
	13. NUMBER OF PAGES: 259
14. MONITORING AGENCY NAME & ADDRESS *(if different from Controlling Office)*: Same	15. SECURITY CLASS. *(of this report)*: UNCLASSIFIED
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE: N/A

16. DISTRIBUTION STATEMENT *(of this Report)*

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT *(of the abstract entered in Block 20, if different from Report)*

Same

18. SUPPLEMENTARY NOTES

RADC Project Engineer: Patricia J. Baskinger (ISCP)

19. KEY WORDS *(Continue on reverse side if necessary and identify by block number)*

Pattern Recognition
Pattern Analysis
Classification
Clustering
OLPARS
Data Entry

20. ABSTRACT *(Continue on reverse side if necessary and identify by block number)*

This report contains the user's manuals and software documentation for the Remote Data Entry System which is the front-end to the MULTICS Pattern Recognition Facility and the Cluster Analysis package which was added to MULTICS OLPARS. The Remote Data Entry System was designed to allow users of the MULTICS Pattern Recognition Facility the ability to input their data over the ARPANET from a Tektronix remote storage device. Once the data is input into the MULTICS System, routines are provided so that the user can easily restructure or cluster his database to perform different classification experiments.

DD FORM 1 JAN 73 1473

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE *(When Data Entered)*

390 101

FINAL REPORT REMOTE DATA ENTRY

Preface

This is a Final Report by Pattern Analysis and Recognition Corporation, 228 Liberty Plaza, Rome, New York, and it represents work performed under Contract F30602-77-C-0174, Job Order Number 55971234 for the Rome Air Development Center, Griffiss Air Force Base, New York.

Mrs. Patricia J. Baskinger was the RADC Project Engineer.

TABLE OF CONTENTS

TABLE OF CONTENTS

EVALUATION

This report contains the user's manual and software documentation for the MULTICS Remote Data Entry System (RDE) developed under Contract F30602-77-C-0174. The RDE System was developed to allow the users of the MULTICS Pattern Recognition Facility the ability to enter data remotely from a Tektronix 4921 floppy disk or a 4923 tape cassette which are attached to their 4014 terminal.

The MULTICS Pattern Recognition Facility, which includes the MULTICS versions of WAVES, a waveform processing system, and OLPARS (On-Line Pattern Analysis and Recognition System), is available to remote users via access over the ARPANET.

Once the data is in the MULTICS System, programs are provided which give the user the ability to restructure the data set on-line or to perform clustering analysis on the data set. Volume I of this report contains the documentation for RDE, and Volume II contains the user's manual for the clustering analysis additions to MULTICS OLPARS. This system is a major enhancement to the MULTICS Pattern Recognition Facility which provides the Air Force with a powerful capability for solving a wide range of target identification problems in the areas of command, control, communications and intelligence.

Patricia J. Baskinger

PATRICIA J. BASKINGER
Project Engineer

1 INTRODUCTION

OLPARS (On Line Pattern Analysis and Recogntion System) was conceived and implemented by RADC and PAR corporation as an interactive pattern analysis and recognition tool. Implementation of OLPARS on the Honeywell Information Systems (HIS) 6180 Computer facility at RADC (MOOS - MULTICS OLPARS Operating System), has made the system available through the ARPANET. Consequentially, users throughout the country have used OLPARS in the solution of their pattern recognition problems.

The Remote data Entry (RDE) system was designed to permit persons using OLPARS through the ARPANET to transmit and receive vector and waveform data to and from a remote storage device. Waveform data to be analyzed using WAVES (Long Waveform Analysis System) may also be transmitted and received using the RDE system.

This report contains a description of the RDE system, its use, and the programs it contains.

The RDE user's manual is described in section 2 and gives complete instructions on how to send data back and forth to MOOS and WAVES. The Command Translator System (CTS) is described in Section 3. CTS is designed to enable a user to edit his data once it is in the MULTICS system. Sections 4 and 5 give functional descriptions of the

RDE and CTS systems and are designed for software maintenance personnel. Section 4 also contains a description of floating point conversion routines that were implemented under this effort.

2 RDE USER'S MANUAL

2.1 General Remarks

This section contains descriptions of all RDE user functions. It is designed to provide a novice user of the system with sufficient information to allow for easy use of the system capabilities. Description of the computions performed by each system program is documented in section 4.

The standard terminal from which RDE commands are executed is the Tektronix 4014-1 storage tube display. Attached to the 4014-1 are a 4921 single disk drive, a 4922 double disk drive, or a 4923 tape cassette device. The 4923 device can control the 4014-1 screen only if there is an attached 4921/4922 device. A hard-copy unit may also be attached to the 4014-1 terminal, but is not essential for the operation of the RDE system.

User function calls are entered through the terminal keyboard, and consist of simple program names followed by any required or optional parameters. Within the system programs, dialogue concerning additional information required for program operation is handled by standard terminal input/output operations as specified in this section.

In the MULTICS system, program-calling arguments are specified as "-arg" or "-arg parameter". The first form is used to set or reset specific program switches or perform specific action. The second form is generally used to specify a file, where the "-arg" identifies the file type.

For example, the program dumptree may be called as:

```
dumptree -in treepath -out datapath
```

This convention was used in the RDE system. Missing arguments are solicited from the user. The possible arguments are described in the individual program descriptions.

To create an OLPARS tree from the data on a remote device:

- The user calls the program "remote_to_multics" (rtom).
- The program "rtom" invokes the program "remote_data_" to control I/O.
- Data is transmitted to an ASCII data file in the users' default working directory (the directory he logs on under).
- The user may elect to edit the data file and/or create

subfiles by using the program "Command _Translator_System" (CTS).

- The user calls the program "maketree" to create an intermediate file in the users' process (temporary) directory.

- The maketree program calls the program "treeiput" to create a MOOS tree file in the users' process directory.

The creation of an OLPARS tree from data stored on a remote device is shown in Figure 1.

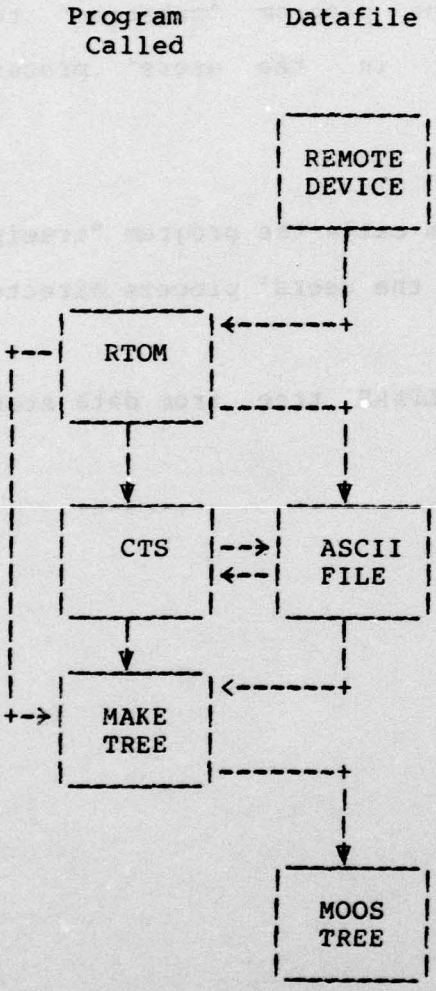


Figure 1 Remote Data File to MOOS Tree

To transfer an OLPARS tree to a remote device:

- The user calls the program "dumptree".
- The program "dumptree" calls the program "treeoput" to convert a MOOS treefile to an intermediate file in the users' process directory.
- The program "dumptree" then converts the intermediate file to an ASCII datafile in the users' default working directory.
- The user may elect to edit the data file using the program "CTS".
- The user calls the program "multics_to_remote" (mtor).
- The program "mtor" invokes the program "remote_data_" to control I/O.
- The data is transferred from the ASCII data file to the remote device.

The creation of a remote device data file from a MOOS tree is

shown in Figure 2.

To create a WAVES tree the user first calls rtom to create the ASCII file in his working directory and then calls the WAVES command unpack_tree. This command then creates the WAVES tree. To transfer a WAVES tree to a remote device the WAVES command pack_tree is used followed by the RDE command mtor. CTS may also be used to edit the intermediate file. Description of the WAVES commands can be found in Sections 2.7 and 2.8 and in the Final Technical Report to the MULTICS Long Waveform Analysis System, RADC-TR-78-218.

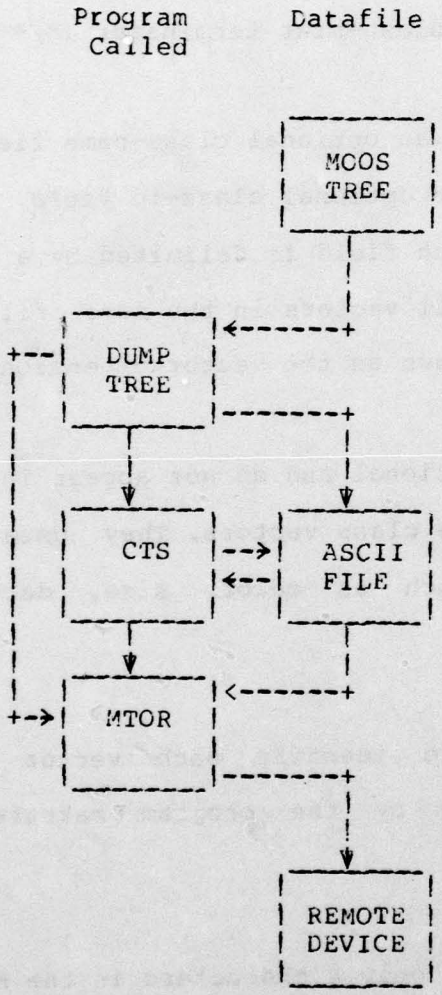


Figure 2 MOOS Tree to Remote Data File

2.2 ASCII data file format

Figure 3 shows the format of the ASCII data file. The file consists of a series of vectors, separated by semi-colons (";"). The last vector is followed by a slash-star terminator ("/*").

Each vector consists of an optional class-name field, any number of optional keyword fields, an optional class-id field, and a fixed number of data fields. Each field is delimited by a blank, a comma (","), or both characters. All vectors in the same file consist of the same number of fields, known as the vector dimension.

Keywords, which are optional and do not appear in the MOOS tree file, are used to describe the class vectors. They usually represent human-readable attributes such as color, size, date, time, data source, etc.

The id numbers serve to identify each vector uniquely. If missing, they are generated by the program maketree for use by treeiput.

Class names, which may be only 4 characters in the MOOS file, can be inserted by the CTS program if not present in the ASCII file. Any attempt to use the ASCII data file without the class names by maketree would result in errors.

An example of two vectors which contain all the fields is:

```
class1,car,1,3.7,4.9;class2,tank,2,3.2,5.7;/*
```

An example of two vectors containing only data is:

```
3.7,4.9;3.2,5.7;/*
```

The ASCII data is written on the remote device storage media in 128-character blocks. The last data block must be padded with stars ("*") if it contains less than 128 characters, including the "slash-star" terminator ("/*").

NOTE: Data is written to the 4923 tape cassette device in 128 character blocks containing 126 data characters, a "stop read" control character (DC3), and a "dispensable" character (LF). The last block indicating end of file consists of 126 stars ("*"), a "stop device" character (DC4), and a "dispensable" character (LF). The device control characters (DC3 and DC4) permit the rtom program to read one block at a time from the device. The LF character signals end of transmission to the terminal.

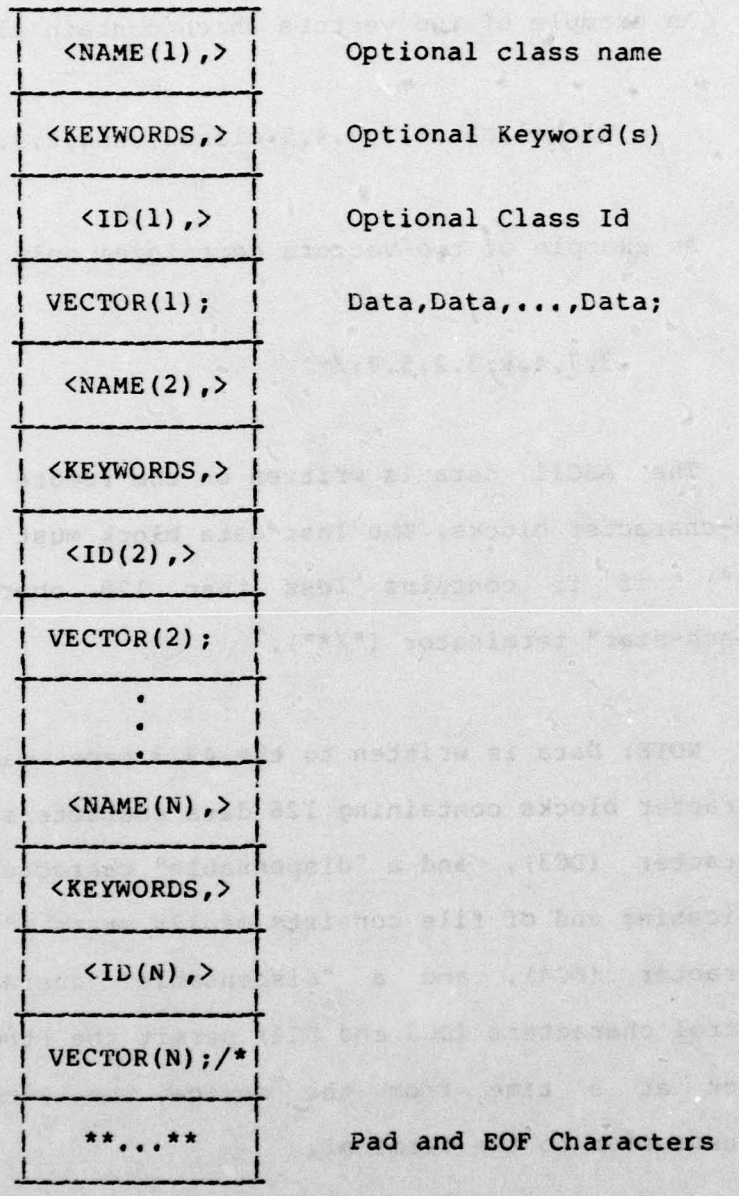


Figure 3 ASCII data format

2.3 remote_to_multics, rtom

Function: rtom (device) (-debug)

Parameters:

device	name of attached remote device
-debug, -db	test switch

Description:

The user function rtom transfers an ASCII data file from a remote device to a MULTICS file.

The test switch "-debug" or "-db" is used by maintenance personnel to trace the program execution and data values. It should not be specified by the normal user.

User Interaction:

If the remote device is incorrectly specified on the call, the following error message is displayed:

(1) Remote device must be "4921", "4922", "4923", or "tape".

If the remote device is incorrect or missing, the following request will be displayed:

(2) Enter remote device type: xxxxLF

Type in the remote device type followed by a line-feed (LF).

After the information has been accepted, the system will display the following message:

(3) Enter output data file name: nameLF

Type in the desired output file name followed by a LF.

If the output data file exists in the user's default working directory, the following message is displayed:

(4) Data file already exists. Do you wish to overwrite it? replyLF

Type in "yes" or "no", followed by a LF. A reply of "no" will cause the following question to be asked:

(5) Do you wish to continue? replyLF

Type in "yes" or "no", followed by a LF. A reply of "no" terminates the program. A reply of "yes" displays the following message before returning to question (3).

Choose another file name.

If the remote device is "4922", the following question is asked:

(6) Enter disk number: nLF

Type a "1" or a "2", followed by a LF.

(7) Do you want the screen on during data transmission? replyLF

Type "yes" or "no", followed by a LF. The user should be aware that the screen can display only 8448 characters (64 lines times 132 chars per line) before overwriting the displayed information. This represents approximately 66 sectors, or 2 tracks of data.

If the remote device is "4921" or "4922", questions (8) and (9) are displayed.

(8) Enter starting track number: nnLF

Type a two-digit number between 00 and 63, followed by a LF.

(9) Enter starting sector number: nnLF

Type a two-digit number between 00 and 31, followed by a LF.

If the remote device is "4923", question (10) is displayed.

(10) Do you wish instructions? replyLF

Type "yes" or "no", followed by a LF.

After the instructions are displayed, the user is asked if he is ready to continue. Type a "yes", followed by a LF. The program will then display the following message, and wait for 20 seconds for the user to comply before initiating data transfer:

Depress the RUN button once.

If the remote device is "tape", questions (11) through (17) will be asked.

(11) Enter volume id: VVVVVVLF

Type 6-digit tape reel number with leading zeroes, followed by a LF.

(12) Enter optional console message: messageLF

Type LF for no message, or up to 64 characters (including blanks) followed by a LF. A message greater than 64 characters will be truncated.

(13) Enter file name: nameLF

Enter up to 17 characters followed by a LF. A name field greater than 17 characters will be truncated, and a message informing the user will be printed.

At this point, all required information has been entered or generated. If the user wishes to change fields, enter new fields, or delete any fields in the tape attach description, the program will permit editing. The following question is asked automatically:

(14) Do you wish to change any fields? replyLF

Type "yes" followed by a LF if you wish to edit any of the fields in the attach description.

If the user types "yes" in response to question (14), the program continues with questions (15) through (17).

(15) Enter literal to be changed: literalLF

Type in any string to be changed, followed by a LF (the literal string may contain blanks).

(16) Enter literal to be changed to: literalLF

Type in replacement literal (a null string or a literal with imbedded blanks is permissible), followed by a LF.

The edited attach description is now displayed.

(17) Do you wish to continue editing? replyLF

Type "yes" followed by a LF to continue. The program will then display questions (15) through (17) again. Type "no" to terminate editing.

After completion of the data transfer, the program will display the number of 128-character records read. If the remote device was "tape", the program will display the number of 2560-character blocks (20 128-character records) read.

2.4 multics_to_remote, mtor

Function: mtor (device) (-debug)

Parameters:

device	name of attached remote device
-debug, -db	test switch

Description:

The user function mtor transfers an ASCII data file from a MULTICS file in the users' default working directory to a remote device.

The test switch "-debug" or "-db" is used by maintenance personnel to trace the program execution and data values. It should not be specified by the normal user.

User Interaction:

If the remote device is incorrectly specified on the call, the following error message is displayed:

(1) Remote device must be "4921", "4922", "4923", or "tape".

If the remote device is incorrect or missing, the following request will be displayed:

(2) Enter remote device type: xxxxLF

Type in the remote device type followed by a line-feed (LF).

(3) Enter input data file name: nameLF

Type in the desired input file name followed by a LF.

If the data file does not exist, the next two responses are displayed:

(4) Data file does not exist.

(5) Do you wish to try another file? replyLF

Type in "yes" or "no", followed by a LF. A reply of "no" terminates the program. A reply of "yes" displays the following message before returning to question (3).

Choose another file name.

If the remote device is "4922", the following question is asked:

(6) Enter disk number: nLF

Type a "1" or a "2", followed by a LF.

(7) Do you want the screen on during data transmission? replyLF

Type "yes" or "no", followed by a LF. The user should be aware that the screen can display only 8448 characters (64 lines times 132 chars per line) before overwriting the displayed information. This represents approximately 66 sectors, or 2 tracks of data.

If the remote device is "4921" or "4922", questions (8) and (9) are displayed.

(8) Enter starting track number: nnLF

Type a two-digit number between 00 and 63, followed by a LF.

(9) Enter starting sector number: nnLF

Type a two-digit number between 00 and 31, followed by a LF.

If the remote device is "4923", question (10) is displayed.

(10) Do you wish instructions? replyLF

Type "yes" or "no", followed by a LF.

After the instructions are displayed, the user is asked if he is ready to continue. Type a "yes", followed by a LF. The program will then display the following message, and wait for 20 seconds for the user to comply before initiating data transfer:

Depress WRITE and RUN buttons.

The user must follow the instruction within 20 seconds, or the device will not be able to accept the data transmitted by the program. In such case, the program should be allowed to terminate normally, then re-executed.

If the remote device is "tape", questions (11) through (17) will be asked.

(11) Enter volume id: VVVVVVLF

Type 6-digit tape reel number with leading zeroes, followed by a LF.

(12) Enter optional console message: messageLF

Type LF for no message, or up to 64 characters (including blanks) followed by a LF. A message greater than 64 characters will be truncated.

(13) Enter file name: nameLF

Type up to 17 characters followed by a LF. A name greater than 17 characters will be truncated.

At this point, all required information has been entered. If the user wishes to correct misspelled fields, enter new fields, or delete any fields in the attach description, the program will permit editing. The following question is asked automatically:

(14) Do you wish to change any fields? replyLF

Type "yes" followed by a LF if you wish to edit any of the fields in the attach description.

If the user types "yes" in response to question (14), the program continues with questions (15) through (17).

(15) Enter literal to be changed: literalLF

Type in any string to be changed, followed by a LF (the

literal string may contain blanks).

(16) Enter literal to be changed to: literalLF

Type in replacement literal (a null string or a literal with imbedded blanks is permissible), followed by a LF.

The edited attach description is now displayed.

(17) Do you wish too continue? replyLF

Type "yes" followed by a LF to continue. The program will then display questions (15) through (17) again. Type "no" to terminate editing.

After completion of the data transfer, the program will display the number of 128-character records written. If the remote device was "tape", the program will display the number of 2560-character blocks (20 128-character records) written.

2.5 maketree

Function: maketree (-control_arguments)

Control arguments:

-in path	input ASCII data file path name
-out path	output MOOS tree name
-debug, -db	test switch

Functional Description:

This program reads the named ASCII input data file from the user's default working directory, writes an intermediate data file (treedata) in the user's process directory, and then calls the program treeiput to convert the intermediate file to MOOS format. See treeiput for a description of the treedata file format.

The test switch "-debug" or "-db" is used by maintenance personnel to trace the program execution and data values. It should not be specified by the normal user.

User Interaction:

If the input ASCII data file is not specified in the call to maketree, the following question is asked:

(1) Enter default working directory input ASCII data file name: nameLF

Enter the input file name, followed by a LF.

If the program cannot locate the input file, the following question is asked:

(2) Input ASCII file <name> does not exist. Do you wish to try another input file? replyLF

A reply of "yes", followed by a LF will cause the program to branch to question (1). A reply of "no" terminates the program.

If the output MOOS data file is not specified in the call to maketree, the following question is asked:

(3) Enter 5 to 8_character output MOOS data file name: nameLF

Type the file name, followed by a LF. A name less than 5 characters or greater than 8 characters will cause the program to return to question (3). The MOOS data file is checked for

existence in the program treeiput. Therefore, it is not checked by maketree.

(4) Enter number of keywords in input data file <name>. numberLF

Enter the number of keywords to skip, followed by a LF.

(5) Are class id's present? replyLF

Type "yes" or "no", followed by a LF. A "yes" will cause the program to use them in the file conversion. A "no" will cause the program to generate them for the file conversion.

The accompanying Figure 4 shows the relationship between the input ASCII data file, the intermediate data file (treedata), and the resultant MOOS tree.

As described earlier, the program maketree calls the program treeiput to convert the intermediate file to the MOOS tree. It is never necessary for the user to call the program treeiput.

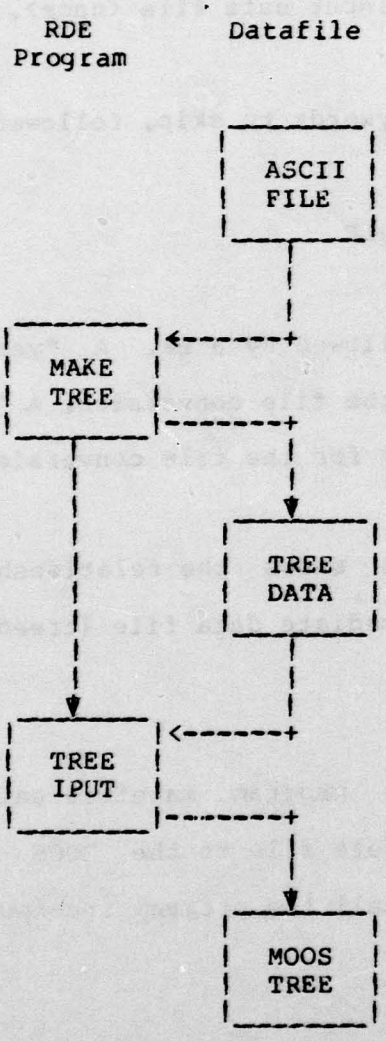


Figure 4 ASCII File to MOOS Tree Conversion

2.6 dumptree

Function: dumptree (-control_arguments)

Control arguments:

-in path — input MOOS tree path name

-out path — output ASCII data file path name

-debug, -db — test switch

Functional Description:

This program calls the program treeoput to convert the input MOOS tree in the user's process directory to an intermediate file (treedata) in the user's process directory, then processes the intermediate file to create an output ASCII data file in the user's default working directory.

The test switch "-debug" or "-db" is used by maintenance personnel to trace the program execution and data values. It should not be specified by the normal user.

User Interaction:

If the input MOOS data file is not specified in the call to dumptree, the following question is asked:

(1) Enter 5 to 8 character input MOOS data file name: nameLF

Enter the input file name, followed by a LF. A name of less than 5 characters or greater than 8 characters will cause the program to return to question (1). The existence of the input MOOS data file is checked by the program treeoput. Therefore, it is not checked by dumptree.

If the output ASCII data file is not specified in the call to dumptree, the following question is asked:

(2) Enter output ASCII data file name: nameLF

Enter the output file name, followed by a LF.

If the output file exists, the program will ask the following question:

(3) Do you wish to overwrite existing file <name>? replyLF

If the reply is "yes", processing continues. If the reply is "no", the program asks the following question:

(4) Do you want to try another file? replyLF

If the reply is "yes", the program branches to question (2). If the reply is "no", the program terminates.

If the treedata file cannot be found in the user's process directory after executing the program treeiput, the following message is displayed before the program terminates:

(5) "treedata" file does not exist in process directory - program terminated.

Figure 5 shows the relationship between the input MOOS tree, the intermediate data file (treedata), and the resultant ASCII data file.

As mentioned before, the program dumptree first calls the program treeoput to convert the MOOS tree to the intermediate data file. Then the program dumptree converts the intermediate data file to an ASCII file. The program treeoput is displayed first in Figure 5 only because this program is the first to do a file conversion. In actual execution, the user calls dumptree only, and need never call the program treeoput.

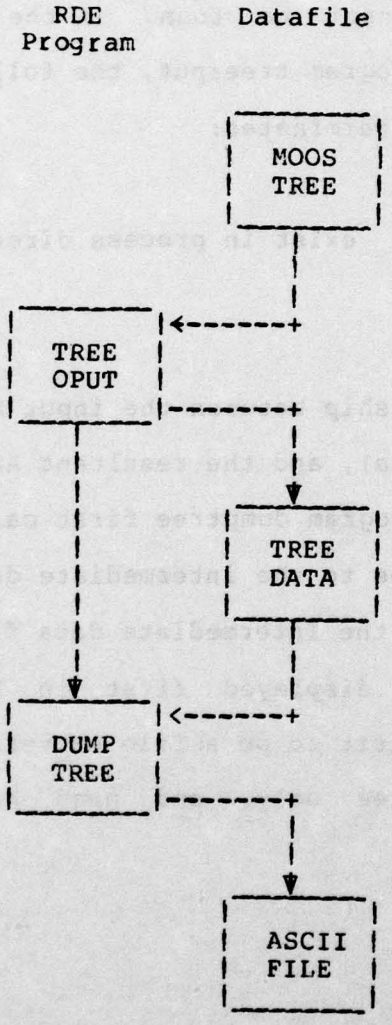


Figure 5 MOOS Tree To ASCII File Conversion

2.7 unpack_tree, upt

Calling Sequence:

```
unpack_tree treename
```

Arguments:

treename is the name of a WAVES tree to be unpacked.

Function Description.

Generates a WAVES tree from the ASCII file treename.rde located in the user's default working directory.

If a format or data error occurs while executing unpack_tree, the operation is halted. An error will be reported to the user. If any waveforms were successfully unpacked (up to the last semicolon passed), then the tree will be found in the user's file and will be selected (put on the data set stack).

The format of a WAVES RDE ASCII file is given in figure 5A. For more information on the use of unpack-tree and pack-tree, the user is referred to the MULTICS Long Waveform Analysis System User's Manual, (RADC-TR-78-218), A062111.

File Items

```
nodename,           /* from 1 to 27 characters, no periods */
id,                 /* from 1 to 10 characters, representing valid
                       positive integer */
tf,                 /* domain: "time" or "freq" */
units,              /* display units: "s", "ms", "us", "ns", "Hz", "kHz",
                       "MHz", "GHz" */
st_sf,              /* start time in seconds or frequency in hertz */
sr_fr,              /* sampling interval in seconds or hertz */
type,               /* waveform type: "real" or "complex" */
text,               /* waveform text */
tree_text,          /* tree text: assigned to the tree only if text
                              is non-null */
nseg_marks,         /* number of segmentation marks */
mark(1),            /* segmentation marks  word  coordinates marked */
  .
  .
  .
mark (nseg_marks),
X(1),               /* First waveform point */
  .
  .
  .
X(N);               /* semicolon marks end-of-waveform */
nodename,           /* repeat of above for each waveform */
  .
  .
  .
X(M);
/*                   /* "slash-star" marks end-of-tree */
```

Figure 5A WAVES I/O File Format

2.8 pack_tree, pt

Calling Sequence:

pack_tree (tree)

Arguments:

tree is the name of a WAVES data tree. (optional)

Function Description.

Converts a WAVES data tree into an ASCII formatted file whose format is specified in Figure 5A.

Detailed Description:

If a tree is not specified, the current tree is used.

This command does not affect the data set stack.

3 CTS USER'S MANUAL

3.1 Overview

The editing and extraction of data is achieved through computer programs which do the following: 1) scan data for specific keywords, 2) scan data for specific fields, and 3) evaluate boolean expressions which are functions of the data measurements.

A syntax (grammar) has been designed as a format by which the user should structure his commands. These commands must be recognized as being in the correct format before the editing specified by the commands is actually performed. The editing is accomplished through recognition of the commands, input by the user. There are four editing commands: "change", "move", "insert" and "delete". Briefly, the "change" command will change keywords or fields within a vector, the "move" command will move a field to either the head or tail of the vector, the "insert" command will insert a field anywhere in the vector, and the "delete" command will delete any field within a vector. A more detailed description of each command will follow later in this section. Once a command is recognized, it is translated into a PL/1 statement which is placed into a PL/1 program. This program can be compiled and executed. Provisions have been made controlling both the compilation and the execution of the program, which will also be discussed later in this section.

If the user mistypes a command, CTS will attempt to continue

processing the command, making corrections as it encounters the errors. When an error is detected, depending on the type of error, CTS will either make the correction itself or query the user for the correction. The corrected command is displayed to the user, who is then asked if the corrected command is acceptable. The user should reply "yes" or "no". If the reply is "yes", CTS will continue processing the corrected command from the point of the error. If the user replies "no", then CTS will respond with the question "Enter either ""cease"" or ""quit"":". Only at this stage in CTS can the user use the terms "cease" which terminates processing of all commands, or "quit" which terminates processing of the current command.

The original data file, located in the user's default working directory, is read, one vector at a time, into a buffer within CTS. Any editing (where editing consists of changing, moving, inserting and deleting data, keywords or fields within a vector) is then done on each vector individually. After each vector is edited, it is written into a file located in the process working directory. After the last vector of the original data file has been edited and written into the file, the original data file is then replaced by the edited data file (i.e. the original data file is written over by the edited data file).

The extraction of vectors, if any, occurs after the original data file has been replaced by the edited data file. Extractions are accomplished through recognition of the "extract" command as a CTS command. This command extracts pertinent vectors from the edited data file. Once recognized, the command will be translated into PL/1

statements which will then be appended to the same PL/1 program containing the editing commands. This program will not only edit the original data file, but also extract pertinent vectors and place them into a separate, user-named file located in the user's working directory for subsequent input to MOOS.

In processing each command, the PL/1 program will operate as a two-pass, multi-operational system. In the first pass, each "edit" command (where an "edit" command is any of the four commands listed above) will examine each individual vector to determine if the command can be applied to it. If it can, the necessary editing in done. After each edit command has examined a vector, the next vector is examined. This process continues until all the vectors have been examined. The second pass will then do any extractions by examining each vector and extracting the pertinent ones.

CTS will operate in one of two modes. It will operate interactively with the user as he is sitting at a terminal keyboard, or as in batch processing, with all commands read from a file with the editing done according to the given commands. The user must specify, when calling CTS, which mode of operation he wishes to use. More information concerning the two modes will be discussed later in this section.

CTS was designed as a fairly machine-independent system. There are, however, a few Multics system subroutines included in CTS. They are:

1) com_err_ prints the actual system error message instead of just the error number.

2) cu_$arg_count returns the number of arguments a subroutine is called with.

3) cu_$arg_ptr a pointer to the arguments of a command.

4) cu_$cp command processor which passes commands from within a program to the command level. It is used for the compilation and execution of the user's program.

5) get_default_wdir_ gets the default working directory.

6) get_pdir_ gets the process directory.

7) hcs_$delentry_file deletes a file in a specified directory.

8) hcs_$star returns the entry count of a segment or directory.

6) ioa_ used for formatting a character string from fixed_point numbers, floating-point numbers, character strings, bit strings, and pointers.

7)	iox_$attach	attaches the i/o switch "error-output".
7)	iox_$close	closes the i/o switch.
7)	iox_$detach_iocb	detaches the i/o switch "error-output".
9)	iox_$find_iocb	gets a pointer to the i/o switch "error-output".
10)	iox_$open	opens the i/o switch.

3.2 Definitions

The important terms used within CTS are defined here.

1) command - a statement which tells CTS what type of editing to do on the data file.

2) control arguments - arguments which specify the function of CTS - i.e. if the user wants a listing of the generated PL/1 program, or just a syntax check of the commands, etc.

3) data - a measurement of some attribute of an item under analysis. If a vector ID is given, it precedes the data. Otherwise the data immediately follows the keywords.

4) default working directory - the working directory of a user when he first logs on to the system.

5) degree of severity - Compilation of a program produces a list of errors, if any exist. Each error has a level of severity; which range from 0 to 4 as follows:

0 - indicates no errors were detected.

1 - indicates a minor error.

Compilation continues with no ill effect.

2 - indicates a correctable problem error. The compiler remedies the and usually continues with no ill effect.

3 - indicates a fatal error has been detected.

4 - indicates an unrecoverable error has been detected. The compiler cannot continue beyond this point.

A program having a degree of severity of 2 or less can be compiled. However, if the degree of severity is greater than 2, compilation of the program is terminated.

6) delimiters - the following delimiters are recognized in CTS:

comma (,) - separates fields within the vector.

dash (-) - indicates a CTS pathname or control argument immediately follows.

period (.) - indicates the end of an edit or extract command.

semi-colon (;) - denotes the end of a vector.

slash-asterisk (/*) - denotes the end of the data file.

7) field - a data vector in the data file has the format

classname, keyword1,keyword2,...,vector ID,data,...;

Each char string between commas is called a field. In this example, field(3) is keyword2.

8) keyword - a particular word or character string which the user has designated as important. Keywords follow the classname and precede the vector ID (if one is given) and any data.

9) parse - resolving a CTS command into its grammatical parts.

10) path - name of a file.

11) pathname arguments - arguments which specify an input, output or saved file.

12) process working directory - a directory containing those segments that are meaningful only during the life of a group of programs in execution.

13) production - rules of a grammar.

14) recursive - procedures are recursive if they can call themselves either directly or indirectly through a chain of other procedure

calls.

15) subfield - a substring within a field.

16) token - a word or character of the input command line.

17) top down processing - a processor which recognizes syntax productions higher up in the grammatical tree before those lower down.

3.3 CTS Initiation

Function:

Accepts commands to generate a PL/1 program which processes an ASCII RDE data file.

Syntax:

cts {-pathname_args} {-control_args}

Pathname Arguments:

-old path

path is the file name of the input segment or multi_segment ASCII RDE data file. If not present, the program will query the user for the input data file segment name.

-new path

file name of ASCII RDE data file to be created from applying cts to the input data file. Default path name will be "empty".

-in path

name of the segment from which CTS control arguments and commands are to be taken. This segment must have ".cts" as

a suffix, but is not required in the command line. May be used only as a calling argument.

-out path

name of the segment to which CTS control arguments and commands are be copied. If missing, a suffix of ".cts" will be applied by the program. This argument is incompatible with the -in argument above. The default is no output control segment.

-save path, -sv path

causes the generated PL/1 program to be saved in the current working directory. A suffix of ".pl1" is appended to the segment but is not required in the command line.

Control Arguments:

-list

causes a listing of the generated PL/1 program. Default is no listing.

-check, -ck

checks syntax of input. No translation or program generation.

-noxqt, -nx

Used with the -save argument to generate, but not execute the program.

-arguments argstring, -args argstring

passes the PL/1 compilation arguments "argstring" to the PL/1 compiler. If this argument is used, it must be the last control argument. The format for "argstring" is "-arg1 -arg2 ...-argn". The severity level argument, if specified, is ignored. Default is no compilation arguments.

-noquery, -nq

inhibits message soliciting user for control arguments, PL/1 arguments, and commands. Implies that all arguments and commands will be read from a file. Default is to query the user for all arguments and commands.

-echo

causes the input control arguments and commands to be echoed back to the terminal as they are read in. Default is no echo.

-debug, -db

causes a trace of statements executed. Default is no trace.

-menu

prints a listing of all the arguments available in CTS.

-panic

will notify the user if he attempts to put a vector in two different classes and will terminate execution of the generated PL/1 program. Default is notification of this occurrence, creation of a file containing all vectors placed in more than one class, and continued execution of the generated PL/1 program.

Note:

Control arguments may be specified at the time of the call to cts or as input commands from either the terminal or the file specified by the "-in" arguments. When both modes are used, arguments specified at calling time take precedence over input arguments.

User interaction:

If the input data file name is not specified when CTS is called, the program displays the following questions:

1) Enter missing input (-old) data file name:

Type in segment name of the data file followed by a linefeed(lf).

2) Enter control arguments:

Type in control argument followed by a linefeed(lf).

The user will continue being prompted for control arguments until he either types only a linefeed(lf) or types in the argument "-args" at which point question (3) is displayed.

3) Enter target program PL/1 argument:

Type in compilation arguments which will be used in compiling the generated PL/1 program. The user will continue being prompted for individual compilation arguments until he types in only a linefeed(lf).

Question (4) is displayed if after compiling the generated PL/1 program, an error of severity >2 has been found.

4) Do you want to execute the generated PL/1 program?

Type in "yes" or "no" followed by a linefeed(lf). If no compilation errors were found, or an error of severity <2 was found, then the

generated PL/1 program is automatically executed, unless the user has specified the argument "-noxqt".

Following are examples of how CTS may be called, and the results of the call.

1) cts -old vehicles -new cars

If the optional control argument "-out" was specified with the pathname "auto", the commands entered from the terminal would also be saved in the file "auto.cts".

Once the commands to CTS have been saved in a file, the user may choose to then use them instead of entering them at the terminal.

For example, if the commands to extract vectors from a file were in the entry "auto.cts", the following call to CTS would function as in example (1), but the user would not have to list the commands:

2) cts -old vehicles -new cars -in auto

Statements read from the input file may be echoed at the terminal (i.e. displayed) by including the argument "-echo", as shown in example (3):

3) cts -old -new cars -in auto -echo

Each read by CTS is preceded by a query to the user. This query may be disabled by including the argument "-noquery" or "-nq" as shown in example (4):

```
4) cts -old vehicles -new cars -in auto -echo -noquery
```

If the user simply wants to check the syntax of his commands, he should include the argument "-check" or "-ck". No translation of the commands or program generation is performed.

If the user expects to use the same program with various input data sets, he may elect to save the program rather than delete it after the execution. This is accomplished by including the argument "-save" or "-sv", and the name the program is to be saved under. Example (5) would cause the generated program to be saved in the file "extract_cars.pl1":

```
5) cts -old vehicles -new cars -in auto -echo -sv extract_cars
```

Upon completion of CTS, the user may then use the MULTICS command "pr" to print a copy of the saved source segment.

If the user elects not to execute the generated program, he should include the argument "-noxqt" or "-nx", as shown in example (6):

6) cts -old vehicles -new cars -in auto -echo -sv extract_cars -nx

The user may then elect to submit the saved program as part of an absentee job by using the MULTICS command "ear".

Should the user decide to see the generated program listed, he should include the argument "-list", as shown in example (7);

7) cts -old vehicles -new cars -in auto -echo -sv extract_cars -list

Should the user wish to see the flow of the program, he should include the argument "-debug" or "-db" as illustrated in example (8):

8) cts -old vehicles -new cars -in auto -echo -list -db

It is not recommended to use the argument "-debug" or "-db" since it was included specifically as a debugging and maintenance tool. However, the option is there for the user.

If the user wants a table or a map of the compiled program [see the MULTICS Programmers' Manual - Commands and Active Functions - "PL/1" function for explanations on how to compile a program with a map and/or table option], he should include the argument "-arguments argstring" or "-args argstring" where "argstring" is the string of arguments used for compilation purposes. These arguments will be passed to the PL/1 compiler with the program the user wants executed. If this argument "-args argstring" is used, it must be the last

control argument as shown in example (9):

9) cts -old vehicles -new cars -in auto -list -args -map -sv3

If the "-arg" argument is not specified, the program will be compiled with only the argument "-sv4".

One of the convient features of CTS is the way in which commands in the "-in" file are overridden by the calling arguments. Any argument which is set by the call may not be reset by "-in" file commands. Thus, the call may specify a different "-save" file name than what is in the "-in" file. Additionally, the call may set switches not specified in the "-in" file. Thus, if the "-in" file does not specify "-nx", but the call does, the program will not be executed.

3.4 Keyword Specification

Keywords are words within a vector which help distinguish one vector from another. If the vector data contains keywords, it is helpful in the editing and extraction process to specify and use these keywords to identify the correct vector. The purpose of process_keywords is to obtain these keywords and store them in an array for later reference. Subfields can also be specified, but are optional. Subfields are substrings of characters within the keywords. If a user would rather not have to refer to the whole keyword, but only a few characters, he can specify this through the use of subfields. If subfields are given, they should be in the form "a,b" where "a" and "b" are integers. The first subfield, "a", represents the beginning position within a field where the substring should begin. This field has been defined by the keyword denoted by the user. The second subfield, "b", represents the length or the number of characters to be considered. If "b" is not given, then beginning with "a", the rest of the string is extracted. If no subfields are given, the whole field is extracted.

Upon entering process_keywords, the the following questions are displayed to the user: 1) How many keywords?

Type in an integer representing the number of keywords in each vector in the data file.

2) Enter keyword and optional subfields:

User types in the keyword. If there are subfields, the user can follow the keyword by a space and then the subfields. If the user wants to skip naming the keywords, he types in "skip" followed by a linefeed (lf). Question (4) will be displayed prompting the user for subfields for each keyword. If he mistakenly hits only a linefeed(lf), question (3) will be displayed.

3) Please enter a keyword or "skip":

User types in either a keyword or "skip" followed by a linefeed(lf).

4) Enter keyword(n) subfields:

where n runs from the current keyword number to the total number of keywords. User types in the subfields followed by a linefeed(lf). For examples of keyword subfields, refer to the program cts_process_keywords documentation.

If a keyword subfield is not an integer or not in the correct format, then question(5) will be displayed.

5) Enter keyword subfield(s):

User types in keyword subfield or subfields for the current keyword.

To illustrate the use of subfields, let the vector

blue-purpel,oval,brown,5.6,.5,1.0;

be the one the user is interested in correcting. Note the incorrect spelling of the first keyword, "blue-purpel". The user wants to change the last two letters from "el" to "le", but would rather not type in the complete keyword. He is able to do this by specifying, when he is asked for the first keyword and subfields, the following:

color 10,2

When CTS examines the first keyword, it will skip to the tenth character and begin its examination from that point for two characters, which in this case is the end of the word. If the user had typed in the command:

change color 10,2 ="el" to "le".

the first keyword of the vector would now look like this:

blue-purple,oval,brown,5.6,.5,1.0;

If the user is more concerned with the location of the keyword in the vector rather than the actual keyword, he can type in "skip" when he is asked to enter a keyword and optional subfields. Once this is done, the remaining keywords will be generated as keyword(i) where i is $1 \leq i \leq n$ and n is the number of keywords. For each generated keyword, the user is asked for optional subfields. If the user is not concerned with all or some of the subfields, he should type "nsubs". This will indicate to process_keywords that there are no subfields for the remaining keywords that are to be generated.

For example:

The user is asked for the number of keywords and he types in "3". He is then asked for the first keyword along with its subfield. He types in "plane 2,3". He is asked for the second keyword. He types "skip nsubs". He is specifying that there are no subfields for the generated keywords. So the keywords that will be recognized are: 1) plane 2,3 2) keyword(2) and 3) keyword(3) where keyword(2) and keyword(3) refer to the second and third keyword in a vector.

3.5 CTS Command Syntax

CTS generates a PL/1 program based on the user-supplied CTS commands which will operated on the data file. During the process of generating the Pl/1 program, CTS obtains edit and extraction commands and passes these commands to the corresponding subroutines for further processing.

User interaction:

1) Do the vectors in the data file have a classname located as the first point of the vector?

Type in either "yes" or "no" followed by a linefeed(lf).

2) Is there a vector ID following the keywords?

Type in either "yes" or "no" followed by a linefeed(lf).

If the user fails to answer yes or no to questions (1) and (2), the following question is displayed.

3) Please answer "yes" or "no":

Type in either "yes" or "no" followed by a linefeed(lf).

4) Enter command line:

Type in a command(s) followed by a linefeed(lf). For the format of commands, see the syntax charts. Once the syntax of the command line has been checked, and if the line does not end in "end", then the subroutine "get_char" will ask the user for the next command line. If there is an uneven number of quotes in the line, an error subroutine "error" notifies the user of the erroneous command line.

There are five commands that will be recognized. They are: "change", "extract", "move", "delete" and "insert". A syntax, as mentioned earlier, has been designed as a guide to the user in formating his commands. It is a context-free LL(1) grammar having recursive procedures and top down processing. The grammar is presented in the following figure and demonstrates the manner in which CTS will parse each command. Any term enclosed within "<...>" is a nonterminal while all other terms are terminals. A nonterminal is parsed until a terminal is encountered, indicating the end term (a terminal) which is acceptable to CTS. The terminals "and" and "or" are represented as "&" and "|" respectively in the grammar.

The following figures represent the syntax graphs of each production of the grammar. In the syntax graphs, all nonterminals are enclosed within "<...>" and all terminals are enclosed within "(...)". Optional terminals are enclosed within "{ }". In both the

grammar and the syntax graphs, the term "null" indicates a nullable production meaning that the production may not be needed in every command or series of commands.

NOTE: It is important that the user enter all the "edit" commands before any "extract" commands are entered. If not, CTS will terminate. Each command must end in a period. If it does not, the user will be told of the error, a period appended to the end of the command, and CTS will continue to the next command. After the user has entered all his commands, he should enter the word "end" indicating to CTS that all the commands have been entered.

```
<syntax> ::=                    <command>end
<command> ::=                   <change> <move> <insert> <delete> <extract>
                                |<move> <insert> <delete> <extract>
<change> ::=                    change <change expression>.
<extract> ::=                   extract <Boolean term> <change_class>.
                                |null
<move> ::=                      move <field_loc>=<string> to <loc>.
                                |null
<insert> ::=                    insert <field_loc>=<string> <string>.
                                |null
<delete> ::=                    delete <field_loc>=<string>.
                                |null
<change expression> ::=         <relation> to <string>
<Boolean term> ::=              <Boolean factor> <or_term>
<or_term> ::=                   | <Boolean term>
                                |null
<Boolean factor> ::=            <Boolean secondary> <and_fac>
<and_fac> ::=                   & <Boolean factor>
                                |null
<Boolean secondary> ::=         <Boolean primary>
                                |^<Boolean primary>
<Boolean primary> ::=           <relation>
                                |(<Boolean term>)
```

Figure 6 CTS Command Syntax

```
<change_class> ::=          class=<string>
<relation> ::=              <data> <relational operator> <value>
<data> ::=                  <predefined keyword> <position>
                           |<field> <position>
<relational operator> ::=  <|^>|=|^<|>|^=
<position> ::=              <integer><more integer>
                           |null
<more integer> ::=          ,<integer>
                           |null
<predefined keyword> ::=    keyword(<integer>)
                           |user-defined keyword
<field> ::=                 data
                           |field(<integer>)
<value> ::=                 <number>
                           |<string>
<number> ::=                any type of number
<field_loc> ::=             field(<integer>)
<string> ::=                "string of ASCII_chars"
                            {excluding control chars}
<loc> ::=                   head
                           |tail
<integer> ::=               any combination of the following
                            integers - 0|1|2|3|4|5|6|7|8|9
```

Figure 6 CTS Command Syntax (continued)

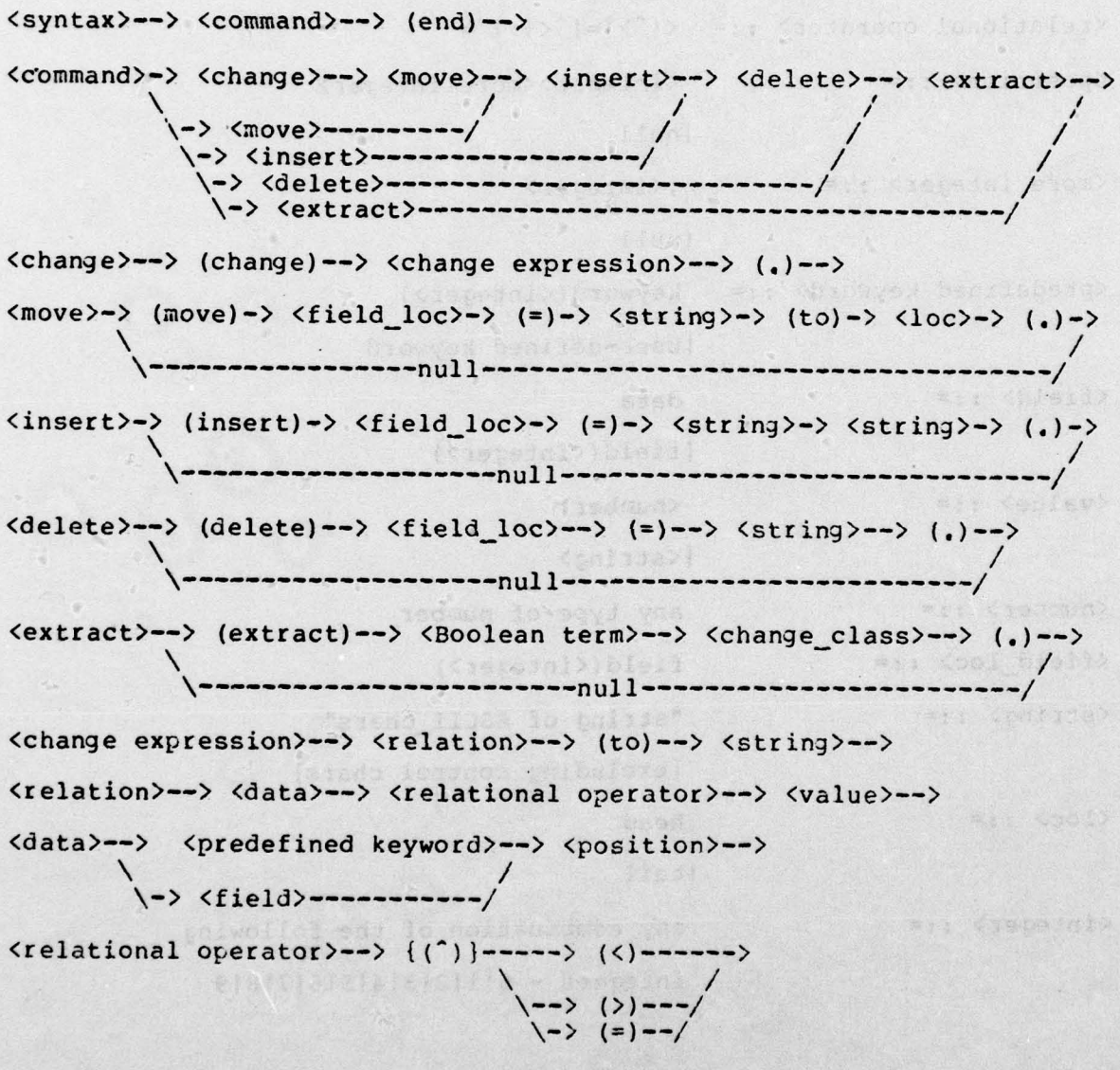


Figure 7 CTS Syntax Graph

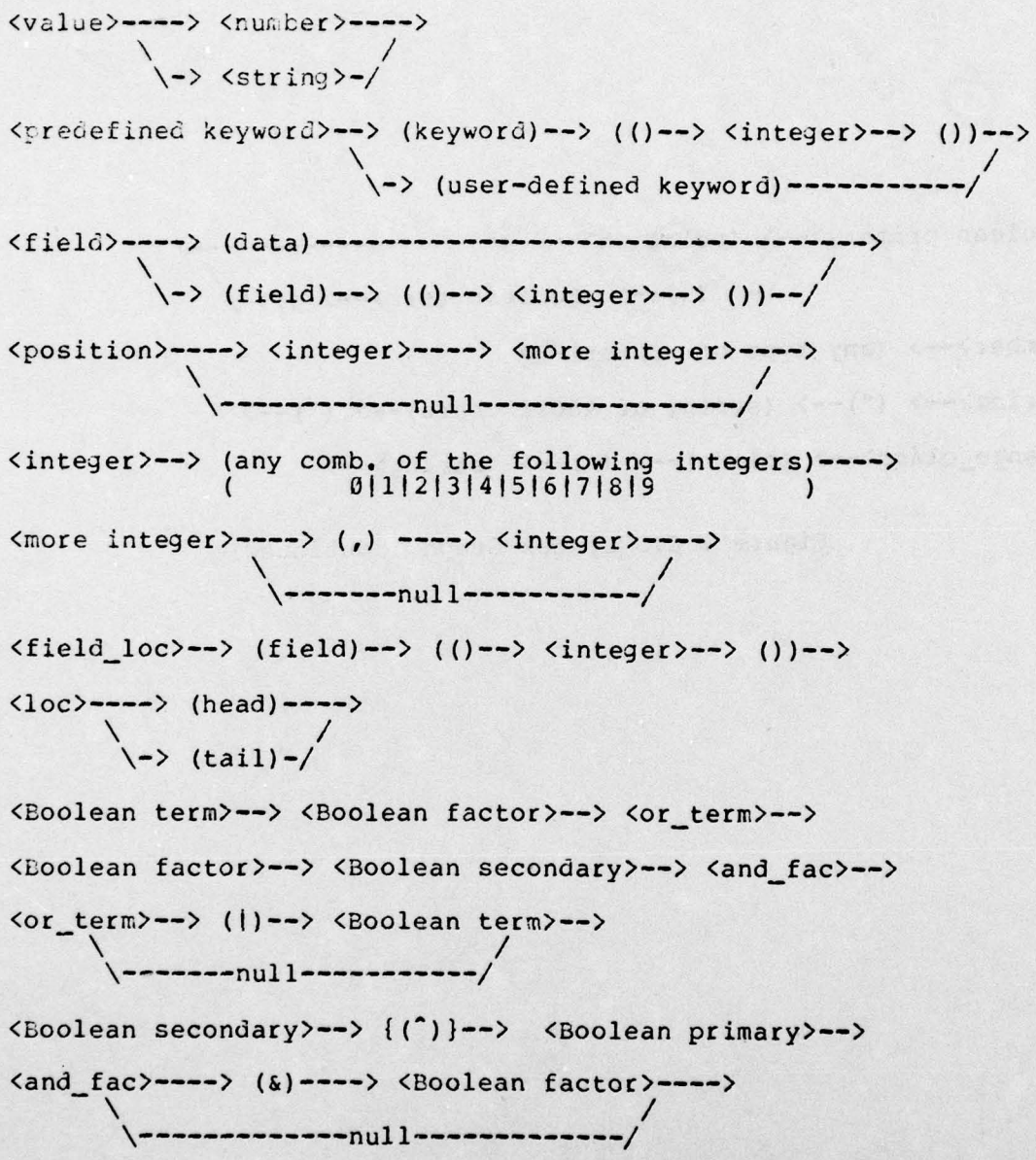


Figure 7 CTS Syntax Graph (continued)

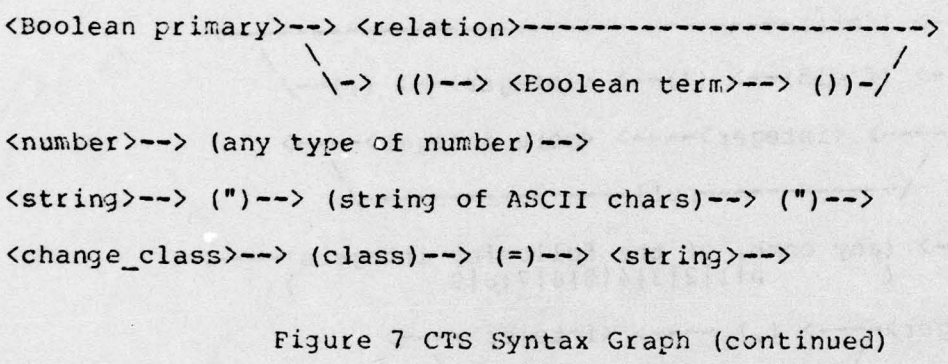


Figure 7 CTS Syntax Graph (continued)

3.6 Change Command

change <data> {<position>} <rel op> <value> to <string>.

where

1) <data> - consists of one of the four following terms:

   a) user-defined keyword - a keyword the user has specified earlier when asked for the keywords.

   b) keyword(i) - i is an integer and refers to the i-th keyword in the vector. The i should be enclosed in parentheses.

   c) data - refers to the data fields in the vector.

   d) field(i) - i is an integer and field refers to the i-th field in the vector The i should be enclosed in parentheses.

2) <position> - is an optional entry and refers to the location within a field plus the number of characters to consider. It is an optional entry since it may not be needed in every "change" command. The format of "position" is

"a,b" or "a" where "a" and "b" are integers. The position has the same meaning as the keyword subfields mentioned earlier.

3) <rel op> - consists of one of the following:

The relational operators "<" and ">" are to be used only with data or field(i) relations.

4) <value> - consists of either a string of ASCII characters enclosed within quotes or a number.

5) <string> - a string of ASCII characters enclosed within quotes.

Command Description:

The "change" command is used to change either keywords, fields, or data within a vector. All changes occur on the current version of the vector. An example follows using various "change" commands to demonstrate its use.

The given vector is:

jeaps,fourwheel,Ø125,4.236E-14,56.05,-32,-.5068E-05;

If the first command given were: change data=56.05 to "5.605", the resulting vector would look like this:

jeaps,fourwheel,Ø125,4.236E-14,5.605,-32,-.5068E-05;

Suppose the next command given was: change field(1) 3,1="a" to "e", the vector would look like this:

jeeps,fourwheel,Ø125,4.236E-14,5.605,-32,-.5068E-05;

The third command given is: change keyword(2) 1,4="four" to "",

the vector becomes this:

jeeps,wheel,Ø125,4.236E-14,5.605,-32,-.5068E-05;

3.7 Delete Command

delete field(i)=<string>.

where

1) field(i) - refers to the i-th field in the vector. The i should be in parentheses.

2) <string> - is a string of ASCII characters enclosed within double quotes. If the character within the quotes is "*" then all field(i)'s will be deleted.

Command Description:

The "delete" command is used to delete fields within the vector. This command states that the i-th field containing a certain string is to be deleted. As in the "change" and "insert" commands, all deletions of fields are relative to the current version of the vector.

For example:

Suppose the original vector is:

planes,silver,jet,twoseater,150,375,0.24E09;

and the first command given is: delete field(2)="silver". The resulting vector would be:

planes,jet,twoseater,150,375,0.24E09;

If the next command given were: delete field(4)="150". the vector becomes:

planes,jet,twoseater,375,0.24E09;

If the last command given was: delete field(2)="*". then all vectors would have field(2) deleted.

3.8 Insert Command

insert field(i)=<string> <string>.

where

1) field(i) - refers to the i-th field in the vector. The i should be enclosed in parentheses.

2) <string> - is a string of ASCII characters enclosed within double quotes. If the first string within quotes is "*" then all field(i)'s will have the second string inserted before them.

Command Description:

The "insert" command is used to insert fields in a vector. This command states that the second string of characters "string" is to be inserted before the i-th field containing a certain string so that the string originally located in the i-th field is relocated to the i+1 field and the i-th field will contain the new string. It is important to remember that all insertions are relative to the current version of the vector.

For example:

Suppose the original vector given is:

cat,stripes,grey,bobbed,32.5,2.97,15;

The command: insert field(3)="grey" "multitoed". would yield the following vector:

cat,stripes,multitoed,grey,bobbed,32.5,2.97,15;

If the next command given were: insert field(6)="32.5" "2". the following vector would be produced:

cat,stripes,multitoed,grey bobbed,2,32.5,2.97,15;

If the next command given were: insert field(2)="*" "blue-eyed". the following vector would be produced:

cat,blue-eyed,stripes,multitoed,grey,bobbed,2,32.5,2.97,15;

3.9 Move Command

move field(i)=<string> to <loc>.

where

1) field(i) - refers to the i-th field in the vector. The i should be enclosed in parentheses.
2) <string> - is a string of ASCII characters enclosed within double quotes. If the string within the quotes is "*" then all field(i)'s will be moved to the position specified by <loc>.

3) <loc> - consists of one of the following:

a) head - refers to the first field in the vector.
b) tail - refers to the last field in the vector.

Command Description:

The "move" command is used to move a field within a vector to either the head or tail of the vector. This command states that the i-th field containing a certain string is to be repositioned to either the head or the tail of a vector. As mentioned earlier in the "change", "insert" and "delete" commands, all moving of fields is relative to

the current version of the vector.

For example: Suppose the original given vector is:

tanks,.100E-05,2.68,62,beta;

and the first command given is: move field(5)="beta" to head. the resulting vector would look like this:

beta,tanks,.100E-05,2.68,62;

If the second command given were: move field(3)=".100E-05" to tail. the resulting vector would look like this:

beta,tanks,2.68,62,.100E-05;

If the third command given were: move field(2)="*" to head. the vector would look like this:

tanks,beta,2.68,62,.100E-05;

3.10 Extract Command

```
extract {^}(<data> {<position>} <rel op> <value>
        {& {^}(<data> {<position>} <rel op> <value>)}
        {| {^}(<data> <position>} <rel op> <value>)})
        class=<string>.
```

where

1) anything enclosed within braces is an optional string. Not all "extract" commands will be as complicated as the format illustrated above.

2) parentheses are optional except when an integer follows the word "field" or "keyword". In this instance, parentheses are required around the integer. If parentheses are used, they must be in pairs.

3) the symbol "^" is optional and means "not".

4) there is no limit to the occurrences of the strings

"& <data> <position> <rel op> <value>" and
"| <data> <position> <rel op> <value>".

5) the string "class=" is a required string. It refers to the class name given to the extracted vector.

6) <string> - is a string of ASCII characters enclosed within double quotes.

Command Description:

The "extract" command is used to extract certain vectors from the edited data file and relocate them into a user-named file for subsequent input to MOOS. The extracted vectors are placed into the file, located in the user's working directory, which was specified earlier in CTS with the "-new" argument. The extractions are based upon fulfillment of certain criteria found in the Boolean expressions in the "extract" commands. The extracted vectors are placed into classes specified in the "extract" command. In the following example all the extracted vectors have been categorized into the same class, "vehicles". There could, however, be many vectors extracted with many different classes created. Note in the syntax above, that the star convention used in the commands "move", "insert", and "delete" is not useable in the "extract" command. Also, "and" is represented as "&" and "or" is represented as "|" in both the syntax diagrams this command.

For example:

Suppose the edited data file consists of the following vectors:

```
jeeps,wheels,Ø125,4.236E-14,5.605;
cat,striped,multitoed,32.5,2.97;
planes,jet,twoseater,375,Ø.24EØ9;
beta,tanks,2.68,62,.1ØØE-Ø5;/*
```

If the first "extract" command given is:

```
extract keyword(1)="jeeps" | keyword(1)="planes" class="vehicle".
```

the extracted vectors based upon this command would be:

```
vehicle,jeeps,wheels,Ø125,4.236E-14,5.605;
vehicle,planes,jet,twoseater,375,Ø.24EØ9;
```

leaving these two vectors remaining in the edited data file:

```
cat,striped,multitoed,32.5,2.97;
beta,tanks,2.68,62,.1ØØE-Ø5;/*
```

If the next command given is:

```
extract field(2) 1,4="tank" & data>2.5 class="vehicle".
```

this vector would be extracted:

```
vehicle,beta,tanks,2.68,62,.100E-05;
```

leaving only one vector remaining in the edited data file:

```
cat,striped,multitoed,32.5,2.97;/*
```

It is important to note that the word "data" in the extract command does not have a substring position following it. Only the words "field(i)", "keyword(i)" and a user-defined keyword are allowed to use a substring position.

3.11 Command Summary

To summarize the commands, there are four "edit" commands and one "extract" command. The four "edit" commands are "change", "insert", "move", and "delete". The only "extract" command is "extract". An example using one of each command follows.

Suppose the original data file consists of the following vectors:

```
cats,grey,blue,4,4,20;
mules,brown,blue,12,4,50;
cycles,fourspeed,sixcylinder,25,120,12.25;
trainer,jet,white,115,250,.105E-06;
fighter,jet,green,375,115,.82E-02;/*
```

If the following commands were given to CTS:

```
change field(3) 7="inder" to "".
move field(2)="jet" to head.
insert field(3)="white" "strmline".
delete field(4)="white".
extract keyword(2)="jet" | data>75 class="vehicle".
```

the results would be as follows:

1) The first command "change" would examine each vector and edit any that it pertained to. The only one it pertains to is the third vector in the file. The resulting vector is:

cycles,fourspeed,sixcyl,25,120,12.25;

2) The next command "move" examines each vector and can operate on two vectors. The new vectors are:

jet,trainer,white,115,250,.105E-06;
jet,fighter,green,375,115,.825E-02;

3) The third command "insert" pertains to only one vector. The new vector is:

trainer,jet,strmline,white,115,250,.105E-06;

4) The fourth command "delete" pertains to only one vector creating the vector:

trainer,jet,strmline,115,250,.105E-06;

5) The fifth command "extract" can be applied to three vectors. The new vectors placed into a file for input to MOOs are:

```
vehicle,cycles,fourspeed,sixcyl,25,120,12.25;
vehicle,jet,trainer,strmline,115,250,.105E-06;
vehicle,jet,fighter,green,375,115,.82E-02;
```

In this command, any vector having either a keyword(2)="jet" or any data>75 will be extracted. The vectors remaining in the edited data file are:

```
cats,grey,blue,4,4,20;
mules,brown,blue,12,4,50;
```

3.12 Modes of Operation

It was mentioned earlier that CTS could be used interactively or in batch. In the interactive mode, the user must be at the terminal keyboard. In the batch mode, all data is read from a file.

As CTS scans for arguments in the interactive mode, the user will be asked for the arguments if none have been found in the call. He is then asked for the number of keywords . If the number is greater than zero, the user is asked to input each keyword and subfields of the keyword, if any. At this stage, CTS is ready for receiving the commands. If both editing and extraction commands are used, all editing commands must be input first. The user will be queried for a command, which will be input one line at a time, regardless of the length of the command. It is permissible to have more than one command per line, and a command which occupies more than one line. CTS will process each command and upon completion of processing a command line, the user is queried for a new command line. This cycle continues until CTS encounters the word "end" which indicates that all commands have been entered. If while processing a command, an error is detected, the user will be immediately notified of the error, told where the error occurs in the line, and either asked to correct it or told of its correction. Processing will then continue.

The user has two options in terminating commands. The first option is "quit" which means to terminate processing of the current command, skip to the next command, and begin processing the new

command. The second option is "cease" which indicates that processing of all commands and consequently, the routine process_commands, are to terminate.

As mentioned earlier, the only time the user may use "cease" or "quit" is when an error has been corrected in the command line and the user is queried as to the acceptability of the corrected command. If the user indicates that the corrected command is unacceptable, then he will be asked to enter either "cease" or "quit".

In the batch mode, as stated above, all data is read from a file. If while processing, an error is detected, the error and its location in the line are written into a file for the user to refer to at the termination of CTS. The processor will continue processing the commands until it encounters an uncorrectable error or it gets completely lost in parsing the commands, preventing it from continuing. If this occurs, the program will automatically terminate.

The format for the input argument file is shown in a few examples below. The various examples illustrate the different ways the argument file can be set up.

Example 1:

```
-noquery
-old data_file
-save newfile
-new newfilel
```

```
-args
-map
-table

3
color
shape
texture
no
no
extract keyword(1)="blue" class="same",
extract (keyword(1)="blue" & keyword(3)="hard") class="special",
end
```

Example 2:

```
-noquery
-noxqt
-check

3
skip nsubs
no
no
```

```
move field(3)="rat" to head. delete field(1)="rat".
insert field(2)="pie" "grasshopper".
end
```

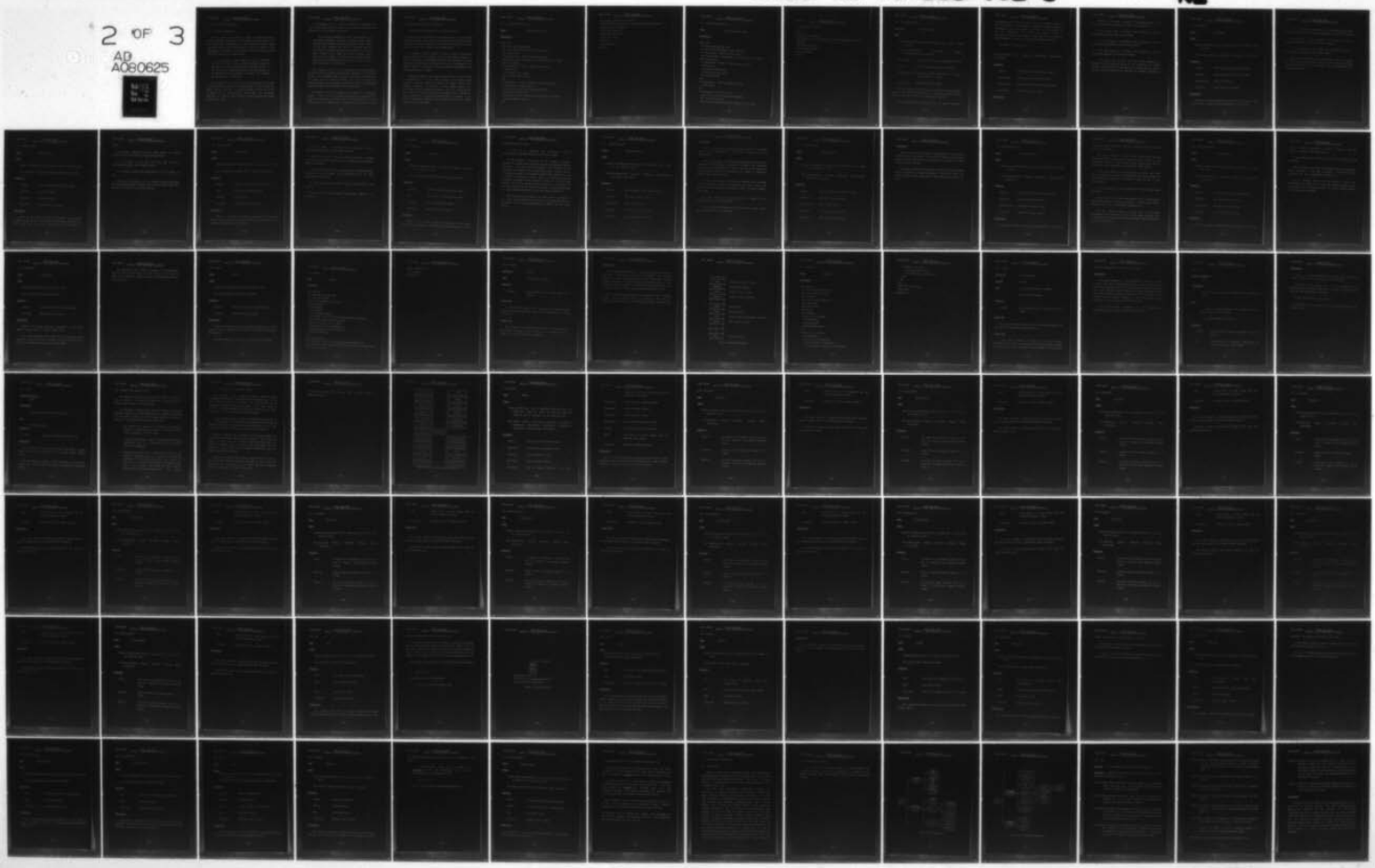
Example 3:

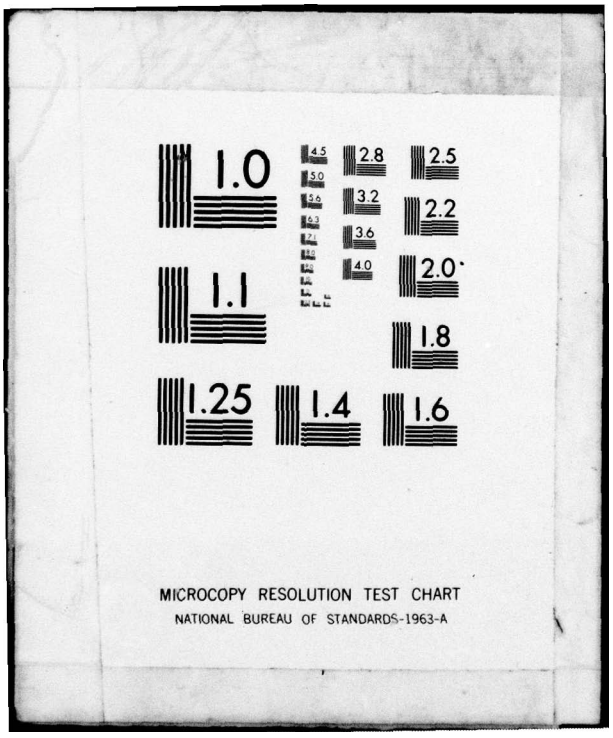
```
-noquery
-noxqt
-old data_file
-save nffile
-new nffile1
-panic
-args -map -table

3
skip
2,4
3,1
2
no
no
extract field(1)="blue" class="first".
extract data>6.5 class="second". end
```

AD-A080 625 PATTERN ANALYSIS AND RECOGNITION CORP ROME N Y F/G 9/2
MULTICS REMOTE DATA ENTRY SYSTEM. VOLUME I.(U)
OCT 79 D BIRNBAUM, J J CUPAK, J D DYAR F30602-77-C-0174
UNCLASSIFIED PAR-79-89 RADC-TR-79-265-VOL-1 NL

2 OF 3
AD
A080625




1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

4 RDE FUNCTIONAL DESCRIPTION

4.1 I/O Module Introduction

An I/O module was written in order to transmit the correct control characters required to turn the Tektronix 4014-1 screen on or off during data transmission, or start and stop a specific remote (peripheral) device under program control. This I/O module, called "remote_data_", generates an I/O switch which then controls the data transmitted through the terminal system.

> "An I/O switch is like a channel in that it controls the flow of data between program accessible storage and devices, files, etc. The switch must be attached before it can be used. The attachment specifies the source/target for I/O operations and the particular I/O module that performs the operations." [1, Section 5]

The attachment of the I/O switch is performed when the program "remote_to_multics (rtom)" or "multics_to_remote (mtor)" calls the MULTICS system subroutine "iox_$attach_ioname" with the name of the I/O module and its related remote device name. This attachment mechanism is described further under the remote_data_attach program documentation, below.

The attachment of an I/O switch generates an associated I/O control block (iocb) which is used by the I/O module to determine what operation is to be performed when called.

> "The principal components of an I/O control block are the pointer variables and entry variables whose values describe the attachment and opening of the I/O switch. There is one entry variable for each I/O operation with the exception of the attach operation, which does not have an entry variable since there can be only one attach entry point in the I/O module. To perform an I/O operation through the switch, the corresponding entry value in the control block is called." [4, Section 4]

The location of the iocb is returned as the pointer variable "iocb_ptr" by the iox_$attach_ioname system subroutine. This pointer is used in all subsequent calls to any iox_ subroutine to identify the I/O module and its internal entries. Thus, any call to an iox_ subroutine which uses the iocb_ptr variable will cause the appropriate related I/O module entry to perform the specified work.

For example, once the I/O switch has been opened for sequential output, a call by mtor to iox_$write_record with the iocb_ptr, will cause the remote_data_ I/O module entry "write_first_record" to be executed. This is a result of the I/O module iocb_ptr variable for the

write_record routine being specified as:

"iocb_ptr->iocb.write_record = write_first_record;"

All functions that are performed by specific entries in the I/O module are thus indicated in the iocb. The particular pointer values for each iocb entry is further described in the following documentation, and in greater detail in the program listing documentation.

The MULTICS control language has been utilized to the fullest possible extent in the development of the RDE system therefore a working knowledge of the MULTICS environment is essential for anyone considering modification of this system. The user is referred to the manuals listed in Appendix B for a more complete description of the MULTICS system and its usages.

Information regarding the system subroutines "iox_" may be found in references [3] and [4]. During testing, the MULTICS command "io_call" is used to verify the various I/O module entries. For further information on the use of this command consult [2]. Additionally, MULTICS system error codes of the form "error_table_" were used, and their description may be found in [1]. Finally, the user is referred to the Tektronix manuals listed in Appendix B for further information regarding the Tektronix devices and related required control characters.

4.2 remote_to_multics, rtom

Entry: remote_to_multics, rtom

Description:

```
begin rtom;
  get input remote_device name;
  get output ASCII multi-segment file (msf) name;
  open output file;
  if remote_device = "4922" then get disk number;
  attach input switch "data_input" to "remote_data_" I/O module;
  open input switch;
  if remote_device = "4921" or "4922" (disk) then do;
    instruct user;
    get starting track number;
    get starting sector number;
  end;
  if remote_device = "4923" (cassette) then instruct user;
  if remote_device = "tape" then do;
    query user for attach description parameters;
    edit attach description;
    attach output switch "tapefile" to "tape_ansi_" I/O module;
    open the tapefile switch;
  end;
```

```
  get pointer to start of msf buffer;
  read data from remote_device or tape through appropriate switch;
  append data to msf buffer;
  close input switch(s);
  detach input switch(s);
  set msf bit count;
  close msf;
  write statistics;
end rtom;
```

4.3 multics_to_remote, mtor

Entry: multics_to_remote, mtor

Description:

```
begin mtor;
  get output remote_device name;
  get input ASCII multi-segment file (msf) name;
  if remote_device = "4922" get disk number;
  attach output switch "data_output" to "remote_data_" I/O module;
  open output switch;
  if remote_device = "4921" or "4922" (disk) then do;
    instruct user;
    get starting track number;
    get starting sector number;
    position seek head;
  end;
  if remote_device = "4923" (cassette) then do;
    instruct user;
  end;
  if remote_device = "tape" then do;
    query user for attach description parameters;
    edit attach description;
    attach input switch "tapefile" to "tape_ansi_" I/O module;
```

```
    open "tapefile" switch;
  end;
  open msf;
  get pointer to msf buffer;
  write data from msf to remote_device or tape through appropriate switch;
  close msf;
  close output switch(s);
  detach output switch(s);
  write statistics;
end mtor;
```

4.4 remote_data_attach

Entry Name: remote_data_attach

Usage:

```
declare iox_$attach_ioname entry (char(*), ptr, char(*), fixed
    bin (35));

call iox_$attach_ioname (switch_name, iocb_ptr,
    attach_description, error_code);
```

switch_name "data_input" for rtom, "data_output" for mtor

iocb_ptr switch's control block pointer (Output)

attach_description "remote_data_ XXXX", where "XXXX" is "4921", "4922", "4923", or "tape"

error_code system status code (Output)

The attach entry is the most crucial operation of the I/O module. It must set up and initialize the iocb, and return a pointer to the iocb (the iocb_ptr) for use by all subsequent iox_ calls.

The user program rtom or mtor must call the system subroutine

iox_$attach_ioname with the value for the arguments as described above. The system will then attempt to locate the attach routene by concatenating the word "attach" to the name of the I/O module. Thus for the remote_data_ I/O module, the attach routine is "remote_data_attach". This attach routine must have a very specific entry declration in order to be located and attached correctly.

The format for this entry is:

```
module_nameattach: entry (iocb_ptr, option_array, error_switch,
error_code);
```

Arguments:

iocb_ptr	switch's control block pointer (Input)
option_array	device_type attach argument (Input)
error_switch	initiates verbose error messages (Input)
error_code	system status code (Output)

Description:

Called by the system subroutine "iox_$attach_ioname" or the system command "io_call attach" to attach the I/O switch.

If the device type is missing, the error_code is set to error_table_$noarg, and the entry returns.

If the iocb_ptr is unattached, the error_code is set to error_table_$not_attached, and the entry returns.

If the device_type is incorrect, the error_code is set to error_table_$bad_arg, and the entry returns.

If no errors are detected, then the attach description is generated, the iocb pointers are set to the appropriate I/O module entries, and the control is returned to the calling program. See the program listing documentation for specific and detailed information.

4.5 open_device

Entry: open_device

Usage:

```
declare iox_$open entry (ptr, fixed bin, bit (1) aligned, fixed
     bin (35));

call iox_$open (iocb_ptr, open_mode, extend_option, error_code);
```

Arguments:

iocb_ptr	switch's control block pointer (Input)
open_mode	sequential/direct input/output (Input)
extend_option	always "0"b (Input)
error_code	system status code (Output)

Description:

Called by the system subroutine "iox_$open" or the system command "io_call open" to open the switch designated by the iocb_ptr.

If the open mode is incompatible with the attached device type, the error_code is set to error_table_$bad_mode, and the entry returns.

If the switch is not attached, the error_code is set to error_table_$not_attached, and the module returns.

If the switch is not closed, the error_code is set to error_table_$not_closed, and the module returns.

Otherwise, the open description is generated, the iocb entries are set to additional I/O module entries, and the control is returned to the calling program. See the program listing documentation for detailed information.

4.6 terminal_modes

Entry: terminal_modes

Usage:

```
declare iox_$modes entry (ptr, char(*), char(*), fixed bin (35));

call iox_$modes (iocb_ptr, new_modes, old_modes, error_code);
```

Arguments:

iocb_ptr	switch's control block pointer (Input)
new_modes	mode string (Input)
old_modes	mode string (Output)
error_code	system status code (Output)

Description:

Called by the system subroutine "iox_$modes" or the system command "io_call modes" to set the modes for the switch designated by the iocb_ptr (see the set_tty active function for a discussion of

modes).

If the attach description has not been created, the module returns with an error code of error_tble_$not_attached.

If the switch is not open, the error code is set to error_table_$not_open, and the module returns.

If the external static switch "rde_debug" is on, no modes are set.

Otherwise, this module calls the system subroutine iox_$modes with the iox_$user_output iocb_ptr to set the user terminal modes (see set_tty), and returns to the calling program.

4.7 device_control

Entry: device_control

Usage:

```
declare iox_$control entry (ptr, char(*), ptr, fixed bin (35));

call iox_$control (iocb_ptr, order, info_ptr, error_code);
```

Arguments:

iocb_ptr	switch's control block pointer (Input)
order	device control command (Input)
info_ptr	unused (Input)
error_code	system status code (Output)

Description:

Called by the system subroutine "iox_$control" or the system command "io_call control" to perform a specified control order on the I/O switch designated by the iocb_ptr.

If the order is incorrect, the error_code is set to error_table_$no_operation, and the entry returns.

An order of "next_disk" starts the 4921/2 disk device, instructs the user to insert the next disk, calls iox_$seek_key (see disk_seek entry), and returns.

An order of "next_4923" stops the 4923 device, instructs the user to insert the next cassette, calls iox_$control with the order "start_4923", and returns.

An order of "on" sets the external static screen switch to 1 and returns.

An order of "off" sets the external static screen switch to 0, and returns.

4.8 disk_seek

Entry: disk_seek

Usage:

```
declare iox_$seek_key entry (ptr, char(256) varying, fixed bin
     (21), fixed bin (35));

call iox_$seek_key (iocb_ptr, key, rec_len, error_code);
```

Arguments:

iocb_ptr	switch's control block pointer (Input)
key	disk, track, and sector values (Input)
rec_len	set to 128 on return (Output)
error_code	system status code (Output)

Description:

Called by the system subroutine "iox_$seek_key" or the system command "io_call seek_key" to position the 4921/2 floppy disk to the

specified track and sector.

If the key is incorrect, the error_code is set to error_table_$improper_data_format, and the entry returns.

The seek command is given by the ESC (escape) & sequence, followed by an ASCII control character and three track-sector position characters. The ASCII control characters are selected from the characters 33 (decimal) to 92 (decimal) based upon 1) the disk number, 2) whether the screen is to display the data during transmission, and 3) whether the function is read or write. The track-sector position characters are formed from the numbers 00 through 63 for the track location, and an ASCII character from 65 (decimal) to 96 (decimal) for the sector as shown in [6]. Refer to the program listing documentation for a more detailed description of the seek command generation.

After positioning, the status of the remote device is checked. If the status code indicates that the disk is write_protected, then the error_code is set to error_table_$device_not_usable, and the entry returns.

4.9 read_first_record

Entry: read_first_record

Usage:

```
declare iox_$read_record entry (ptr, ptr, fixed bin (21), fixed
    bin (21), fixed bin (35));

call iox_$read_record (iocb_ptr, buffer_ptr, bytes_to_read,
    bytes_read, error_code);
```

Arguments:

iocb_ptr	switch's control block pointer (Input)
buffer_ptr	data buffer location (Input)
bytes_to_read	buffer length (Input)
bytes_read	how many were read (Output)
error_code	system status code (Output)

Description:

Called by the system subroutine "iox_$read_record" or the system command "io_call read_record" to read stream characters from the remote device.

This entry attaches a switch named "input_stream" with the attach description "record_stream_ user_input" as shown in [4]. Subsequent reads are through this switch in order to remove the CR-LF transmitted by the Tektronix device. An additional LF is removed by "flushing" the I/O buffers after the read.

If the remote device is the 4923 tape cassette drive, the ready mode is turned off, the user is requested to turn on the device, and the first read is delayed for 20 seconds to permit the user to prepare the device.

The iocb entry for read_first_record is changed to use read_next_record in subsequent calls.

For a more detailed explanation of the read record entry, refer to the program listing documentation.

4.10 read_next_record

Entry: read_next_record

Usage:

```
declare iox_$read_record entry (ptr, ptr, fixed bin (21), fixed
     bin (21), fixed bin (35));

call iox_$read_record (iocb_ptr, buffer_ptr, bytes_to_read,
     bytes_read, error_code);
```

Arguments:

iocb_ptr	switch's control block pointer (Input)
buffer_ptr	data buffer location (Input)
bytes_to_read	buffer length (Input)
bytes_read	how many were read (output)
error_code	system status code (Output)

Description:

Called by the system subroutine "iox_$read_record" or the system command "io_call read_record" to read the next (second and subsequent) record from the remote device by transmitting the appropriate command characters before reading.

Characters following the terminating symbols "/*" are effectively deleted by returning a reduced byte_count. A block of all stars ("*") signals the end of file, and causes the bytes_read count to be set to zero.

4.11 write_first_record

Entry write_first_record

Usage:

```
declare iox_$write_record entry (ptr, ptr, fixed bin (21), fixed
     bin (35));

call iox_$write_record (iocb_ptr, buffer_ptr, bytes_to_write,
     error_code);
```

Arguments:

iocb_ptr	switch's control block pointer (Input)
buffer_ptr	data buffer location (Input)
bytes_to_write	characters to output (Input)
error_code	system status code (Output)

Description:

Called by the system subroutine "iox_$write_record" or the system

command "io_call write_record" to write a record at the first location on the remote device.

This entry attaches a switch named "output_stream" with the attach description "record_stream_ user_output -nnl" as shown in [4]. Subsequent writes are then directed through this switch in order to remove the CR-LF transmitted by the Tektronix 4014-1 device.

If the remote device is the 4923 cassette tape drive, the ready mode is turned off, the user is instructed to turn on the device, and the first write is delayed for 20 seconds to permit the user to turn on the device.

If the bytes_to_write is less than 128, "*"'s are written as pad characters.

Data is written to the 4923 tape cassette device in 128 character blocks consisting of 126 data characters, a "stop read" control character (DC3), and a "dispensable" character (LF).

The DC3 character signals the 4923 device to stop data transmission when reading until a "start read" control character (DC1) is detected. The LF character signals the Tektronix 4014-1 terminal of transmission completion.

4.12 write_next_record

Entry write_next_record

Usage:

```
declare iox_$write_record entry (ptr, ptr, fixed bin (21), fixed
    bin (35));

call iox_$write_record (iocb_ptr, buffer_ptr, bytes_to_write,
    error_code);
```

Arguments:

iocb_ptr	switch's control block pointer (Input)
buffer_ptr	data buffer location (Input)
bytes_to_write	characters to output (Input)
error_code	system status code (Output)

Description:

. Called by the system subroutine "iox_$write_record" or the system

command "io_call write_record" to write a record at the next sequential location on the remote device.

If the bytes_to_write is less than 128, "*"'s are written as pad characters.

Data is written to the 4923 tape cassette device in 128 character blocks consisting of 126 data characters, a "stop read" control character (DC3), and a "dispensable" character (LF).

The DC3 character signals the 4923 device to stop data transmission when reading until a "start read" control character (DC1) is detected. The LF character signals the Tektronix 4014-1 terminal that transmission is complete.

4.13 close_device

Entry: close_device

Usage:

```
declare iox_$close entry (ptr, fixed bin (35));

call iox_$close (iocb_ptr, error_code);
```

Arguments:

iocb_ptr	switch's control block pointer (Input)
error_code	system status code (Output)

Description:

Called by the system subroutine "iox_$close" or the system command "io_call close" to close the remote device.

A file terminator block of 126 stars ("*"), a device stop control character (DC4), and a LF char is written as the last data block. This is used to recognize end-of-file during later reads.

The appropriate ASCII control character(s) are transmitted to stop and free the remote device, turn the screen on if it was off, close the I/O switch(s), update the iocb, set the open description to null, and return.

4.14 detach

Entry: detach

Usage:

```
declare iox_$detach_iocb entry (ptr, fixed bin (35));

call iox_$detach_iocb (iocb_ptr, error_code);
```

Arguments:

iocb_ptr	switch's control block pointer (Input)
error_code	system status code (Output)

Description:

Called by the system subroutine "iox_$detach_iocb" or the system command "io_call detach_iocb" to detach the i/o switch designated by the iocb_ptr.

The attach description is set to null, and the entry returns.

4.15 maketree

Entry: maketree

Description:

```
begin maketree;
  get input ASCII data file name;
  get number of keywords;
  query user for presence of class_id's;
  for each vector do;
    get class_name;
    skip keyword(s);
    get or generate class_id;
    if class_name exists in class_list then increment vector_count;
    else allocate next entry and initialize;
    open temp class_file (new or append);
    convert vector data to floating_point
    write floating_point data to temp file;
    close temp class file;
  end;
  open treedata file;
  copy class header info from linked class_list to treedata file;
  copy class_id and floating_point data from each temp file to treedata file;
  delete each temp file;
```

```
  close treedata file;

  call treeiput;

end maketree;
```

4.16 treeiput

Subroutine: treeiput

Call: call treeiput (treename);

Arguments:

treename	eight character name of the MOOS tree to be created

Input File:

A file named "treedata" must be created in the process directory prior to calling treeiput. The format of the "treedata" file is shown in Figure 8.

Output File:

The "sysdata" file reflects the addition of a new tree in the MOOS system, a treename file is created, and a dataclass file is created for each class stored in "treedata".

Description:

The program treeiput creates a MOOS tree named treename from the data stored in the process directory file "treedata". The treeiput program may accept trees with greater than 100 dimensions. The chief difference between the "treedata" format utilized by treeiput and the "filedata" format utilized by fileinput is that each vector in the "treedata" file has a unique user-supplied identification number.

For a more detailed description of the operation of the treeiput program, see the program listing documentation. Also, see the description of the following treeiput$treeoput program documentation.

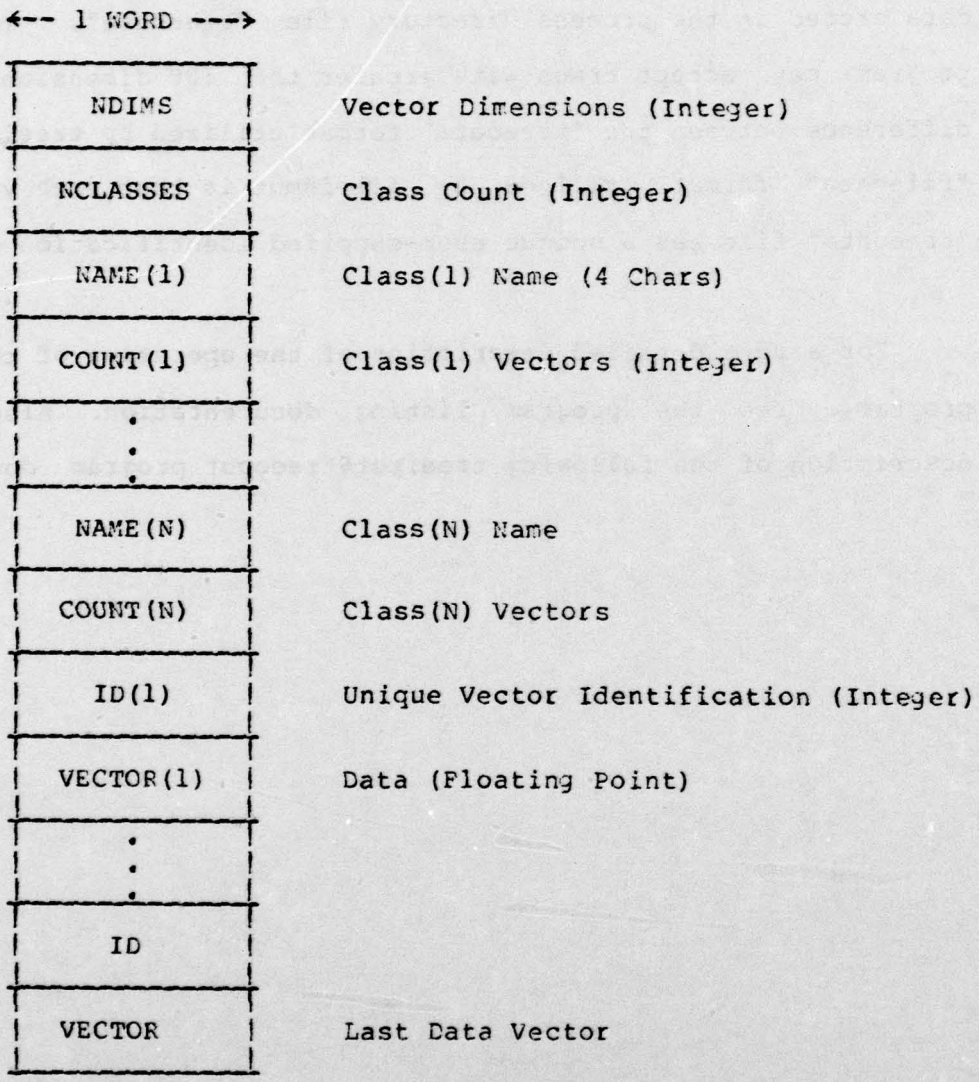


Figure 8 TREEDATA file format

4.17 dumptree

Entry: dumptree

Description:

```
begin dumptree;
  get input MOOS tree file name;
  get output ASCII file name;
  call treeoput;
  query user to write class_id's;
  open input file;
  open output file;
  read ndims;
  read nclasses;
  do loop1 = 1 to nclasses;
    read class_name;
    read class_count;
    allocate class entry;
  end;
  do loop2 = 1 to nclasses;
    get class_count;
    do loop3 = 1 to class_count;
      write class_name to ASCII file;
      write class_id to ASCII file is requested;
```

```
      do loop4 = 1 to ndims;
        read floating_point word;
        write ASCII chars to ASCII file;
      end;
    end;
  end;
  write "/*" to ASCII file;
  close files;
end dumptree;
```

4.18 treeoput

Subroutine: treeiput$treeoput

Synonym: treeoput

Call: call treeiput$treeoput (treename);

call treeoput (treename);

Arguments:

treename eight character name of MOOS tree file to output

Input File:

The tree named treefile must exist in the MOOS file system. The tree may be an excess measurement mode tree.

Output File:

A file named "treedata" is created in the user's process directory and the vectors associated with the MOOS tree named treename are placed in this file. The format of the "treedata" file is given by

the user's description of the treeiput function.

Description:

The program treeoput (or treeiput$treeoput) creates a process directory file named "treedata" and places the vectors associated with the MOOS tree treename into this file. The program treeoput may output trees with greater than 100 dimensions. The chief difference between the "treedata" format created by treeoput and the "filedata" format utilized by the program fileinput is that each vector in the "treedata" file has a unique identification number.

For a more detailed description of the operations of treeiput$oput, see the program listing documentation.

4.19 any

External Procedure:

any

Declaration:

declare any entry (char (*) var, char (*) var) returns (bit (1));

Call:

This is a function procedure which returns a true or false value. Thus its usage is:

if any (choices, arg) then ....

Arguments:

choices	an ASCII string of choices separated by an "|" symbol.
arg	an ASCII string of characters representing the value to search for in the choices.

Description:

This is a recursive procedure. The choices must be in the form "c1|c2|...|cn", where each choice may be of any length.

For example, let us suppose that we wish to determine if the user's reply to a query was "yes", "no", or "maybe". We would test his reply with the any function as follows:

```
if any ("yes|no|maybe", reply) then ....
```

The procedure is very simple, and is adequately documented in the program listing.

4.20 delay

External Procedure:

delay

Declaration:

```
declare delay entry (fixed binary (17));
```

Call:

```
call delay (amount);
```

Arguments:

amount	delay time in seconds (|amount| < 60)

Description:

This procedure is used by the I/O module "remote_data_" to pause before the first read or write when using the 4923 remote cassette device.

If the amount to pause is less than zero, it is set to its positive value, and the delay clock values are displayed for debugging purposes.

4.20 floating point general remarks

The floating point conversion program was written to fulfill a need to convert a floating point number from a source machine to a floating point number on a target machine.

The components of floating point numbers of various machines were analyzed to determine the basic components and necessary operations for conversions. Working from the concepts expressed by Knuth [10, Chapter 4], the following precepts were established:

- The floating point number (e, f) is defined as f*b**(e-q), where f is a signed fraction, |f|<1, b is the base, e is the integer exponent, and q is the bias.

- A floating point number is said to be normalized if the most significant digit of the representation of f is non-zero. That is b**(1/2)≤f≤b**1.

- Exponents from (-b**(n-1)) to (+b**(n-1)-1), where n is the number of exponent bits, are represented by the binary equivalents of 0 (zero) through (b**n-1). For example, if the bias, q, is 128 decimal (200 octal), and there are eight bits in the exponent, then exponents of -128 to +127 are represented by the binary equivalents of 0 (zero) through

255 decimal (0 through 377 octal).

The floating point conversion program employed several user-callable procedures. These user-callable procedures are in the form "X_to_Y", where "X" represents the source machine, and "Y" represents the target machine. For example, if the user wanted to convert a MULTICS floating point word to an IBM floating point word, the call would be to the procedure "MULTICS_to_IBM".

If the user should want to convert a floating point word from or to a machine that is not defined in the procedures, he may call the procedure "convert". The use of this entry is described in the next section. This entry is called by all of the "X_to_Y" entries.

Special procedures were designed to mimic general assembler or machine-language instructions for bit manipulation. Since they are written in PL/1, they are naturally slower than they would be if written in assembler, but are more easily understood. They are described in more detail in the following documentation and in the program documentation listing.

Conversion of exponent bases (such as from IBM base 16 to base 2) was accomplished through an algorithm described by Perry [5]. The interested user is referred to the source paper and the program listing documentation for further information.

The general structure of the floating point conversion program is shown in Figure 9.

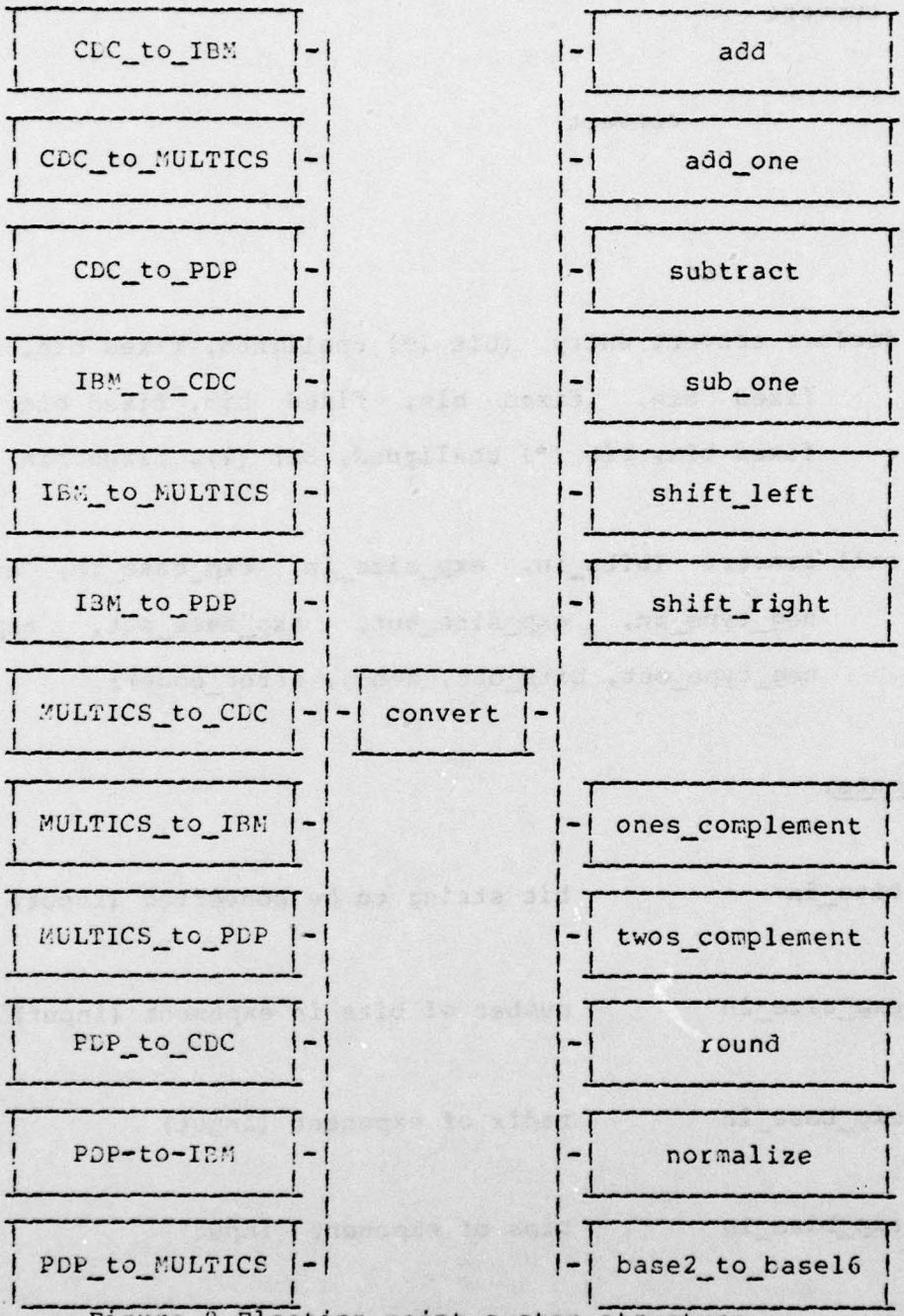


Figure 9 Floating point system structure

4.21 convert

Name: convert

Usage:

```
declare convert entry (bit (*) unaligned, fixed bin, fixed bin,
    fixed bin, fixed bin, fixed bin, fixed bin, fixed bin,
    fixed bin, bit (*) unaligned, bit (1), fixed bin (35));

call convert (bits_in, exp_size_in, exp_base_in, exp_bias_in,
    neg_type_in, exp_size_out, exp_base_out, exp_bias_out,
    neg_type_out, bits_out, debug, error_code);
```

Arguments:

bits_in	bit string to be converted (input)
exp_size_in	number of bits in exponent (input)
exp_base_in	radix of exponent (input)
exp_bias_in	bias of exponent (input)
neg_type_in	type of negative fraction: 2 for two's

	complement, 1 for one's complement, and 0 for signed true form (input)
exp_size_out	number of bits in exponent (output)
exp_base_out	radix of exponent (output)
exp_bias_out	bias of exponent (output)
neg_type_out	type of negative fraction (output)
bits_out	bit string to be created (output)
debug	test switch to monitor program flow and variable values (input)
error_code	result of operations (output)

Description:

This is the floating_point conversion routine. As such, it uses information about the input and output floating point number formats to convert an input bit string to an output bit string.

4.22 CDC_to_IBM

Name: CDC_to_IBM

Usage:

```
declare CDC_to_IBM entry (bit (*), fixed bin, bit (*), bit (1),
     fixed bin (35));

call CDC_to_IBM (bits_in, precision, bits_out, debug,
     error_code);
```

Arguments:

bits_in	bit string to be converted. It must be in the form of: fraction sign, exponent, fraction (input)
precision	number of words of precision, usually 1 - 4 (input)
bits_out	bit string for output machine. Will be in format of: fraction sign, exponent, fraction (output)

debug	logical switch to trace program flow and variable assignments (input)
error_code	indicator of error in system (output)

Description:

This entry converts a floating point word bit string (bits_in) from CDC machine format to IBM machine format (bits_out).

Care should be taken to insure correct format of the input bit string before the call.

4.23 CDC_to_MULTICS

Name: CDC_to_MULTICS

Usage:

```
declare CDC_to_MULTICS entry (bit (*), fixed bin, bit (*), bit
     (1), fixed bin (35));

call CDC_to_MULTICS (bits_in, precision, bits_out, debug,
     error_code);
```

Arguments:

bits_in	bit string to be converted. It must be in the form of: fraction sign, exponent, fraction (input)
precision	number of words of precision, usually 1 - 4 (input)
bits_out	bit string for output machine. Will be in format of: fraction sign, exponent, fraction (output)

debug	logical switch to trace program flow and variable assignments (input)
error_code	indicator of error in system (output)

Description:

This entry converts a floating point word bit string (bits_in) from CDC machine format to MULTICS machine format (bits_out).

Care should be taken to insure correct format of the input bit string before the call.

4.24 CDC_to_PDP

Name: CDC_to_PDP

Usage:

```
declare CDC_to_PDP entry (bit (*), fixed bin, bit (*), bit (1),
    fixed bin (35));

call CDC_to_PDP (bits_in, precision, bits_out, debug,
    error_code);
```

Arguments:

bits_in	bit string to be converted. It must be in the form of: fraction sign, exponent, fraction (input)
precision	number of words of precision, usually 1 - 4 (input)
bits_out	bit string for output machine. Will be in format of: fraction sign, exponent, fraction (output)

debug	logical switch to trace program flow and variable assignments (input)
error_code	indicator of error in system (output)

Description:

This entry converts a floating point word bit string (bits_in) from CDC machine format to PDP machine format (bits_out).

Care should be taken to insure correct format of the input bit string before the call.

4.25 IBM_to_CDC

Name: IBM_to_CDC

Usage:

```
declare IBM_to_CDC entry (bit (*), fixed bin, bit (*), bit (1),
    fixed bin (35));

call IBM_to_CDC (bits_in, precision, bits_out, debug,
    error_code);
```

Arguments:

bits_in	bit string to be converted. It must be in the form of: fraction sign, exponent, fraction (input)
precision	number of words of precision, usually 1 - 4 (input)
bits_out	bit string for output machine. Will be in format of: fraction sign, exponent, fraction (output)

debug	logical switch to trace program flow and variable assignments (input)
error_code	indicator of error in system (output)

Description:

This entry converts a floating point word bit string (bits_in) from IBM machine format to CDC machine format (bits_out).

Care should be taken to insure correct format of the input bit string before the call.

4.26 IBM_to_MULTICS

Name: IBM_to_MULTICS

Usage:

```
declare IBM_to_MULTICS entry (bit (*), fixed bin, bit (*), bit
     (1), fixed bin (35));

call IBM_to_MULTICS (bits_in, precision, bits_out, debug,
     error_code);
```

Arguments:

bits_in	bit string to be converted. It must be in the form of: fraction sign, exponent, fraction (input)
precision	number of words of precision, usually 1 - 4 (input)
bits_out	bit string for output machine. Will be in format of: fraction sign, exponent, fraction (output)

debug	logical switch to trace program flow and variable assignments (input)
error_code	indicator of error in system (output)

Description:

This entry converts a floating point word bit string (bits_in) from IBM machine format to MULTICS machine format (bits_out).

Care should be taken to insure correct format of the input bit string before the call.

4.27 IBM_to_PDP

Name: IBM_to_PDP

Usage:

```
declare IBM_to_PDP entry (bit (*), fixed bin, bit (*), bit (1),
    fixed bin (35));

call IBM_to_PDP (bits_in, precision, bits_out, debug,
    error_code);
```

Arguments:

bits_in	bit string to be converted. It must be in the form of: fraction sign, exponent, fraction (input)
precision	number of words of precision, usually 1 - 4 (input)
bits_out	bit string for output machine. Will be in format of: fraction sign, exponent, fraction (output)

debug	logical switch to trace program flow and variable assignments (input)
error_code	indicator of error in system (output)

Description:

This entry converts a floating point word bit string (bits_in) from IBM machine format to PDP machine format (bits_out).

Care should be taken to insure correct format of the input bit string before the call.

4.28 MULTICS_to_CDC

Name: MULTICS_to_CDC

Usage:

```
declare MULTICS_to_CDC entry (bit (*), fixed bin, bit (*), bit
     (1), fixed bin (35));

call MULTICS_to_CDC (bits_in, precision, bits_out, debug,
     error_code);
```

Arguments:

bits_in	bit string to be converted. It must be in the form of: fraction sign, exponent, fraction (input)
precision	number of words of precision, usually 1 - 4 (input)
bits_out	bit string for output machine. Will be in format of: fraction sign, exponent, fraction (output)

debug	logical switch to trace program flow and variable assignments (input)
error_code	indicator of error in system (output)

Description:

This entry converts a floating point word bit string (bits_in) from MULTICS machine format to CDC machine format (bits_out).

Care should be taken to insure correct format of the input bit string before the call.

4.29 MULTICS_to_IBM

Name: MULTICS_to_IBM

Usage:

```
declare MULTICS_to_IBM entry (bit (*), fixed bin, bit (*), bit
     (1), fixed bin (35));

call MULTICS_to_IBM (bits_in, precision, bits_out, debug,
     error_code);
```

Arguments:

bits_in	bit string to be converted. It must be in the form of: fraction sign, exponent, fraction (input)
precision	number of words of precision, usually 1 - 4 (input)
bits_out	bit string for output machine. Will be in format of: fraction sign, exponent, fraction (output)

debug	logical switch to trace program flow and variable assignments (input)
error_code	indicator of error in system (output)

Description:

This entry converts a floating point word bit string (bits_in) from MULTICS machine format to IBM machine format (bits_out).

Care should be taken to insure correct format of the input bit string before the call.

4.30 MULTICS_to_PDP

Name: MULTICS_to_PDP

Usage:

```
declare MULTICS_to_PDP entry (bit (*), fixed bin, bit (*), bit
     (1), fixed bin (35));

call MULTICS_to_PDP (bits_in, precision, bits_out, debug,
     error_code);
```

Arguments:

bits_in	bit string to be converted. It must be in the form of: fraction sign, exponent, fraction (input)
precision	number of words of precision, usually 1 - 4 (input)
bits_out	bit string for output machine. Will be in format of: fraction sign, exponent, fraction (output)

debug	logical switch to trace program flow and variable assignments (input)
error_code	indicator of error in system (output)

Description:

This entry converts a floating point word bit string (bits_in) from MULTICS machine format to PDP machine format (bits_out).

Care should be taken to insure correct format of the input bit string before the call.

4.31 PDP_to_CDC

Name: PDP_to_CDC

Usage:

```
declare PDP_to_CDC entry (bit (*), fixed bin, bit (*), bit (1),
    fixed bin (35));

call PDP_to_CDC (bits_in, precision, bits_out, debug,
    error_code);
```

Arguments:

bits_in	bit string to be converted. It must be in the form of: fraction sign, exponent, fraction (input)
precision	number of words of precision, usually 1 - 4 (input)
bits_out	bit string for output machine. Will be in format of: fraction sign, exponent, fraction (output)

debug	logical switch to trace program flow and variable assignments (input)
error_code	indicator of error in system (output)

Description:

This entry converts a floating point word bit string (bits_in) from PDP machine format to CDC machine format (bits_out).

Care should be taken to insure correct format of the input bit string before the call.

4.32 PDP_to_IBM

Name: PDP_to_IBM

Usage:

```
declare PDP_to_IBM entry (bit (*), fixed bin, bit (*), bit (1),
     fixed bin (35));

call PDP_to_IBM (bits_in, precision, bits_out, debug,
     error_code);
```

Arguments:

bits_in	bit string to be converted. It must be in the form of: fraction sign, exponent, fraction (input)
precision	number of words of precision, usually 1 - 4 (input)
bits_out	bit string for output machine. Will be in format of: fraction sign, exponent, fraction (output)

debug	logical switch to trace program flow and variable assignments (input)
error_code	indicator of error in system (output)

Description:

This entry converts a floating point word bit string (bits_in) from PDP machine format to IBM machine format (bits_out).

Care should be taken to insure correct format of the input bit string before the call.

4.33 PDP_to_MULTICS

Name: PDP_to_MULTICS

Usage:

```
declare PDP_to_MULTICS entry (bit (*), fixed bin, bit (*), bit
     (1), fixed bin (35));

call PDP_to_MULTICS (bits_in, precision, bits_out, debug,
     error_code);
```

Arguments:

bits_in	bit string to be converted. It must be in the form of: fraction sign, exponent, fraction (input)
precision	number of words of precision, usually 1 - 4 (input)
bits_out	bit string for output machine. Will be in format of: fraction sign, exponent, fraction (output)

debug	logical switch to trace program flow and variable assignments (input)
error_code	indicator of error in system (output)

Description:

This entry converts a floating point word bit string (bits_in) from PDP machine format to MULTICS machine format (bits_out).

Care should be taken to insure correct format of the input bit string before the call.

4.34 add

Name: add

Usage:

```
declare add entry (bit (*), bit (*), bit (1), fixed bin (35));

call add (bits1, bits2, debug, error_code);
```

Arguments:

bits1	bit string to add (input/output)
bits2	bit string to add (input)
debug	test switch (input)
error_code	system error (output)

Description:

This procedure adds two bit strings serially from the right (least significant bit, lsb) to the left (most significant bit, msb)

using the concept of a one-bit full adder [11].

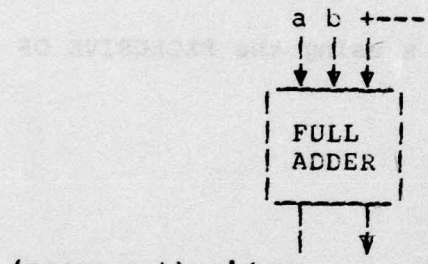
A one-bit full adder must be capable of accepting three inputs. Two are the original bits (shown as "a" and "b" in Figure 10.), and the third input "c" is the carry out from the previous bit addition. For the first add, the c input is initialized to "0"b. There are two outputs of the adder: the sum bit, "s", and the carry out bit, "c'".

The canonical minterm form for s using the EXCLUSIVE OR operation is:

$$s = a \neq b \neq c.$$

The carry out term is derived from:

$$c' = bc \vee ac \vee ab.$$ ("V" denotes "OR")



$s \equiv a \neq b \neq c$ ($\neq$ denotes EXCLUSIVE OR)

$c \equiv bc \vee ac \vee ab$ ($\vee$ denotes OR)

Figure 10 Full Adder Design

4.35 add_one

Name: add_one

Usage:

```
declare add_one entry (bit (*), bit (1), fixed bin (35));
call add_one (bits, debug, error_code);
```

Arguments:

bits	bit string to be incremented (Input/output)
debug	test switch (Input)
error_code	set to 1 if overflow, else set to 0 (output)

Description:

This procedure adds one bit to the input bit string using Boolean algebra. Addition of the carry_bit continues from the right-most (lsb) to the left-most (msb) bit until the carry bit is zero or the msb is reached. If the carry bit is set after the addition is complete, the error_code is set to 1 to indicate overflow.

4.36 subtract

Name: subtract

Usage:

```
declare subtract entry (bit (*), fixed bin, bit (1), fixed bin
    (35));

call subtract (bits, value, debug, error_code);
```

Arguments:

bits	bit string to subtract value from (input/output)
value	value to subtract from bit string (input)
debug	test switch (input)
error_code	procedure result (output)

Description:

This procedure subtracts the given value from the bit string by adding the one's complement bit representation of te value to the bit string.

4.37 sub_one

Name: sub_one

Usage:

```
declare sub_one entry (bit (*), bit (1), fixed bin (35));

call sub_one (bits, debug, error_code);
```

Arguments:

bits	bit string to be decremented (Input/output)
debug	test switch (Input)
error_code	set to 1 if overflow, else set to 0 (output)

Description:

This procedure subtracts one bit from the input bit string using Boolean algebra.

4.38 shift_left

Name: shift_left

Usage:

```
declare shift_left entry (bit (*), fixed bin, bit (1), fixed bin
     (35));

call shift_left (bits, places, debug, error_code);
```

Arguments:

bits	bit string to subtract value from (input/output)
places	number of places to shift left (input)
debug	test switch (input)
error_code	procedure result (output)

Description:

This procedure shifts the bit array left the number of specified

places. The bit string is zero filled on the right.

If the number of places to shift is greater than the size of the bit string, the error_code is set to 1.

If any of the bits shifted off the left end are equal to "1"b, the error_code is set to 2 to indicate truncation.

4.39 shift_right

Name: shift_right

Usage:

```
declare shift_right entry (bit (*), fixed bin, bit (1), fixed bin
     (35));

call shift_right (bits, places, debug, error_code);
```

Arguments:

bits	bit string to subtract value from (input/output)
places	number of places to shift right (input)
debug	test switch (input)
error_code	procedure result (output)

Description:

This procedure shifts the bit string right the number of places

specified. The string is zero filled on the left.

If the number of places to shift is greater than the size of the bit string, the error_code is set to 1.

If any of the bits shifted of the right end are equal to "1"b, the error_code is set to 2 to indicate truncation.

4.40 ones_complement

Name: ones_complement

Usage:

```
declare ones_complement entry (bit (*), bit (1), fixed bin (35));

call ones_complement (bits, debug, error_code);
```

Arguments:

bits	bit string to complement (input/output)
debug	test switch (input)
error_code	procedure result (output)

Description:

This procedure produces the one's complement of the input bit string by replacing each bit in the string with its complementary (negated) value.

4.41 twos_complement

Name: twos_complement

Usage:

```
declare twos_complement entry (bit (*), bit (1), fixed bin (35));

call twos_complement (bits, debug, error_code);
```

Arguments:

bits	bit string to complement (input/output)
debug	test switch (input)
error_code	procedure result (output)

Description:

This procedure converts the given bit string to the two's complement representation by adding the value of one to the one's complement representation of the bit string.

4.42 round

Name: round

Usage:

```
declare round entry (bit (*), bit (*), fixed bin, bit (1), fixed
    bin (35));
call round (exponent, fraction, places, debug, error_code);
```

Arguments:

exponent	exponent (input/output)
fraction	fraction (input)
places	number of places to round (input)
debug	test switch (input)
error_code	procedure result (output)

Description:

The floating_point number is rounded to the specified number of places by adding one and then normalizing the result.

4.43 normalize

Name: normalize

Usage:

```
declare normalize entry (bit (*), bit (*), bit (1), fixed bin
    (35));

call normalize (exponent, fraction, debug, error_code);
```

Arguments:

exponent	exponent (input/output)
fraction	fraction (input)
debug	test switch (input)
error_code	procedure result (output)

Description:

This procedure normalizes a floating point number by shifting the fraction bits left one bit and adding one to the exponent bits until

the leftmost fraction bit is set ("1"b). This is described by [10, p.181]:

> "A floating-point number (e, f) is said to be <u>normalized</u> if the most significant digit of the representation of f is non-zero, so that
>
> $1/b \leq |f| < 1$;
>
> or if f = 0 and e has its smallest possible value" .

4.44 base2_to_base16

Name: base2_to_base16

Usage:

```
declare base2_to_base16 entry ((*) bit (1) unaligned, (*) bit (1)
     unaligned, bit (1), fixed bin (35));

call base2_to_base16 (exponent, fraction, debug, error_code);
```

Arguments:

exponent	bit array representation (input/output)
fraction	bit array representation (input/output)
debug	test switch (input)
error_code	procedure result (output)

Description:

This procedure is used to convert base2 floating point numbers to the base16 format used by IBM.

The algorithm used in this procedure was taken from [9].

The rational approximant, p/q, mentioned in the paper was chosen so that d = (b↑(p/q))(d↑epsilon) was exact. That is, with p/q = 1/4 and b = 16, b↑(1/4) is exactly equal to d (2). Thus, epsilon is zero.

The base16 exponent (u, in the paper) is derived by multiplying the base2 exponent (s, in the paper) by p/q. Since p/q = 1/4, the base2 exponent was divided by 4 (shifted right by 2). The multiplier (FP(ps/q), in the paper) was obtained by "catching" the bits as they shifted off the right end during the division.

This multiplier, which will have only the values 0, 1, 2, or 3, is then used to compute the non-normalized base16 fraction, b↑FP(ps/q), by multiplying (shifting left) the base2 fraction.

The fraction bits are checked for overflow, and normalized by dividing the fraction by 16 (shifting right by 4), and subtracting one from the exponent for each division.

5 CTS Functional Description

5.1 CTS Structure

There are four major subroutines within CTS: process_args, process_keywords, process_commands, and process_program. Each of these have been broken down into smaller modules, each having a specific function. Process_commands, has been broken into the greatest number of modules: nine in all.

The first major subroutine, process_args, receives all controlling and pathname arguments input by the user, verifies that they are legal arguments, and set switches accordingly. The second major subroutine, process_keywords, obtains vector keywords from the user. The third major subroutine, process_commands, uses 9 smaller modules, five of which are command routines, which process the commands individually. The sixth module that process_commands uses, advance, is used to return each token or word to the five command routines. It does so, by using a seventh module, get_char, which returns individual characters. Advance combines the individual characters to form a word of the command line. The eighth module is an error routine which can be called by process_five commands, by the five command modules, by advance, or by get_char when an error is encountered while processing a command. The ninth module, query_user, is used to obtain the commands and any corrections from the user. The fourth major subroutine is process_ program which indents and compiles

the generated PL/1 program.

The names in the structural design of CTS are shown in the accompanying figure. The names of the routines and modules are in capital letters while the functional aspects are in lower case letters.

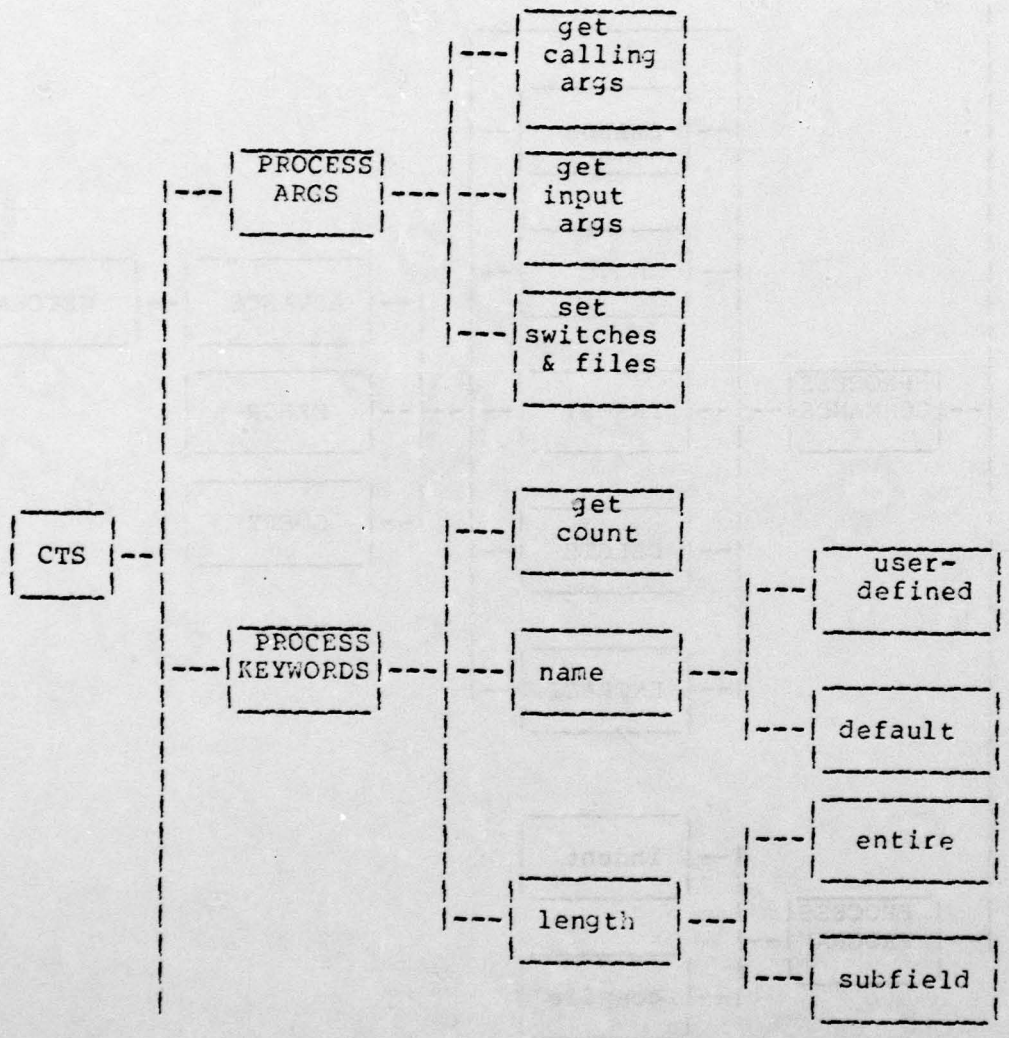


Figure 11 CTS Structure

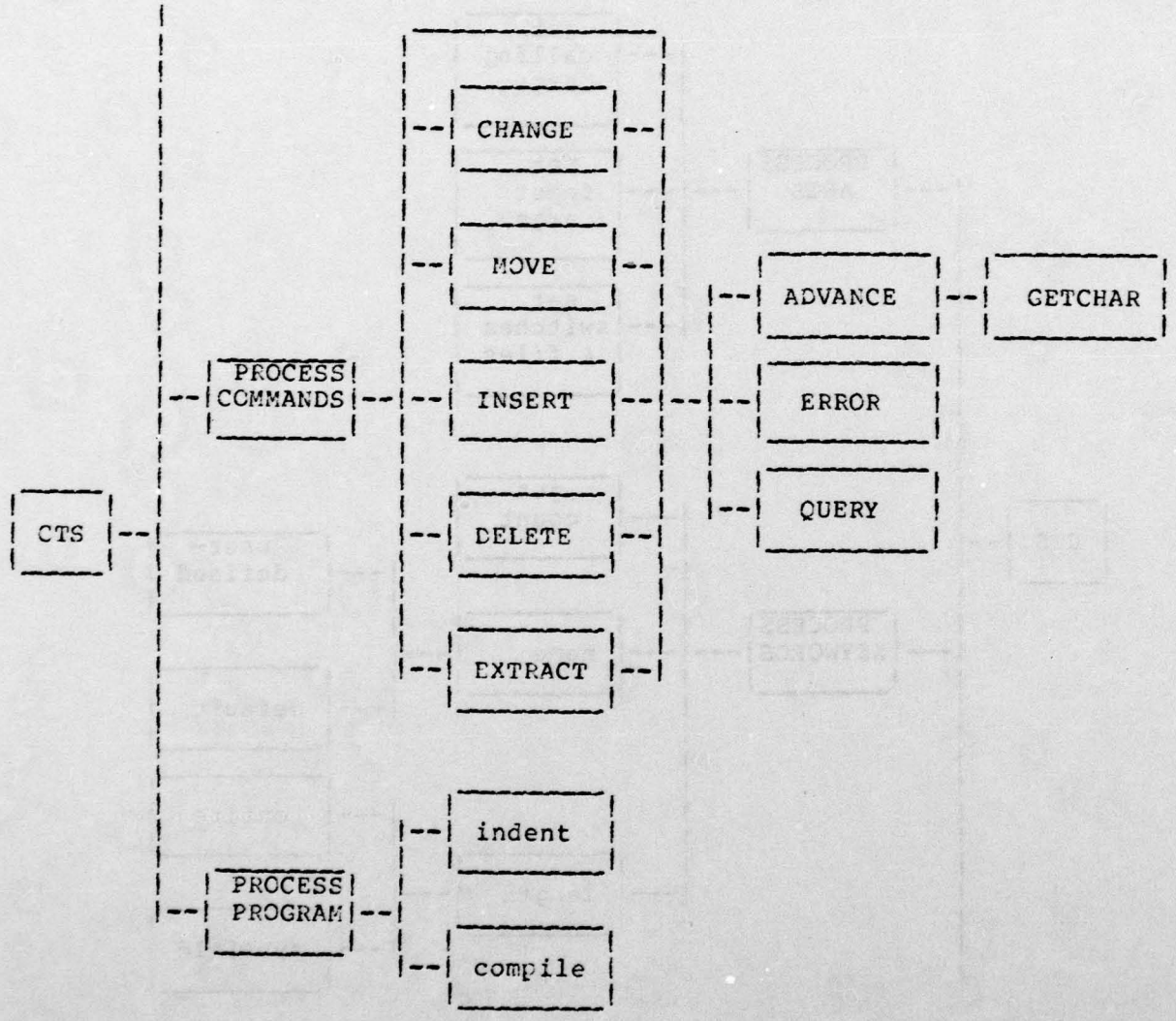


Figure 11 CTS Structure (continued)

5.2 cts

Function: cts {-pathname arguments} {-control arguments}

Parameters: arguments may be specified when calling CTS. If omitted, user will be prompted for them.

-old pathname required pathname of the input segment or multisegment ASCII RDE data file. If not present, CTS will query the user for the data file segment name.

-new pathname pathname of ASCII RDE data file to be created from applying CTS to the input data file. If no pathname is entered, the default pathname is "empty".

-in pathname pathname of the segment from which CTS control arguments and commands are to be taken. This segment name must have ".cts" as a suffix, but is not required in the command line. May be used only when calling CTS.

-out pathname pathname of the segment to which CTS control arguments and commands are to be copied to. A suffix of ".cts" will be appended to the pathname. This argument is incompatible with the "-in" argument. Default is no output segment.

-save pathname, -sv pathname causes generated PL/1 program to be saved in the current working directory. A suffix of ".pl1" is appended to the pathname, but is not required in the command line.

-list causes a listing of the generated PL/1 program. Default is no listing.

-check, -ck checks syntax of input commands. There is no translation or program generation.

-noxqt, -nx Used with the "-save" argument to just generate the PL/1 program

-noquery, -nq inhibits message asking user for control arguments and commands. Should only be used when working in the batch mode.

-echo causes a display at the terminal of the input control arguments and commands as they are read in. Default is no echo.

-debug, -db causes a trace of statements executed. Is mainly for use by a systems maintenance person.

-menu prints a listing of all the arguments available in CTS.

-panic will notify the user if he attempts to put a vector into two different classes and will terminate execution of the generated PL/1 program. Default is notification of this occurrence, creation of a file containing all vectors placed in more than one class, and continued execution of the generated PL/1 program.

-arguments argstring, -args argstring passes arguments "argstring" to the PL/1 compiler as compilation arguments for compilation of the generated PL/1 program.

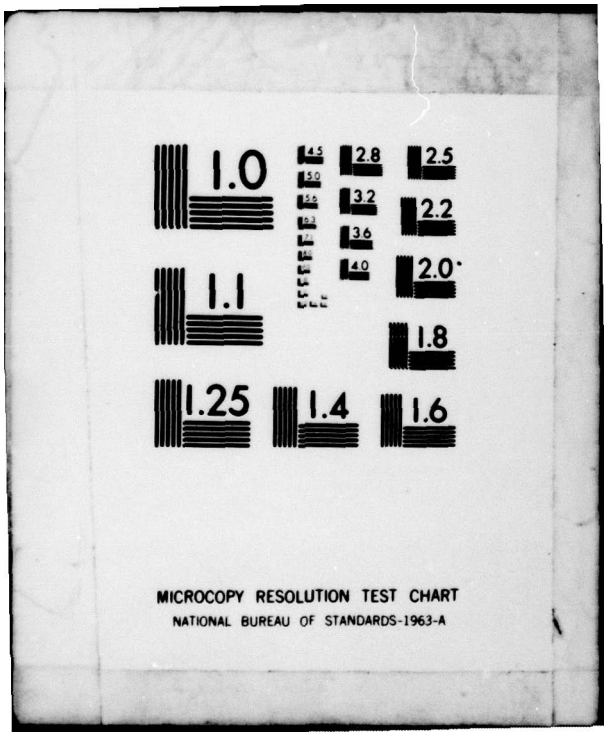
Description:

The user function CTS reads in control arguments either from an input file or from the terminal, then calls cts_process_args to process the arguments and set switches. CTS then calls cts_process_keywords to obtain keywords from the user. The next subroutine called by CTS is cts_process_commands which processes the edit and extraction commands. The last subroutine called is cts_process_program which indents and compiles the generated PL/1 program. CTS can then list and execute the generated program if the user so desires. All switches are then turned off, and CTS is completed.

AD-A080 625 PATTERN ANALYSIS AND RECOGNITION CORP ROME N Y F/G 9/2
MULTICS REMOTE DATA ENTRY SYSTEM. VOLUME I.(U)
OCT 79 D BIRNBAUM, J J CUPAK, J D DYAR F30602-77-C-0174
UNCLASSIFIED PAR-79-59 RADC-TR-79-265-VOL-1 NL

3 OF 3
AD
A080625

END
DATE
FILMED
3-80
DDC


1.0
1.1
1.25
1.4
1.6
2.8
3.2
3.6
4.0
2.5
2.2
2.0
1.8
MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
begin cts;

get default working directory;
get number of args from command line when calling cts;
if number of args>0 then do m=1 to number of args;
     get a pointer to arg;
     verify is a legal argument;
     if a double word argument, get second word;
     call cts_process_args;
end;
open input file;
if no args in command line then do;
     get arg from user or input file;
     verify is a legal argument;
     if a double word argument get second word;
     call cts_process_args;
end;
if user wants args written to outfile, then do;
     open outfile;
     write each argument to outfile;
end;
if user wants to know flow of cts then do;
     print each argument and value of its switch;
end;
call cts_process_keywords;
```

```
call cts_process_commands;
if an error in processing of commands, then delete generated PL/1 program;
else do;
     if pathname of generated PL/1 program ="empty" then rename to "gfile.pl1";
     call cts_process_program;
     if user wants a listing, then print listing;
     if user wants the generated program executed, then do;
          if severity of errors<2 then execute program;
          else ask user if he wants program executed;
          if "yes" then execute program;
     end;
end;
close infile;
if opened outfile, then close outfile;
if user does not want generated program saved, then delete program;
turn all argument switches off;
end cts;
```

5.3 cts_process_args

Function: call cts_process_args (arg1, arg2, old_path, new_path, in_path, out_path, save_path, arg_string)

Parameters:

arg1	word in a 2 word argument or only word in a 1 word argument
arg2	second word in a 2 word argument
old_path	pathname of old data_file
new_path	pathname of new data file created from applying cts to old data file
in_path	pathname of file containing all cts arguments and commands
out_path	pathname of file to which all cts arguments and commands are written
save_path	pathname under which the generated PL/1 program is saved
arg_string	string of arguments which are passed to the PL/1 compiler

and are used in compilation of the generated PL/1 program

Description:

The subroutine CTS_process_args sets switches based on the user-supplied CTS arguments. If a switch has previously been set, the user is notified of this condition.

```
begin cts_process_args;
if arg1 is a legal one word argument then do;
     if switch is already set then do;
          print error message;
     end;
     else set switch;
end;
if arg1 is the first word of a legal 2 word argument & (arg1 ^=("-save" | "-arg
  if switch is already set then do;
       print error message;
   end;
   else do;
        set switch;
        type_path = arg2;
    end;
end;
else if arg1 = "-save" then do;
```

```
     if switch is already set then do;
          print error message;
     end;
     else do;
          verify saved program ^= "cts.pl1" | "cts";
          if does, notify user is not permissible;
          ask user if he wants to try another program;
          if he does, then get new program name;
     end;
     set switch;
     if ".pl1" not in program name, add it;
     save_path = arg2;
end;
else if arg1 = "-arguments" | "-args" then do;
     if switch already set then do;
          print error message;
     end;
     else do;
          set switch;
          arg_string = arg2;
     end;
end;
end cts_process_args;
```

5.4 cts_process_keywords

Function: call cts_process_keywords (number_of_keywords, keyword_table, in, out, echo, noquery, debug)

Parameters:

number_of_keywords	number of keywords in each vector in data file
keyword_table	pointer to table of keywords
in	switch which determines whether or not to read from a file containing cts args and commands - same as the argument "-in pathname"
out	switch which determines whether or not to write to a file the cts args and commands - same as the argument "-out pathname"
echo	switch which dethermines whether or not to echo each line read in - same as the "-echo" argument
noquery	switch which determines whether or not to query the user - same as the "-noquery" argument.

debug switch which determines whether or not user wants to see the flow of the program - same as the "-debug" argument

Description:

cts_process_keywords function is intended to obtain the number of keywords(if any), to allocate storage for those keywords, and then to ask the user for each keyword and its subfields (if any).

```
begin cts_process_keywords;

get number_of_keywords;
allocate storage for keywords;
do loop for number of keywords;
     if skip_keyword switch on then do;
          generate keyword;
          get subfields;
          if subfields, add to keyword;
     end;
     else do;
          ask user for keyword and subfields;
          if keyword = "skip" then do;
               generate keyword;
               if no subfields ask user for subfields;
```

```
                    set skip_keyword switch on;
               end;
          end;
     end;
     if subfields in keyword, then do;
          check if one or 2 subfields;
          if 2 subfields then do;
               verify subfield before comma is an integer;
               verify subfield after comma is an integer;
          end;
          else if only one subfield verify subfield is an integer;
          store keyword and its subfields in array;
     end;
     if user wants to see flow of program, then do;
          print keywords and subfields;
     end;
     if user wants keywords saved in a file then do;
          write keyword and subfields to outfile;
     end;
     end cts_process_keywords;
```

5.5 cts_process_commands

Function: call cts_process_commands (old_path, new_path, save_path, in_path, out_path, number_of_keywords, keyword_table)

Parameters:

old_path	pathname of old data_file
new_path	pathname of new data file created from applying cts to old data file
save_path	pathname under which the generated PL/1 program is saved
in_path	pathname of file containing all cts arguments and commands
out_path	pathname of file to which all cts arguments and commands are written
number_of_keywords	number of keywords in the vectors in the data file
keyword_table	pointer to table containing keywords

```
begin cts_process_commands;

find out if classname in vectors;
if "yes" then turn classname switch on;
find out if vector ID in vectors;
if "yes" turn vectorid switch on;
get both process and default working directory;
open gfile;
if conversion error occurs then do;
     notify user of conversion error
     reset onsource value;
     reset token value;
     set error_flag_g;
end;
open input file
if not "-check", then generate declarations for generated PL/1 program;
get first command line;
if input line = "end" then do;
     notify user that cts_process_commands is terminated;
     set error_code;
     set error_flag;
end;
do while (^even number of quotes);
     see if even number of quotes in command line;
```

```
     if not, then call error routine;
end;
do while (^end of commands);
     call advance subroutine to get token;
     depending on token, generate code for generated PL/1 program;
     do while (token^="end");
          if user uses edit command after extract command then do;
               notify user that processing of commands is terminated;
               notify user of no execution of generated program;
               set noxqt switch;
               set error_code;
               set token = "end";
          end;
          do while (^correct_token);
               if token = "change"|"insert"|"delete"|"move"  then do;
                    call subroutine to process the command;
                    set correct_token switch to on;
               end;
               else if token = "extract" then do;
                    if not "-check" then do;
                         set extract switch on;
                         if edit commands preceded, then do;
                              generate code to PL/1 program;
                              turn edit switch to off;
                         end;
```

```
                    end;
                    call extract subroutine to process command;
                    set correct_token switch to on;
               end;
               else if token = "end" then set correct_token switch on;

               else if none of the above then do;
                    call error subroutine;
                    if error_flag then set correct_token switch on;
               end;
          end;
          if an error_flag then token = "end";
          if token ^="end" then do;
               turn command_flag off;
               call advance subroutine for new token;
          end;
     end;
     if error_code = 1 then set end_flat on;
     if error_flag then set token ="stop";
     if token = "end" & ^end_flag the do;
          notify user that command processing is completed;
          if ^extract_switch and not "-check" then generate code;
          set end_flag to on;
     end;
     if command_flag on, turn it off;
```

```
     if error_flag for quitting is on, then do;
          turn error_flag off;
          set correct_token to off;
          do while (^good_token);
               call advance for new token;
               turn token switch off if on;
               if token = "," then turn good_token on;
               else if end of line then turn good_token on;
          end;
     end;
     else if error_flag for ceasing is on then do;
          notify user that processing of all commands is terminated;
          turn end_flag switch on;
     end;
     if token = "end" then turn end_flag switch on;
end;
if not "-check" and error_code = 0 then do;
     close file of generated PL/1 program;
     open file of generated PL/1 program;
     open new file;
     on endfile condition set endfile flag on;
     do while (^endfile flag);
          read line of generated program file;
          if ^end of file and line ^= a certain declaration then
          write line out to new file;
```

```
          else do;
               write line out to new file;
               write more code to new file;
          end;
     end;
     if classname switch is on, turn it off;
     if classid switch is on, turn it off;
     close both generated program file and new file;
     delete the generated program file;
     rename new file to pathname of old generated program;
end;
end cts_process_commands;
```

5.6 cts_process_program

Function: call cts_process_program (program_name, pl1_args, sv, debug, error_code)

Parameters:

program_name	name of generated PL/1 program - same as pathname specified by "-save pathname"
pl1_args	compilation arguments - same as "argstring" specified by "-args argstring"
sv	severity of compilation errors
debug	same as logic_switch.debug in cts - indicates user wants to see the flow of the program
error_code	code which indicates if an error hs occurred within cts_process_program

Description:

cts_process_program first verifies that the program to be tested

exists in the working directory. Then if the program does exist, the program is indented and compiled. Any compilation errors are written to an error file and once compilation is completed, the error file is examined to determine the highest severity of compilation errors (if any). If the severity of errors is >2 then an error_code is set = 2 and the user is notified of the occurrence of compilation errors.

```
begin cts_process_program;
verify program to be tested is not "cts_process_program.pll";
if it is then do;
     notify user that testing of "cts_process_program" is not permissible;
     set error_code;
end;
else do;
     get default working directory;
     verify program exists in working directory;
     if it doesn't then do;
          notify user that program does not exist;
          set error_code;
     end;
     else do;
          indent generated program;
          examine pll_args for the argument "-sv";
          if found, then do;
               notify user that it has been ignored;
```

```
                delete it from the pll_args;
            end;
            compile generated program;
            on end of file then set eof_switch on;
            open error file;
            read line of error_file;
            do while (^eof_switch);
                determine the maximum severity   of errors in the file;
            end;
            close error file;
            if severity > 2 then do;
                notify the user that a compilation error has occurred;
                set error_code;
            end;
        end;
    end;
    end cts_process_program;
```

5.7 cha_nge

Function: call cha_nge (in_line, number_of_keywords, command_index, char_pos, keyword_table, line_len)

Parameters:

in_line	input command(s) line
number_of_keywords	number of keywords in each vector in the data file
command_index	beginning character position of change command in input command line
char_pos	position in command line of current character under consideration
keyword_table	pointer to table containing keywords of the vectors in the data file
line_len	length of input command(s) line

Description:

Cha_nge parses a change command to verify that the syntax of the command is correct. Once the parsing is done, code is generated for the save PL/1 program if the user has not specified the argument "-check" when giving cts arguments. If the argument "-check" was given, then no code is generated for the PL/1 program

```
begin cha_nge;
if conversion error occurs then do;
     notify user of conversion error and
     that processing of current command is terminated;
     reset onsource value;
     reset token value;
     set error_flag_c;
end;
set command value to "change ";
call advance subroutine for next token;
do while (^okay);
     if token ^= "data" | "field" then do;
          if token = "keyword" then do;
               add token to command value;
               call advance subroutine for next token;
               if token ^= "(" then call error subroutine;
               else add token to command value;
               call advance subroutine for next token;
```

```
                if token ^= integer then call error subroutine;
                else add token to command value;
                call advance subroutine for next token;
                if token ^= ")" then call error subroutine;
                else add token to command value;
           end;
           else loop through keyword table for token keyword;
                if it is, add token to command value;
           end;
      end;
      else do;
           if token = "data" then add token to command value;
           else if token = "field" then do;
                add token to command value;
                call advance subroutine for next token;
                if token ^= "(" then call error subroutine;
                else add token to command value;
                call advance subroutine for next token;
                if token ^= integer then call error subroutine;
                else add token to command value;
                call advance subroutine for next token;
                if token ^= ")" then call error subroutine;
                else add token to command value;
           end;
      end;
```

```
end;
call advance subroutine for next token;
if first character in token = integer then do;
     if token ^= integer then call error subroutine;
     else add token to command value;
     call advance subroutine for next token;
     if token = "," then do;
          add token to command value;
          call advance subroutine for next token;
          if token ^= integer then call error subroutine;
          else add token to command value;
     end;
     else set switch advance subroutine_flag off;
end;
if advance subroutine_flag on then get next token;
if token ^= a relational operator then call error subroutine;
else add token to command value;
call advance subroutine for next token;
if token ^= a double quote then do;
     see if token = number;
     if it does, add token to command value;
     else conversion condition occurs;
end;
else do;
     if token ^= double quote then call error subroutine;
```

```
     else add token to command value;
     call advance subroutine for next token;
     add token to command value;
     call advance subroutine for next token;
     if token ^= a double quote then call error subroutine;
     else add token to command value;
end;
call advance subroutine for next token;
if token ^= "to" then call error subroutine;
else add token to command value;
call advance subroutine for next token;
if token ^= a double quote then call error subroutine;
else add token to command value;
call advance subroutine for next token;
add token to command value;
call advance subroutine for next token;
if token ^= double quote then call error subroutine;
else add token to command value;
call advance subroutine for next token;
if token ^= period then call error subroutine;
else add token to command value;
if out.switch on then write command to outfile;
if "-check" switch off then generate code PL/1 program;
```

5.6 in_sert

Function: call in_sert (in_line, line_len, command_index, char_pos)

Parameters:

in_line	input command(s) line
line_len	length of input command(s) line
command_index	beginning character position of insert command in input command(s) line
char_pos	position in command line of current character under consideration

Description:

In_sert parses an insert command to verify that the syntax of the insert command is correct. Once the parsing is complete, code is generated for the saved PL/1 program if the user has not given the argument "-check". Otherwise, no code is generated.

```
begin in_sert;

set command value to "insert ";
call advance subroutine to get next token;
if token ^= "field" then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= "(" then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= integer then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= ")" then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= "=" then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= double quote then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
add token to command value;
call advance subroutine to get next token;
if token ^= double quote then call error subroutine;
```

```
else add token to command value;
call advance subroutine to get next token;
if token ^= double quote then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
add token to command value;
call advance subroutine to get next token;
if token ^= double quote then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= period then call error subroutine;
else add token to command value;
if out.switch on then write command to outfile;
if not "-check" then generate code for PL/1 program;
end in_sert;
```

5.9 de_lete

Function: call de_lete (in_line, line_len, command_index, char_pos)

Parameters:

in_line	input command(s) line
line_len	length of input command(s) line
command_index	beginning character position of insert command in input command(s) line
char_pos	position in command line of current character under consideration

Description:

De_lete parses a delete command to verify that the syntax of the delete command is correct. Once the parsing is complete, code is generated for the saved PL/1 program if the user has not given the argument "-check". Otherwise, no code is generated.

```
begin de_lete;

set command value to "delete ";
call advance subroutine to get next token;
if token ^= "field" then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= "(" then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= integer then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= ")" then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= "=" then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= double quote then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
add token to command value;
call advance subroutine to get next token;
if token ^= double quote then call error subroutine;
```

```
else add token to command value;
call advance subroutine to get next token;
if token ^= period then call error subroutine;
else add token to command value;
if out.switch on then write command to outfile;
if not "-check" then generate code for PL/1 program;
end de_lete;
```

5.10 mo_ve

Function: call mo_ve (in_line, line_len, command_index, char_pos)

Parameters:

in_line	input command(s) line
line_len	length of input command(s) line
command_index	beginning character position of insert command in input command(s) line
char_pos	position in command line of current character under consideration

Description:

Mo_ve parses a move command to verify that the syntax of the move command is correct. Once the parsing is complete, code is generated for the saved PL/1 program if the user has not given the argument "-check". Otherwise, no code is generated.

```
begin mo_ve;
```

```
set command value to "move ";
call advance subroutine to get next token;
if token ^= "field" then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= "(" then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= integer then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= ")" then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= "=" then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
if token ^= double quote then call error subroutine;
else add token to command value;
call advance subroutine to get next token;
add token to command value;
call advance subroutine to get next token;
if token ^= double quote then call error subroutine;
else add token to command value;
```

```
call advance subroutine to get next token;
if token ^= "to" then call error subroutine;
else add token to command value;
call advance subroutine for next token;
if token ^= "head" | "tail" then call error subroutine;
else add token to command value;
call advance subroutine for next token;
if token ^= period then call error subroutine;
else add token to command value;
if out.switch on then write command to outfile;
if not "-check" then generate code for PL/1 program;
end mo_ve;
```

5.11 ext_ract

Function: call ext_ract (in_line, line_len, number_of_keywords, command_index, char_pos, keyword_table)

Parameters:

in_line	input command(s) line
line_len	length of input command(s) line
number_of_keywords	number of keywords in each vector in the data file
command_index	beginning character position of extract command in input command line
char_pos	position in command line of current character under consideration
keyword_table	pointer to table containing keywords of the vectors in the data file

Description:

Ext_ract parses an extract command to verify that the syntax of the command is correct. Once the parsing is done, code is generated for the save PL/1 program if the user has not specified the argument "-check" when giving cts arguments. If the argument "-check" was given, then no code is generated for the PL/1 program.

```
begin ext_ract;

set command value to "extract ";
call advance subroutine to get token;
call bool_term;
if advance_flag on then call advance subroutine for next token;
else add token to command value;
call advance subroutine to get token;
if token ^= "class" then call error subroutine;
else add token to command value;
call advance subroutine to get token;
if token ^="=" then call error subroutine;
else add token to command value;
call advance subroutine to get token;
if token ^= double quote then call error subroutine;
else add token to command value;
call advance subroutine to get token;
```

```
add token to command value;
call advance subroutine to get token;
if token ^= double quote then call error subroutine;
else add token to command value;
call advance subroutine to get token;
if token ^= period then call error subroutine;
else add token to command value;
if out.switch on then write command to outfile;
if "-check" switch off then generate code PL/1 program;
end ext_ract;
```

5.12 bool_term

Function: call bool_term (in_line, line_len, number_of_keywords, token, token_index, token_length, char_pos, keyword_table, advance_flag, keyword_table_code, command)

Parameters:

in_line	input command(s) line
line_len	length of input command(s) line
number_of_keywords	number of keywords in each vector in the data file
token	a word or character of the command line
token_index	beginning character position of token in command line
token_length	length of token
char_pos	position in command line of current character under consideration

keyword_table	pointer to table containing keywords of the vectors in the data file
advance_flag	determines whether or not to call the subroutine "advance" to get the next token
keyword_table_code	element number of keyword in keyword array
command	input line specifying the type of editing to be done on the data file

Description:

Bool_term is a recursive module which continues the processing of an extract command. It calls the module bool_fac, checks for the token "|", and if it is found then bool_term calls itself. If it is not found then bool_term ends.

```
begin bool_term;

if "-debug" switch on then print subroutine name;
call bool_fac;
if advance_flag then call advance subroutine for next token;
if token = "|" then do;
```

```
     add token to command value;
     call advance subroutine to get token;
     call bool_term;
end;
else advance_flag is off;
end bool_term;
```

5.13 bool_fac

Function: call bool_fac (in_line, line_len, number_of_keywords, token, token_index, token_length, char_pos, keyword_table, advance_flag, , keyword_table_code, command)

Parameters:

in_line	input command(s) line
line_len	length of input command(s) line
number_of_keywords	number of keywords in each vector in the data file
token	a word or character of the command line
token_index	beginning character position of token in command line
token_length	length of token
char_pos	position in command line of current character under consideration

keyword_table	pointer to table containing keywords of the vectors in the data file
advance_flag	determines whether or not to call the module "advance" to get the next token
keyword_table_code	element number of keyword in keyword array
command	input line specifying the type of editing to be done on the data file

Description:

This module continues processing of an extract command. It calls the module bool_pri, then calls itself if the token "&" is found, otherwise it exits.

```
begin bool_fac;
if "-debug" switch on then print subroutine name;
call bool_pri;
if advance_flag then call advance subroutine for next token;
if token = "&" then do;
     add token to command value;
     call advance subroutine to get token;
     call bool_fac;
end;
```

```
else advance_flag is off;
end bool_fac;
```

5.14 bool_pri

Function: call bool_pri (in_line, line_len, number_of_keywords, token, token_index, token_length, char_pos, keyword_table, advance_flag, keyward_table_code, command)

Parameters:

in_line	input command(s) line
line_len	length of input command(s) line
number_of_keywords	number of vector keywords in the data file
token	a word or character of the command line
token_index	beginning character position of token in command line
token_length	length of token
char_pos	position in command line of current character
keyword_table	pointer to table of vector keywords in the data file

advance_flag	determines whether or not to call the module "advance" to get the next token
keyword_table_code	element number of keyword in keyword array
command	input line specifying the type of editing to be done on the data file

Description:

Bool_pri continues processing an extract command. It calls advance for tokens of the command line, and verifies that they are in the correct format.

```
begin bool_pri;

if "_debug" then print subroutine name;
if conversion error then do;
     notify user of conversion error and
     that processing of current command  is terminated;
     reset onsource value;
     reset token;
     set error_flag_g;
```

```
end;
if token="^" then do;
     add token to command value;
     call advance subroutine for next token;
end;
if token="(" then do;
     add token to command value;
     call advance subroutine for next token;
     call bool_term;
     if advance_flag call advance subroutine for next token;
     if token^=")" then call error subroutine;
     else add token to command value;
end;
else do;
     do while (^okay);
          if token ^= "data" | "field" then do;
               if token = "keyword" then do;
                    add token to command value;
                    call advance subroutine for next token;
                    if token ^= "(" then call error subroutine;
                    else add token to command value;
                    call advance subroutine for next token;
                    if token ^= integer then call error subroutine;
                    else add token to command value;
                    call advance subroutine for next token;
```

```
                    if token ^= ")" then call error subroutine;
                    else add token to command value;
               end;
               else loop through keyword table for token keyword;
                    if it is, add token to command value;
               end;
          end;
          else do;
               if token = "data" then add token to command value;
               else if token = "field" then do;
                    add token to command value;
                    call advance subroutine for next token;
                    if token ^= "(" then call error subroutine;
                    else add token to command value;
                    call advance subroutine for next token;
                    if token ^= integer then call error subroutine;
                    else add token to command value;
                    call advance subroutine for next token;
                    if token ^= ")" then call error subroutine;
                    else add token to command value;
               end;
          end;
     end;
     add token to command value;
     if first character in token = integer then do;
```

```
call advance subroutine for next token;
if token ^= integer then call error subroutine;
else add token to command value;
call advance subroutine for next token;
if token = "," then do;
     add token to command value;
     call advance subroutine for next token;
     if token ^= integer then call error subroutine;
     else add token to command value;
end;
else set switch advance subroutine_flag off;
if advance subroutine_flag on then get next token;
if token ^= a relational operator then call error subroutine;
else add token to command value;
call advance subroutine for next token;
if token ^= a double quote then do;
     1185a.in -5
     end;
     see if token = number;
     if it does, add token to command value;
     else conversion condition occurs;
end;
else do;
     if token ^= double quote then call error subroutine;
     else add token to command value;
```

```
                call advance subroutine for next token;
                add token to command value;
                call advance subroutine for next token;
                if token ^= a double quote then call error subroutine;
                else add token to command value;
            end;
        end;
    end;
end bool_pri;
```

5.15 get_char

Function: call get_char (Char_char, Char_pos, Char_index, to_ken, in_line, line_len)

Parameters:

Char_char	individual character from input command line
Char_pos	position of character in input line
Char_index	index of character in table
to_ken	a word or character of the input line
in_line	input command(s) line
line_len	length of input command(s) line

Description:

Get_char is a module which returns the individual characters from the input command line to the module advance.

```
begin get_char;

if "-debug" then print subroutine name;
if end of file occurs do;
     set in_eof switch on;
     set command_flag switch on;
end;
if token ^="end" & Char_pos=line_len then turn command_flag on;
if command_flag on then return;
increment Char_pos by 1;
get Character;
get Char_index;
end get_char;
```

section

Function: call advance (to_ken, len_gth, place, in_line, line_len, char_pos, token_switch)

Parameters:

to_ken	word or character of input command line
len_gth	length of token
place	beginning character position of token in input command(s) line.
in_line	input command(s) line
line_len	length of input command(s)
Char_pos	position of character in input line
token_switch	switch indicating a string is to be enclosed within quotes

Description:

This module returns a token, its length, and its position in the input line to other parsing modules.

```
begin advance;
if "-debug" then print subroutine name;
if token_switch on do;
     reset to_ken;
     reset place and len_gth;
     reset token1;
     reset token2;
     if token1 is a null string then turn token_switch off;
end;
do while (need_char);
call get_char for next character;
if command_flag switch on then turn need_char off;
if period_flag & char ^="." then do;
     reset to_ken;
     reset len_gth and place;
     turn need_char off;
end;
else if char^=" " then do;
     if char is a letter then do;
          do while (alpha_char).
               add char to to_ken value;
```

```
                reset len_gth and place;
                call get_char for next character;
                if command_flag on then turn alpha_char switch off;
                if char is not a letter then turn alpha_char off;
           end;
           if command_flag on then turn need_char off;
           else if char=" " then turn need_char off;
           else do;
                decrement char_pos by 1;
                turn need_char off;
                reset place;
           end;
      end;
      else if char=operators then do;
           add character to to_ken value;
           reset place and len_gth;
           call get_char for next character;
           if char=operators then do;
                add character to to_ken value;
                reset len_gth and place;
           end;
           else do;
                decrement char_pos by 1;
                reset place;
           end;
```

```
          turn need_char off;
     end;
     else if char=caret then do;
          add char to to_ken value;
          reset len_gth and place;call get_char for next character;
          if command_flag then turn need_char off;
          if char=operators then do;
               add char to to_ken value;
               reset place and len_gth;
               call get_char for next char;
               if char=operations then do;
                    add char to to_ken value;
                    reset len_gth and place;
               end;
               else do;
                    decrement char_pos by 1;
                    reset place;
               end;
               turn need_char off;
          end;
          else do;
               decrement char_pos by 1;
               reset place;
               turn need_char off;
          end;
```

```
    end;
    else if char=paren then do;
         add char to to_ken value;
         reset place and len_gth;
         turn need_char off;
    end;
    else if char=comma then do;
         add char to to_ken value;
         reset len_gth and place;
         turn need_char off;
    end;
    else if char=double quote then do;
         add char to to_ken value;
         reset len_gth and place;
         if no_string switch on then do;
              turn token_switch off;
              turn need_char off;
         end;
         else if ^ no_string then do;
              do while (more);
                   call get_char for next char;
                   if command_flag on turn need char off;
                   else add char to to_ken value;
                   if char=double quote then do;
                        reset token1;
```

```
                    reset token2;
                    turn more off;
                    turn token_switch on;
                end;
            end;
            if more is off then turn need_char off;
        end;
    end;
    else if char=signs or digits then do;
        do while (need_number);
            add char to to_ken value;
            reset len_gth and place;
            call get_char for next char;
            if command_flag then turn need_number off;
            if char ^=digits then do;
                if char ^="e" then do;
                    if char ^=signs then do;
                        if char ^=period then do;
                            add char to to_ken value;
                            call get_char for next char;
                            if command_flag on
                                then turn need_number off;
                                in -5
                                if char ^=digits then do;
                                    reset to_ken,
```

```
                                        len_gth, and place;
                                        turn need_number off;
                                   end;
                                   else do;
                                        reset to_ken,
                                        length, and place;
                                   end;
                              end;
                              else if char=" " then do;
                                   decrement char_pos by 1;
                                   turn need_number off;
                              end;
                              else do;
                                   decrement char_pos by 1;
                                   reset place;
                                   turn need_number off;
                              end;
                         end;
                    end;
               end;
          end;
          if need_number is off then turn need_char off;
     end;
     else if char=period then do;
          add char to to_ken value;
```

```
                reset len_gth and place;
                call get_char for next char;
                if command_flag on then turn need_char off;
                if char ^=digits then do;
                     decrement char_pos by 1;
                     reset place;
                     turn need_char off;
                end;
                else if char=digits then do;
                     add char to to_ken value;
                     reset len_gth and place;
                end;
           end;
           else if char=specials then do;
                add char to to_ken value;
                reset len_gth and place;
           end;
           else if char=logics then do;
                add char to to_ken value;
                reset len_gth and place;
                turn need_char off;
           end;
           else call error subroutine;
      end;
end;
```

```
end advance;
```

5.17 error

Description:

This module notifies the user of his errors and makes or obtains corrections. It is only in this module that the user can use "cease" or "quit" in terminating commands.

Usage:

call error (error_code, token, token_index, token_length, in_line, char_pos, token_switch, line_len)

Arguments:

error_code	number of error message in error table
token	word or character from input command(s) line
token_index	beginning character position of token in input command(s) line
token_length	length of token
in_line	input line

char_pos	position of character in input command(s) line
token_switch	switch indicating that a string is to be enclosed within quotes
line_len	length of input command(s) line

Performance:

```
begin error;
if "-debug" then print subroutine name;
print command line;
if error that requires a reply then do;
     if "-noquery" then do;
          print error message;
          notify user that processing of current command is terminated
          and processing of next command will begin;
          turn is_correct and error_flag_q on;
     end;
     else call cts_query_user to print error_message and get reply;
end;
else if error not requiring a reply then do;
     print error message;
```

```
     set input line to null string;
end;
do while (^is_correct);
     if input line ^= null string then do;
          if input line = "cease" or "quit" then do;
               turn appropriate error_flag on;
               turn is_correct flag on;
          end;
          else for error_code = 1,6,8, or 12 then do;
               add reply to command line;
               reset line_len,token,token_length,and char_pos;
          end;
          else for error_code = 13 do;
               add reply to command line;
               reset line_len and char_pos;
          end;
          else for error_code = 14 the do;
               set command line = reply;
               reset line_len;
               turn is_correct switch on;
          end;
          else for error_code = 15 do;
               add reply to command line;
               reset token,token_length,and char_pos;
          end;
```

```
        else do;
             print new command line;
             do while (^okay);
                  ask user if okay;
                  if yes then turn okay_switch on;
                  else if no then do;
                       call cts_query_user for "cease" or "quit";
                       turn okay_switch on;
                  end;
             end;
             if input line = "yes" then turn is_correct switch on;
        end;
   end;
   else do;
        if error_code = 2,3,4,5,7,9,10,11, or 13 then do;
             set token to error-supplied correction;
             reset token_length,line_len,in_line,and char_pos;
             if error_code = 2 or 9 then reset token_index;
             if error_code = 2 then turn period_flag off;
             if error_code = 3,4,or 9 then turn token_switch off;
        end;
        if not "-noquery" then do;
        print new command line;
        ask user if okay;
        if yes then turn is_correct switch on;
```

```
          else if no then do;
                    call cts_query_user to ask user to enter "cease" or "quit";
               end;
          end;
     end;
end;
if error_flag on then return;
determine length of old command line and corrected command line;
if corrected line shorter, then pad with blanks;
end error;
```

5.18 cts_query_user

Description:

This module is used instead of MULTICS command_query_ to remove system dependence. It displays a question or statement to the user and retrieves any reply.

Usage:

```
call cts_query_user (query, echo, noquery)
```

Arguments:

query	question or statement to be displayed to user
echo	switch determining whethere or not to echo the input line back to the terminal - same as "-echo" argument
noquery	switch indicating if in batch or interactive mode - same as "-noquery" argument

Performance:

```
begin cts_query_user;
if not "-noquery" then print question or statement;
get reply;
if "-echo" then print reply;
end cts_query_user;
```

REMOTE DATA ENTRY
TABLE OF FIGURES

REFERENCES

[1] "MULTICS Programmers' Manual (MPM) - Reference Guide" (AG91)

[2] "MPM - Commands" (AG92)

[3] "MPM - Subroutines" (AG93)

[4] "MPM - Subsystems Writers' Guide" (AK92)

[5] "Tektronix 4014 and 4014-1 Computer Display Terminal - Users Introduction Manual (1974)"

[6] "Tektronix 4921/4922 Flexible Disk Memory - Users Instruction Manual (1976)"

[7] "Tektronix 4921/4922 Flexible Disk memory Unit - Service Instruction Manual (1975)"

[8] "Tektronix Digital Cartridge Tape Recorder - Users Instruction Manual (1975)"

[9] Perry, C., "Conversion Between Floating Point Representations", Communications of the ACM, Vol. 3, No. 6, p. 352, March 1960.

[10] Knuth, Donald, "The Art of Computer Programming, Volume 2"

(1973).

[11] Aho, Alfred V. and Ullman, Jeffrey D., Principles of Compiler Design, Addison-Wesley Publishing Co., 1977.

[12] Gries, David, Compiler Construction for Digital Computers, John Wiley & Sons,Inc., 1971.

[12] Gries, David, "The use of transition matrices in compiling", Communications of the ACM 2 (Feb. 1968) 26-34.

[14] Lewis II, Philip M., Rosenkrantz, Daniel J. and Stearns, Richard E. Compiler Design Theory, Addison-Wesley Publishing Co., Inc., 1976.

[15] Lyon, G., "Syntax directed least errors analysis for context free languages, a practical approach", Communications of the ACM 17, (Jan. 1974) 3-14.

[16] Wirth, Niklaus, Algorithms + Data Structures = Programs, Prentice-Hall, Inc., 1976.